

TECNOLOGIE WEB

A.Q.

Indice

1 Introduzione	...4
2 URI e URL	...5
3 HTTP	...6
3.1 Messaggi HTTP	
3.2 Modelli Server	
3.3 Cookie, autenticazione e sicurezza	
3.4 Architetture per il Web	
4 HTML	...12
4.1 Tag	
4.2 Attributi e standard MIME	
4.3 Struttura base di un HTML	
4.4 DOM	
5 CSS	...25
5.1 Selettori e raggruppamenti	
5.2 Proprietà e valori	
5.3 Ereditarietà e cascade	
5.4 Colori e testi	
5.5 Box Model	
5.6 Liste	
5.7 Display	
5.8 Posizionamento	
5.9 Tabelle	
6 Web Dinamico	...34
6.1 Modello a contenimento	
6.2 Stato e sessione	
6.3 Architettura nei sistemi web	
7 Servlet	...39
7.1 Ciclo di vita	
7.2 Metodi	
7.3 Deployment	
7.4 Gestione dello stato di sessione	
7.5 Scoped Objects	
7.6 Inclusione di risorse	
8 JSP	...56
8.1 Sintassi e tag	
8.2 Built-in objects	
8.3 Azioni	
8.4 Modello a componenti	
9 JavaScript	...68
9.1 Variabili e Oggetti	
9.2 Funzioni e Costruttori	

9.3 Operatori e Istruzioni	
9.4 Note su <script>	
9.5 Utilità di JavaScript	
9.6 Eventi	
9.7 Muoversi nel DOM	
9.8 Form	
10 AJAX	...81
10.1 XMLHttpRequest	
10.2 JSON	
10.3 Schemi di interazione	
11 React.js	...92
11.1 Concetti base	
11.2 JSX	
11.3 Componenti	
11.4 Gestione eventi	
12 Tecnologie Server-side	...101
12.1 J2EE	
12.2 EJB	
12.2.1 EJB Componenti	
12.2.2 Servizi Container-based	
12.3 Spring	
12.3.1 Dependency Injection	
12.3.2 Bean Factory	
12.3.3 Injection e Bean	
12.3.4 Spring MVC	
13 Node.js	...112
13.1 Event loop	
13.2 Moduli Node.js	
13.3 Listener/Emitter	
13.4 Stream	
13.5 TCP Networking	
14 JSF e Web socket	...119
14.1 JSF	
14.2 Ciclo di vita di un'applicazione Facelets	
14.3 JSF e Templating	
14.4 Navigation Rule	
14.5 WebSocket	
14.6 Caratteristiche WS	
14.7 WebSocket API server	
14.8 WebSocket API client (Javascript)	

1 Introduzione

Il World Wide Web (WWW) è stato proposto nel 1989 da Tim Berners-Lee. L'idea alla base del progetto era quella di fornire strumenti adatti a condividere documenti statici in forma ipertestuale disponibili su rete Internet tramite protocollo semplice e leggero.

http/1.0 → 1996

http/1.1 → 1997-99

http/2.0 → 2015

Un **ipertesto** (hypertext) è un insieme di documenti messi in relazione da link monodirezionali.

Può essere visto come una rete (un grafo) dove i documenti ne costituiscono i nodi.

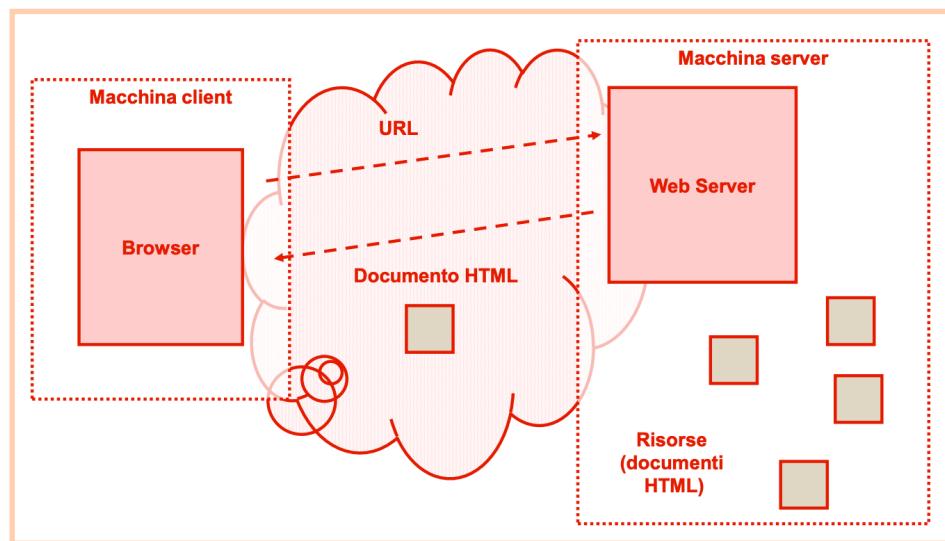
La caratteristica principale è che la lettura può svolgersi in maniera non lineare.

Se si prendono in considerazione non solo testi ma elementi multimediali (immagini suoni, video) si parla di **ipermedia**.

Idee tecniche di base al WWW:

<i>Tre elementi concettuali:</i> <ul style="list-style-type: none">• Meccanismo di locazione di un documento (URL)• Protocollo per accedere alle risorse del documento (HTTP)• Linguaggio per descrivere i documenti (HTML)	<i>Due elementi "fisici":</i> <ul style="list-style-type: none">• SERVER eroga risorse• CLIENT visualizza e naviga <p>(Client attivi – Server Passivi)</p>
---	---

WWW = URL + HTTP + HTML



2 URI e URL

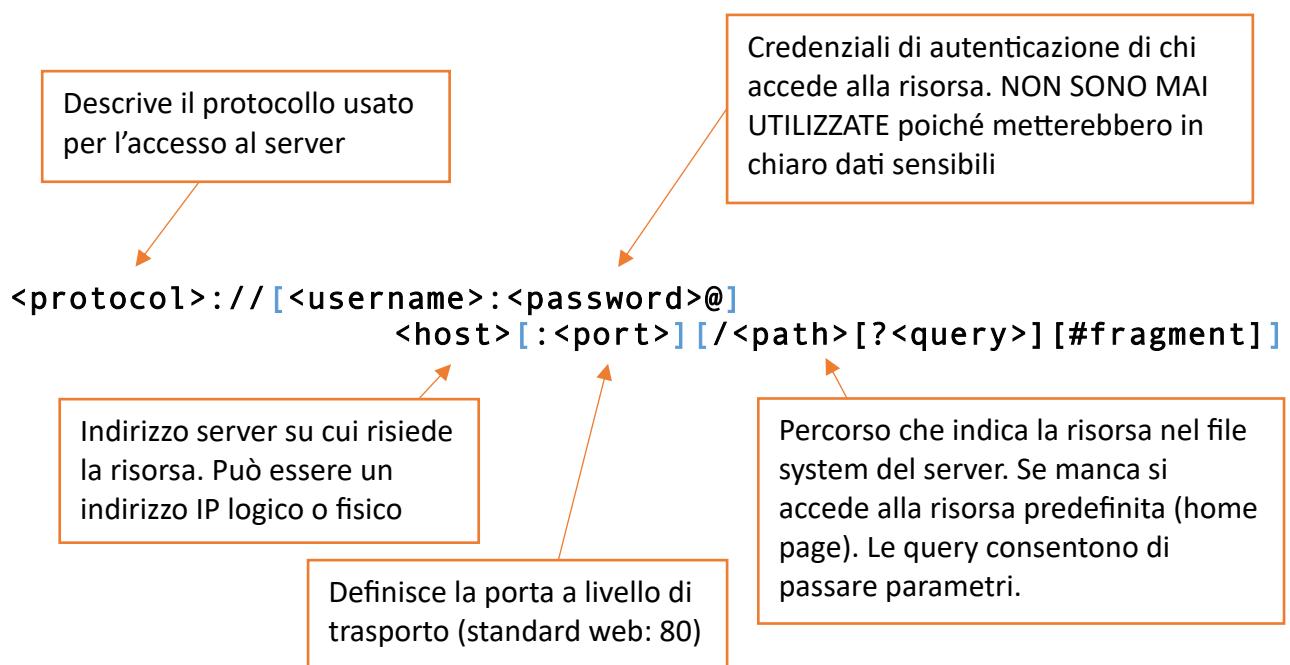
Gli **URI**, Uniform Resource Identifier, possono essere visti come degli identificatori standard di risorse. Sono stringhe con una sintassi definita, dipendente dallo schema scelto.

<schema> : <scheme-specific-part>

Il concetto di URI è molto generale. Ne derivano due specializzazioni del concetto...

- **URN**, *Uniform Resource Name*: identifica una risorsa per mezzo di un “nome” che deve essere globalmente unico e restare valido anche se la risorsa diventa non disponibile o cessa di esistere.
- **URL**, *Uniform Resource Locator*: identifica una risorsa per mezzo del suo meccanismo di accesso primario (es. locazione nella rete) piuttosto che sulla base del suo nome o dei suoi attributi.

Un URL specifica esplicitamente il server che fornisce la risorsa specificata. Indica inoltre il protocollo necessario per il trasferimento della risorsa stessa. Nella sua forma più comune segue una sintassi del tipo:



`https://virtuale.unibo.it:8080/Courses/index.html`

Notiamo che le URI possono essere classificate come:

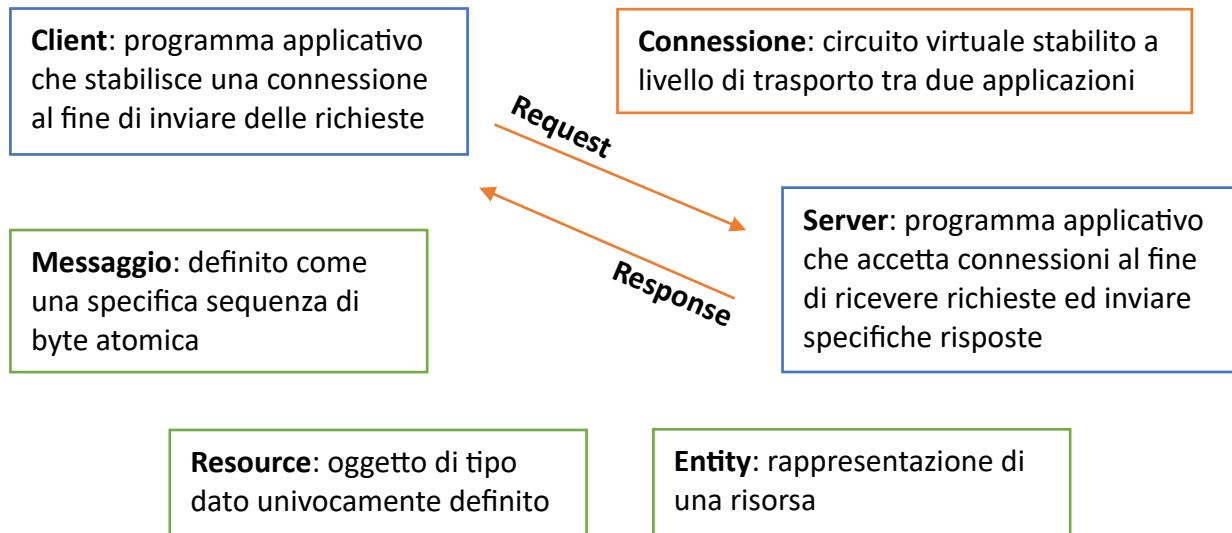
OPACA: non è soggetta a ulteriori operazioni di parsing.

GERARCHICA: è soggetta a ulteriori operazioni di parsing, per esempio per separare l'indirizzo del server dal percorso all'interno file system.

3 HTTP

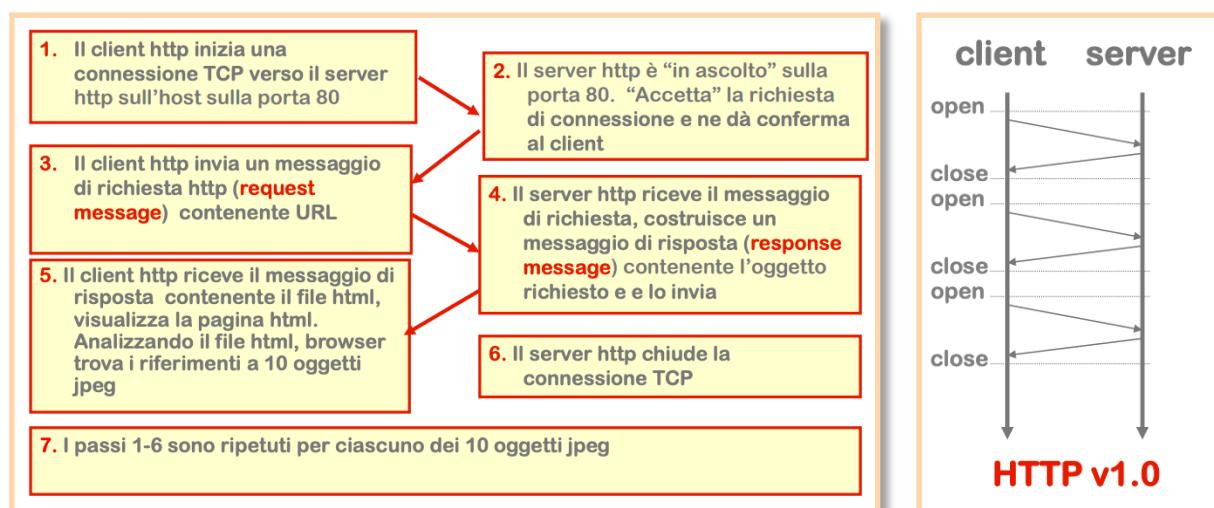
HTTP è l'acronimo di Hyper Text Transfer Protocol; è il protocollo di livello applicativo (livello 7) utilizzato per trasferire le risorse Web. Ha il compito di gestire sia le richieste che le risposte secondo un approccio **stateless**: né il cliente né il server mantengono informazioni relative ai messaggi precedentemente scambiati.

Definiamo...



HTTP nasce come protocollo basato su TCP, infatti sia richieste che risposte sono trasmesse usando stream TCP. Il server rimane in ascolto in modo passivo (tipicamente sulla porta 80) fino a quando un client apre una connessione. L'interazione viene ideata come one-shot: per ogni connessione è previsto un solo scambio.

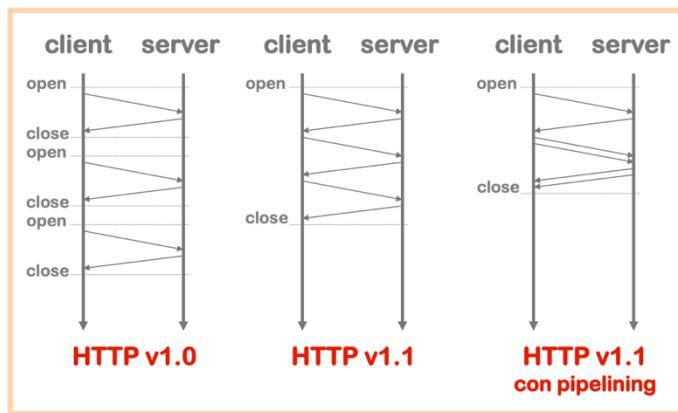
Ipotizziamo di voler richiedere una pagina composta da un file HTML e 10 immagini... per HTTPv1.0...



La differenza principale tra HTTP v1.0 e v1.1 è la possibilità di specificare coppie multiple di richiesta e risposta nella stessa connessione; ragion per cui le connessioni **1.0** vengono dette **non persistenti** mentre quelle **1.1** vengono definite **persistenti**.

Il server HTTP chiude la connessione quando viene specificato nell'header del messaggio (desiderata da parte del cliente) oppure quando non è usata da un certo tempo.

Per migliorare ulteriormente le prestazioni si può usare la tecnica del **pipelining** che consiste nell'invio di molteplici richieste da parte del client prima di terminare la ricezione delle risposte. Le risposte debbono però essere date nello stesso ordine delle richieste, poiché non è specificato un metodo esplicito di associazione tra richiesta e risposta.



HTTPv2 nasce con l'obiettivo di migliorare la performance complessiva con full backward compatibility con HTTPv1.1. HTTPv2 è basato su SPDY, protocollo di cosiddetto open networking promosso da Google.

Esiste anche HTTPv3: miglioramento di HTTP tramite integrazione con protocollo di trasporto QUIC (invece del classico TCP).

3.1 Messaggi HTTP

Tornando all'HTTP tradizionale... un messaggio è definito da due strutture:

Message Header: contiene tutte le informazioni necessarie per identificazione del messaggio.

Message Body: contiene i dati trasportati dal messaggio.

Esistono schemi precisi (standard, definiti e non modificabili) per ogni tipo di messaggio relativamente a header e body. I dati sono codificati secondo il formato specificato nell'header, solitamente sono in formato MIME.

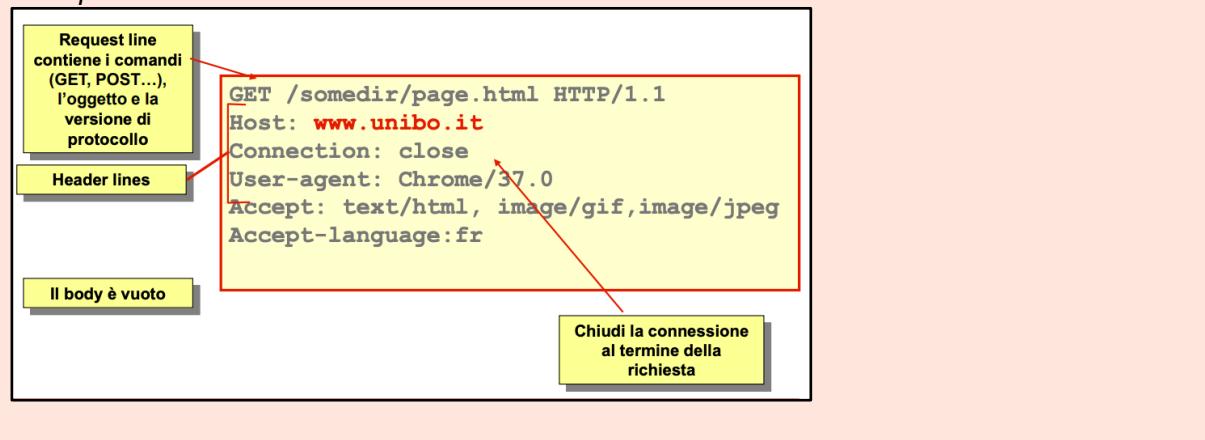
Gli **header** sono costituiti da insiemi di coppie **[nome: valore]** che specificano caratteristiche del messaggio trasmesso o ricevuto.

Il protocollo utilizza messaggi in formato ASCII.

- **Header generali della trasmissione**
 - Data, codifica, versione, tipo di comunicazione, ecc.
- **Header relativi all'entità trasmessa**
 - Content-type, Content-Length, data di scadenza, ecc.
- **Header riguardo la richiesta effettuata**
 - Chi fa la richiesta, a chi viene fatta la richiesta, che tipo di caratteristiche il client è in grado di accettare, quale autorizzazione, ecc.
- **Header della risposta generata**
 - Che server dà la risposta, che tipo di autorizzazione è necessaria, ecc.

Richiesta

Esempio...



Analizziamo i possibili **comandi** della richiesta:

- GET

Serve per richiedere una risorsa ad un server. È il metodo più frequente: 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. È previsto il passaggio di parametri (da parte dell'URL).

La lunghezza massima di un URL è limitata.

- POST

Progettato come messaggio per richiedere una risorsa. A differenza di GET, i dettagli per identificazione ed elaborazione della risorsa stessa non sono nell'URL, ma sono contenuti nel body del messaggio. Non ci sono limiti di lunghezza nei parametri di una richiesta.

- PUT

Chiede la memorizzazione sul server di una risorsa all'URL specificato. Il metodo PUT serve quindi per trasmettere delle informazioni dal client al server.

- DELETE

Richiede la cancellazione della risorsa riferita dall'URL specificato.

- HEAD

Simile al metodo GET, ma il server deve rispondere soltanto con gli header relativi, senza body.

- OPTIONS

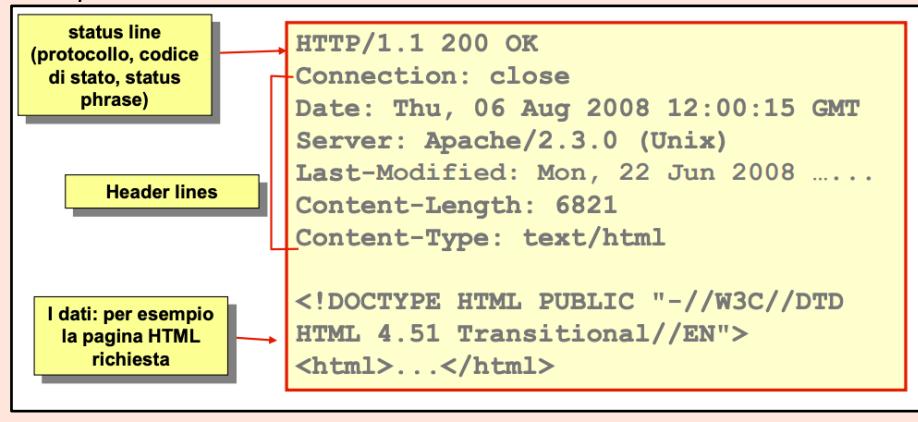
Serve per richiedere informazioni sulle opzioni disponibili per la comunicazione.

- TRACE

Usato per invocare il loop-back remoto a livello applicativo del messaggio di richiesta. Consente al client di vedere che cosa è stato ricevuto dal server.

Risposta

Esempio...



Analizziamo i possibili **codici** di stato, un numero di tre cifre, di cui la prima indica la classe della risposta e le altre due la risposta specifica:

- **1xx Informational**

Una risposta temporanea alla richiesta, durante il suo svolgimento.

- **2xx Successful**

Il server ha ricevuto, capito e accettato la richiesta.

- **3xx Redirection**

Il server ha ricevuto e capito la richiesta, 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**

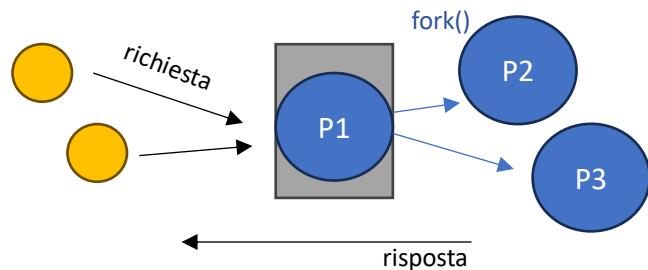
La richiesta può anche essere corretta, ma il server non è in grado di soddisfare la richiesta per un problema interno (suo o di applicazioni CGI).

- **100 Continue** (se il client non ha ancora mandato il body, deprecated da HTTPv1.0)
- **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 CGI mal fatto)
- **501 Not implemented** (metodo non conosciuto dal server)

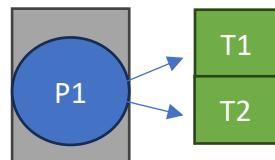
3.2 Modelli server

Il modello server da adottare per l'HTTP non è specificato come standard ma è facile definirne le caratteristiche in base a quello che abbiamo visto. Il server NON deve essere sicuramente mono sequenziale in quanto è necessaria la possibilità di servire più processi.

In una delle prime versioni progettate un processo in ascolto produceva processi pesanti dedicati a ogni richiesta tramite principio fork.

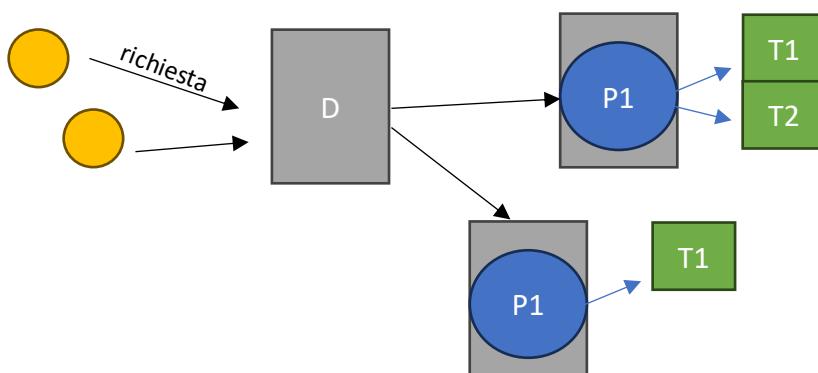


I server web moderni prediligono un singolo processo pesante in ascolto in grado di dedicare ad ogni richiesta un thread. Spesso i server predispongono un numero n di thread, detti "idol", già preesistenti i quali una volta servito un cliente non si distruggono ma rimangono in attesa di un nuovo compito.



Se le richieste superano il numero n di thread possiamo o accodare le richieste o creare un nuovo processo leggero.

Per rendere la soluzione scalabile si associa a più server web un'unica scheda di rete endpoint che riceve le richieste e le ridistribuisce ai server associati (Dispatcher). Grazie all'approccio stateless qualsiasi macchina può servire una qualsiasi richiesta.



3.3 Cookie, autenticazione e sicurezza

I **cookie** sono una struttura dati che si muove come un token dal client al server e viceversa.

Dopo la loro creazione vengono sempre passati ad ogni trasmissione di request e response; hanno come scopo quello di fornire un supporto per il mantenimento di stato in un protocollo come HTTP che è essenzialmente stateless.

Sono una collezione di stringhe contenenti varie informazioni come...

- Key: identifica univocamente un cookie all'interno di un dominio:path
- Value: valore associato al cookie (è una stringa di max 255 caratteri)
- Path: posizione nell'albero di un sito al quale è associato (di default /)
- Domain: dominio dove è stato generato...

...altro

L'**autenticazione** può essere un aspetto fondamentale qualora si voglia restringere l'accesso alle risorse ai soli utenti abilitati. Basare l'autenticazione sull'indirizzo IP del client è una soluzione che presenta vari svantaggi: non funziona se l'indirizzo non è pubblico (vedi esempio dei NAT), non funziona se l'indirizzo IP è assegnato dinamicamente (es. DHCP), esistono tecniche che consentono di presentarsi con un IP falso. Normalmente si usano **FORM** o **HTTP BASIC**.

La **sicurezza** di un canale va studiata a livello di trasporto (livello 4). Le proprietà desiderate sono:

- Confidenzialità: messaggio criptato, non leggibile da un ascoltatore
- Integrità: il messaggio deve raggiungere il destinatario in modo integro,
non deve essere modificato
- Autenticità: autenticità di chi manda il messaggio
- Non ripudio: valore legale del messaggio inviato [poco usato]

Due di questi canali sono SSL (Secure Sockets Layer) e TLS (Transport Layer Security).

Il TLS che sostituisce SSL usa crittografia asimmetrica basata su coppie chiave pubblica-privata.

3.4 Architetture per il Web

Definiamo...

→ **Proxy**: programma applicativo in grado di agire sia come Client che come Server al fine di effettuare richieste per conto di altri clienti. Un proxy interpreta, e se necessario, riscrive le request prima di inoltrarle.

→ **Gateway**: Server che agisce da intermediario per altri Server. Al contrario dei proxy, il gateway riceve le request come il server originale e i Client non sono in grado di identificare che response proviene da un gateway. (Detto anche reverse proxy).

→ **Tunnel**: programma applicativo che agisce come "blind relay" tra due connessioni. Una volta attivo (in gergo "salito") non partecipa alla comunicazione http.

→ **Web cache**: memorizza i documenti che la attraversano. L'obiettivo è usare i documenti in cache per le successive richieste qualora alcune condizioni siano verificate.

Tipi di Web cache:

User Agent Cache

Lo user agent (tipicamente il browser) mantiene una cache delle pagine visitate dall'utente.

Proxy Cache

Forward Proxy Cache: per ridurre le necessità di banda; il proxy intercetta il traffico e mette in cache le pagine.

Reverse Proxy Cache: gateway cache che operano per conto del server e consentono di ridurre il carico computazionale delle macchine. I client non sono in grado di capire se le pagine arrivano dal server o dal gateway.

HTTP definisce vari meccanismi che possono avere effetti collaterali positivi per la gestione «lazy» dell'aggiornamento cache...

Freshness, Validation, Invalidation.

4 HTML

HTML è l'acronimo di HyperText Markup Language, è utilizzato per descrivere le pagine che costituiscono i nodi dell'ipertesto. In generale, ogni documento elettronico è costituito da una stringa di caratteri, i quali vengono rappresentati all'interno di un elaboratore mediante codifica binaria. Per codificare i caratteri si stabilisce una corrispondenza biunivoca tra elementi di una collezione ordinata di caratteri e un insieme di codici numerici ottenendo così un *coded character set*.

Per ciascun *coded character set* si definisce una codifica dei caratteri (*character encoding*).

La codifica mappa una o più sequenze di byte a un numero intero che rappresenta un carattere in un determinato *coded character set*.

I più noti sono:

- ASCII (7 bit)
- Famiglia ISO 8859/ANSI (8 bit)
- Unicode (8, 16 o 32 bit: **UTF-8**, UTF-16 e UTF-32)

Standard che sta prendendo piede come successore di ASCII, specie da quando è diventato la codifica principale di **Unicode** per Internet da parte del consorzio **W3C**

Si parla propriamente di linguaggio di codifica testuale solo in riferimento ai linguaggi che consentono la rappresentazione o il controllo di uno o più livelli strutturali di un documento testuale. Tali linguaggi vengono correntemente denominati **linguaggi a marcatori** (mark-up language).

Un linguaggio di mark-up è composto da:

- un insieme di istruzioni dette **tag** o **mark-up** (marcatori) che rappresentano le caratteristiche del documento testuale
- una **grammatica** che regola l'uso del mark-up
- una **semantica** che definisce il dominio di applicazione

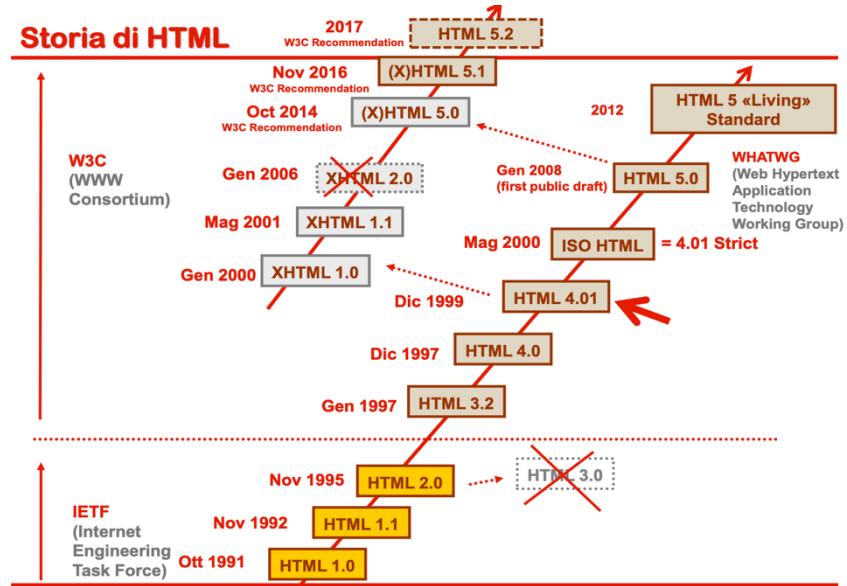
I marcatori vengono inseriti direttamente all'interno del testo cui viene applicato.

Osservazione:

Tradizionalmente i linguaggi di mark-up sono stati divisi in due tipologie:

linguaggi procedurali: il mark-up specifica quali operazioni un dato programma deve compiere su un documento elettronico per ottenere una determinata presentazione (Tex, LateX)

linguaggi dichiarativi: il mark-up descrive la struttura di un documento testuale identificandone i componenti (SGML, HTML, XML)

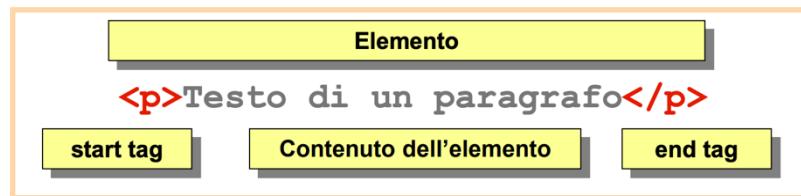


4.1 Tag

I **tag** HTML sono usati per definire il mark-up di elementi HTML. Sono preceduti e seguiti rispettivamente da due caratteri < > e di norma accoppiati.

Il testo tra start tag ed end tag è detto contenuto dell'elemento.

Un documento HTML contiene quindi elementi composti da testo semplice delimitato da tag.



I tag non sono case sensitive e l'apertura e chiusura di tag annidati può essere “incrociata” (<i>Testo</i>). Sono inoltre ammessi elementi senza chiusura.

Esistono però delle buone pratiche che è bene rispettare e che diventano un obbligo in una versione più rigorosa del linguaggio chiamata XHTML:

- Chiudere sempre anche i tag singoli:
</br>, o in forma sintetica

- Tag in minuscolo
- Apertura e chiusura senza incroci (in teoria non ammessi ma tollerati) (<i>Testo</i>)

HTML definisce un certo numero di entità (entity) per rappresentare i caratteri speciali senza incorrere in problemi di codifica.

<ul style="list-style-type: none"> ▪ Caratteri riservati a HTML (<, >, &, „, ecc.) ▪ Caratteri non presenti nell'ASCII a 7 bit
& ; & " ; " ;
< ; < ; > ; < ;
® ; ® ; ; (non-breaking space)
&Aelig; ; È ; Á ; Á ;
À ; À ; Ä ; Ä ;
æ ; æ ; á ; á ;
à ; à ; ä ; ä ;
ç ; ç ; ˜ ; ñ ;

4.2 Attributi e standard MIME

Gli **attributi** sono coppie “nome = valore” contenute nello start tag con una sintassi di questo tipo:

```
<tag attrib1='valore1' attrib2='valore2'>
```

I valori sono racchiusi da apici singoli o doppi, i quali possono essere omessi qualora il valore non contenga spazi.

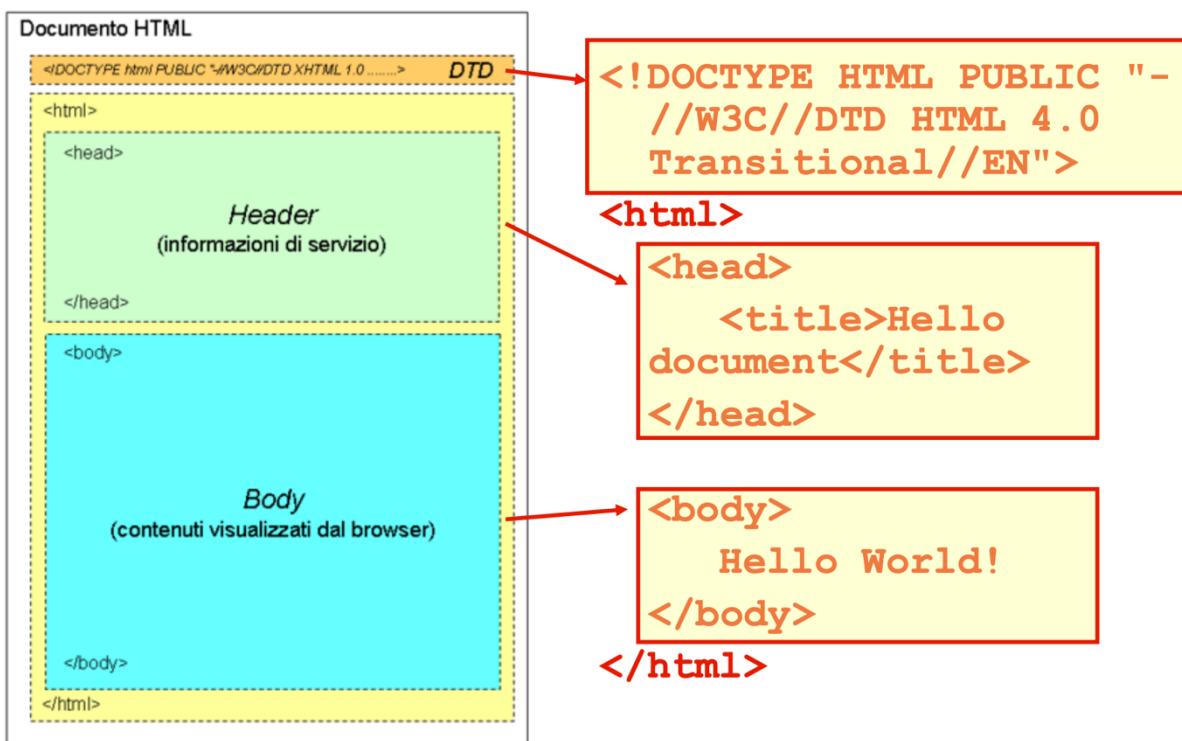
Lo standard **MIME**, oggi noto anche come Internet Media Type, rappresenta il tipo di contenuto di un messaggio. Classifica i tipi di contenuto sulla base di una logica a due livelli tramite sintassi:

tipo/sottotipo

text/plain: testo semplice

È possibile inserire **commenti** in qualunque punto all’interno di una pagina HTML seppur racchiusi tra: <!-- Commento -->

4.3 Struttura base di un HTML



[[Prova HTML!](#)]

DTD

Il primo elemento di un documento HTML è la definizione del tipo di documento (Document Type Definition o **DTD**). Serve al browser per identificare le regole di interpretazione e visualizzazione da applicare al documento

HEADER

L'header è identificato dal tag **<head>**; contiene elementi non visualizzati dal browser (informazioni di servizio).

<title>

titolo della pagina (viene mostrato nella testata della finestra principale del browser).

<meta>

metadati informazioni utili ad applicazioni esterne (es. motori di ricerca) o al browser (es. lingua, codifica dei caratteri utile per la visualizzazione di alfabeti non latini). Esistono due tipi di elementi meta, distinguibili dal primo attributo: **http-equiv** o **name**.

Gli elementi di tipo *http-equiv* danno informazioni al browser su come gestire la pagina.

<meta http-equiv=nome content=valore>

Gli elementi di tipo name forniscono informazioni utili ma non critiche.

<meta name=nome content=valore>

http-equiv=

→ **refresh**: indica che la pagina deve essere ricaricata dopo un numero di secondi definito dall'attributo content

→ **expires**: stabilisce una data scadenza (fine validità) per il documento

→ **content type**: definisce il tipo di dati contenuto nella pagina (di solito il tipo MIME text/html)

name=

→ **author**: autore della pagina.

→ **description**: descrizione della pagina.

→ **copyright**: indica che la pagina è protetta da un diritto d'autore.

→ **keywords**: lista di parole chiave separate da virgole, usate dai motori di ricerca.

→ **date**: data di creazione del documento

<base>

definisce come vengono gestiti i riferimenti relativi nei link.

<link>

collegamenti verso file esterni: CSS, script, icone visualizzabili nella barra degli indirizzi del browser.

<style>

Informazioni di stile (CSS locali).

Esempio:

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<meta name="description" content="Documentazione HTML">
<title>Impariamo HTML</title>
<link href="style.css" rel="stylesheet" type="text/css">
</head>
```

BODY

Il tag **<body>** delimita il corpo del documento; contiene la parte che viene mostrata dal browser.
Ammette diversi attributi tra cui...

→ **background** =uri Definisce l'URI di una immagine da usare come sfondo per la pagina

→ **text** =color Definisce il colore del testo [[140 nomi di colori](#)]

→ **bgcolor** =color In alternativa a background definisce il colore di sfondo della pagina

→ **lang** =linguaggio definisce il linguaggio utilizzato nella pagina (ES: language="it")

Esempio:

```
<body>
<h1>Titolo</h1>
<p>Questo &eacute; un paragrafo</p>
<hr>
<p>Ecco una lista!<br>Esempio:</p>
<ul>
<li>Pane</li>
<li>Latte</li>
</ul>
</body>
```

Titolo

Questo é un paragrafo

Ecco una lista!
Esempio:

- Pane
- Latte

Analizziamo i tipi di elementi del body:

- **Intestazioni**: titoli organizzati in gerarchia
- **Strutture di testo**: paragrafi, testo indentato, ecc.
- **Aspetto del testo**: grassetto, corsivo, ecc.
- **Elenchi e liste**: numerati, puntati
- **Tabelle**
- **Form** (moduli elettronici): campi di inserimento, checkbox e radio button, menu a tendina, bottoni, ecc.
- **Collegamenti ipertestuali e ancore**
- **Immagini e contenuti multimediali** (audio, video, animazioni, ecc.)
- **Contenuti interattivi**: script, applicazioni esterne

Dal punto di vista del layout della pagina gli elementi HTML si dividono in 3 grandi categorie:
elementi “block-level” che costituiscono un blocco attorno a sé, e di conseguenza «vanno a capo» (paragrafi, tabelle, form...), **elementi “inline”** che non vanno a capo e possono essere integrati nel testo (link, immagini,...) e **liste** numerate o puntate.

Un elemento block-level può contenere altri elementi dello stesso tipo o di tipo inline, mentre un inline può contenere solo altri elementi inline.

Un'altra distinzione da ricordare è quella tra elementi **rimpiazzati** (replaced elements) ed elementi **non rimpiazzati**. I rimpiazzati sono quelli di cui il browser conosce le dimensioni intrinseche.

Contenitori di testo

Heading <h1>

I tag <h1>, <h2>, ..., <h6> servono per definire dei titoli in ordine di importanza decrescente. Appaiono in grassetto e lasciano riga vuota.

Paragrafo <p>

Rappresenta l'unità base entro cui suddividere un testo (è un blocco). <p> lascia una riga vuota prima della sua apertura e dopo la sua chiusura. Se si vuole andare a capo:
.

È possibile definire l'allineamento di un paragrafo o un di un titolo mediante l'attributo **align** che può assumere i seguenti quattro valori:

- 1) align = left
- 2) align = center
- 3) align = right
- 4) align = justify

Blocco <div>

Usato spesso al posto di <p> in quanto <div> forma un blocco di testo che NON lascia spazi prima e dopo l'apertura.

Contenitore

 è un contenitore generico che può essere annidato. È un elemento inline, e quindi non va a capo ma continua sulla stessa linea del tag che lo include.

(! Se non viene associato ad uno stile è invisibile)

Horizontal rule <hr>

Il tag <hr> serve ad inserire una riga di separazione. Sono utili i seguenti attributi:

- **align** = {left | center | right} Allineamento della riga rispetto a ciò che la circonda
- **size** = pixels Altezza della riga
- **width** = length Larghezza della riga in modo assoluto o in percentuale
- **noshade** Riga senza effetto di ombreggiatura

HTML consente di definire lo stile di un frammento di testo, combinando fra loro anche più stili. I tag che svolgono questa funzione vengono normalmente suddivisi in fisici e logici:

Tag fisici: definiscono lo stile del carattere in termini grafici indipendentemente dalla funzione del testo nel documento

Tag logici: forniscono informazioni sul ruolo svolto dal contenuto, e in base a questo adottano uno stile grafico

Tag fisici di testo / Tag logici di testo

<tt></tt> Carattere monospaziato
<i></i> Corsivo
 Grassetto
<u></u> Sottolineato
<s></s> Testo barrato

 Visualizzato in grassetto
 Visualizzato in corsivo
<code>/<pre> -Codice- monospaziato
<kbd> -Keyboard- monospaziato
<abbr> Abbreviazione (nessun effetto)
<acronym> Acronimo (nessun effetto)
<address> Indirizzo (in corsivo)
<blockquote> Blocco di citazione (rientrato a destra)
<cite> Citazione (in corsivo)

Font

Il tag permette di formattare il testo, definendo dimensioni, colore, tipo di carattere.

Attributi:

- **size** = [+|-]n Definisce le dimensioni del testo (1-7 o relative)
- **color** = color Definisce il colore del testo [[140 nomi di colori](#)]
- **face** = text Definisce il font del testo

Esempio:

```
<body>
<h1><font color="blue">Titolo blu</font></h1>
<div><em>Blocco di testo corsivo</em></div>
<div align="center"><b>Blocco di testo grassetto</b></div>
<div align="right"><font size="1">Blocco di testo</font></div>
</body>
```

Titolo blu

Blocco di testo corsivo

Blocco di testo grassetto

Blocco di testo

Liste non ordinate

Tramite **** definiamo liste non ordinate. Gli elementi della lista sono definiti mediante ****. L'attributo **type** = {disc | circle | square} definisce la forma dei punti.

Liste ordinate

Tramite **** definiamo liste ordinate. Gli elementi della lista sono definiti mediante ****. L'attributo **type** = {1 | a | A | i | I} definisce la numerazione.

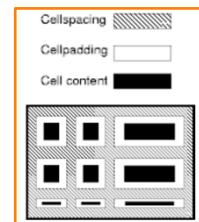
Liste di definizioni <dl>

Tramite **<dl>** definiamo liste di definizione. Gli elementi della lista sono definiti mediante **<dt>** per i titoli delle definizioni e **<dd>** per le definizioni.

Tabella <table>

Il tag **<table>** racchiude la tabella. Prevede gli attributi:

- **align** = “{left|center|right}” Allineamento della tabella rispetto alla pagina
- **width** = “n|n%” Larghezza della tabella (anche in percentuale rispetto alla pagina)
- **bgcolor** = “#xxxxxxxx” Colore di sfondo della tabella [[140 nomi di colori](#)]
- **border** = “n” Spessore dei bordi della tabella (0 = tabella senza bordi)
- **cellspacing, cellpadding**



<tr> è il tag che racchiude ciascuna riga della tabella

- **align** = “{left|center|right|justify}” Allineamento del contenuto delle celle della riga
- **valign** = “{top|middle|bottom|baseline}” Allineamento verticale del contenuto delle celle della riga
- **bgcolor** = “#xxxxxxxx” Colore di sfondo della riga

<th> per le celle della testata

<td> per le celle del contenuto

- **align, valign, bgcolor**

▪ **width, height** = {length|length%} Specifica le dimensioni (larghezza e altezza) della cella, dimensione assoluta (pixels) o valore percentuale

▪ **rowspan, colspan** = n indica su quante righe/colonne della tabella si estende la cella

Le tabelle permettono di realizzare i cosiddetti layout “liquidi”, che si adattano cioè alla risoluzione del monitor dell’utente (grazie all’uso delle dimensioni in %).

Esempio:

```
<body>
<table border="1">
<tr bgcolor="#E6E6FA">
<th rowspan="2"></th>
<th colspan="2">Media</th>
<th rowspan="2">Praticano<br/>sport</th>
</tr>

<tr bgcolor="#FFFACD">
<th>Altezza</th><th>Peso</th>
</tr>

<tr bgcolor="#AFEEEE">
<th>Maschi</th>
<td>1.80</td><td>77</td>
<td>70%</td>
</tr>

<tr bgcolor="#FFF0F5">
<th>Femmine</th>
<td>1.70</td><td>60</td>
<td>43%</td>
</tr>
</table>
</body>
```

Riga 1

Riga 2

	Media	Praticano sport
Maschi	Altezza	Peso
1.80	77	70%
Femmine	1.70	60
		43%

Link

Il link è una connessione tra risorse web; ha direzione di percorrenza monodirezionale tra:
source anchore → destination anchore

Link

Con il tag possiamo definire ancora d'origine, tramite attributo **href** che contiene un URL di destinazione, e ancora destinazione, tramite attributo **name**.

Nel caso in cui si voglia collegare due elementi all'interno di uno stesso documento...

...l'attributo *href* dell'àncora di origine ha la forma **#nome** dove **nome** è il valore dell'attributo *name* dell'àncora di destinazione.

Si può esprimere un'àncora di destinazione in forma "implicita", cioè senza utilizzare il tag.
È sufficiente assegnare l'attributo **id** a un qualunque elemento della pagina. Un elemento **#xxxx** posto alla fine di un URL viene chiamato **fragment**.

Il caso più completo è quello di un link ad un punto preciso di un documento:
URL HTML + FRAGMENT



Gli URL utilizzati nell'attributo HREF possono essere assoluti o relativi. Se sono relativi si procede alla «risoluzione» utilizzando come base quella del documento corrente. Se desidero un comportamento diverso, attivo l'attributo `<base>` dell'header.

Immagini ``

Il tag `` consente di inserire immagini in un documento HTML con la sintassi:

- **src** = uri specifica l'indirizzo dell'immagine (**required**)
- **alt** = text testo alternativo nel caso fosse impossibile visualizzare l'immagine
- **align** = {bottom | middle | top | left | right} (deprecato in HTML 4.01) posizione dell'immagine rispetto al testo che la circonda
- **width, height** = pixels larghezza e altezza dell'immagine in pixel
- **border** = pixels (deprecato in HTML 4.01) spessore del bordo dell'immagine

Modulo form <form>

Il tag <form> racchiude tutti gli elementi del modulo. Sono disponibili i seguenti attributi:

- **action** = uri URI dell'agente che riceverà i dati del form
- **name** = text specifica il nome del form
- **method** = {get|post} specifica il modo in cui i dati vengono inviati
- **enctype** = content-type se il metodo è POST specifica il content type usato per la codifica (encoding) dei dati contenuti nel form

La maggior parte dei controlli viene definita mediante il tag <input>,

l'attributo **type** stabilisce il tipo di controllo:

- **text**: casella di testo monoriga
 - **name** = text nome del controllo
 - **value** = text eventuale valore iniziale
 - **size** = n lunghezza del campo (numero di caratteri)
 - **maxlength** = n massima lunghezza del testo (numero di caratteri)
- **password**: come text ma il testo non è leggibile (****)

Nome:

- **file**: controllo che consente di caricare un file
 - **name** = text specifica il nome del controllo
 - **value** = content-type lista di MIME types per l'upload
 - Richiede una codifica particolare per il form (**multipart/form-data**) perché le informazioni trasmesse con il POST contengono tipologie di dati diverse

```
<form action="http://site.com/bin/adduser" method="post">
  <input type="file" name="attach">
</form>
```

Pane
 Burro
 Acqua

- **checkbox**: casella di spunta
 - **name** = text nome del controllo
 - **value** = text valore restituito se la casella viene spuntata
 - **checked** = "checked"

- **radio**: radio button
 - è una casella di spunta (stessi attributi di checkbox) che serve per realizzare gruppi di scelta mutuamente esclusivi.

- **submit**: bottone per trasmettere il contenuto del form
 - L'etichetta che compare nel bottone viene definita usando l'attributo value

- **image**: bottone di submit sotto forma di immagine

- **reset**: bottone che riporta tutti i campi al valore iniziale
 - L'etichetta che compare nel bottone viene definita usando l'attributo value

- **button**: bottone di azione
 - L'etichetta che compare nel bottone viene definita usando l'attributo value

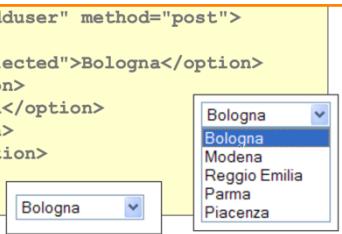
- **hidden**: campo nascosto

Tutti gli input possono essere disabilitati utilizzando l'attributo **disabled** nella forma **disabled = "disabled"**

Form

Il tag **<select>** permette di costruire liste di opzioni.

Per definire le singole opzioni si usa il tag **<option>**; ricorrendo all'attributo **value** si può attribuire il valore.



```
<form action="http://site.com/bin/adduser" method="post">
<select name="provincia">
<option value="BO" selected="selected">Bologna</option>
<option value="MO">Modena</option>
<option value="RE">Reggio Emilia</option>
<option value="PR">Parma</option>
<option value="PC">Piacenza</option>
</select>
</form>
```

(Con l'attributo **selected** si può indicare una scelta predefinita: *selected="selected"*)

Se si utilizza l'attributo **multiple** (nella forma *multiple="multiple"*) non abbiamo più un combo ma una lista sempre aperta. Si può operare una scelta multipla tenendo premuto il tasto **[Ctrl]** durante la selezione.

Con il tag **<optgroup>** è possibile organizzare la lista (sia a scelta singola che multipla) in gruppi.

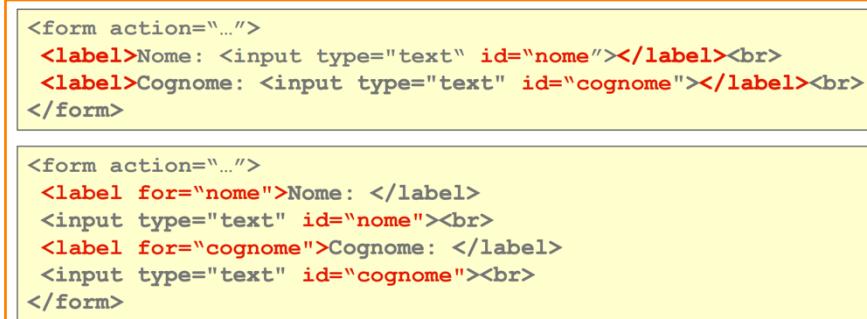


```
<form action="http://site.com/bin/adduser" method="post">
<select name="provincia" multiple="multiple" size=7>
<optgroup label="Capoluogo">
<option value="BO" selected="selected">Bologna</option>
</optgroup>
<optgroup label="Emilia">
<option value="MO">Modena</option>
<option value="RE">Reggio Emilia</option>
<option value="PR">Parma</option>
<option value="PC">Piacenza</option>
</optgroup>
</select>
</form>
```

Con il tag **<fieldset>** si possono creare gruppi di campi a cui è possibile attribuire un nome utilizzando il tag **<legend>**.

Il tag **<label>** permette di associare un'etichetta ad un qualunque controllo di un form.

L'associazione può essere fatta in forma implicita inserendo il controllo nell'elemento **label**, oppure in forma esplicita tramite l'attributo **for** che deve corrispondere all'attributo **id** del controllo.

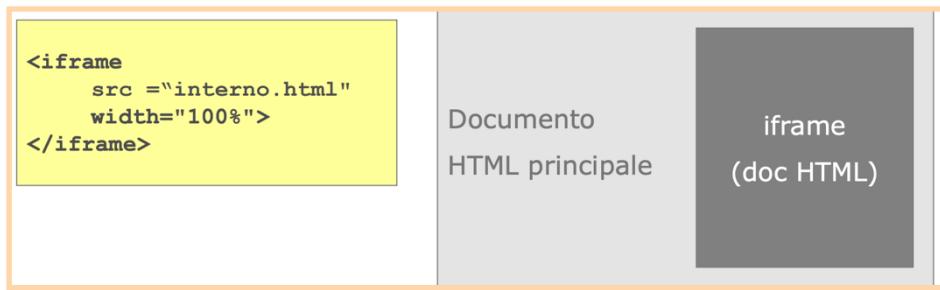


```
<form action="...">
<label>Nome: <input type="text" id="nome"></label><br>
<label>Cognome: <input type="text" id="cognome"></label><br>
</form>

<form action="...">
<label for="nome">Nome: </label>
<input type="text" id="nome"><br>
<label for="cognome">Cognome: </label>
<input type="text" id="cognome"><br>
</form>
```

Inline frame <iframe>

Il tag <iframe> crea un frame inline che contiene un altro documento specificato tramite l'attributo **src**. È deprecato in HTML 4.01 ma ancora molto utilizzato.



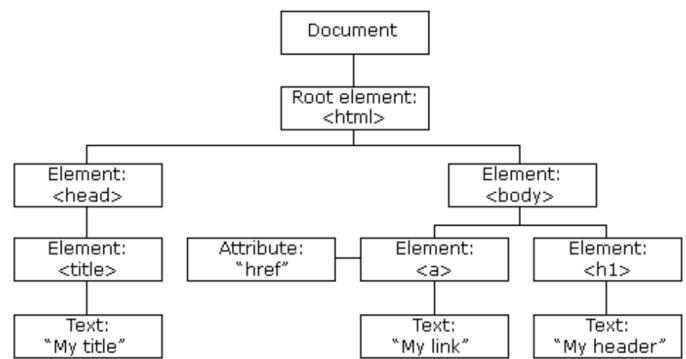
4.4 DOM

Una pagina HTML può essere rappresentata come una struttura ad albero. Questa struttura logica prende il nome di **DOM**: Document Object Model.

Il DOM è scaricato dal browser web sulla RAM.

```
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <a href="...">MyLink</a>
    <h1>My header</h1>
  </body>
</html>
```

Testo HTML



DOM

5 CSS

I fogli di stile a cascata (**CSS**) hanno lo scopo di separare contenuto e presentazione nelle pagine Web. Mentre l'HTML definisce il contenuto, i CSS esprimono come il contenuto deve essere presentato all'utente. È prevista ed è incoraggiata la presenza di fogli di stile multipli, che agiscono uno dopo l'altro, in cascata, per indicare le caratteristiche tipografiche e di layout.

Un documento HTML può essere visto come un insieme di blocchi (contenitori) sui quali si può agire con stili diversi. Ogni tag definisce un blocco.

```
<html>
<head>...</head>
<body>
<h1>title</h1>
<div>
<p> uno </p>
<p> due </p>
</div>
<p> tre
<a href="link.html">link</a>
</p>
</body>
</html>
```

Una volta definito il nostro file **.css**, tramite link nell'head dell'HTML, possiamo collegare i due documenti. Se si sceglie di mettere gli stili in file separati (si parla di *stili esterni*) sono possibili due sintassi diverse:

```
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
<link rel="stylesheet"
      href="hello.css"
      type="text/css">
</HEAD>
<BODY>
```

```
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
<style type="text/css">
  @import url(hello.css);
</style>
</HEAD>
<BODY>
```

Se invece si sceglie di mettere gli stili nella pagina si può procedere in due modi:

- Mettere tutti gli stili nell'header in un tag **<style>** (*stili interni*)
- Inserirli nei singoli elementi (*stili inline*)

```
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
<style type="text/css">
  BODY { color: red }
  H1 { color: blue }
</style>
</HEAD>
<BODY>
```

```
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
</HEAD>
<BODY style="color: red">
  <H1 style="color: blue">
    Hello World!
  </H1>
  <p>Usiamo i CSS</p>
</BODY>
```

Un'espressione come **H1 { color: blue }** prende il nome di **regola CSS**.

Una regola è composta da due parti: **Selettore** (H1) e **Dichiarazione** (color: blue). La dichiarazione a sua volta si divide in: **Proprietà** (color) e **Valore** (blue).

La sintassi è dunque la seguente...

```
selettore { proprietà: valore }
```

O più in generale:

```
selettore {
  proprietà1 : valore1;
  proprietà2 : valore2, valore3; }
```

5.1 Selettori e raggruppamenti

Il selettore serve per collegare uno stile agli elementi a cui deve essere applicato.

Selettore universale: identifica qualunque elemento

`*{...}`

Selettore di tipo: si applicano a tutti gli elementi di un determinato tipo (es. `<p>`)

`tipo_elemento{...}`

Classi: si applicano a tutti gli elementi che presentano l'attributo `class="nome_classe"`

`.nome_classe{...}`

Identificatori: si applicano a tutti gli elementi che presentano l'attributo `id="nome_id"`

`#nome_id{...}`

I selettori di tipo si possono combinare con quelli di classe e di identificatore:

`tipo_elemento.nome_classe{...}`

`tipo_elemento#nome_id{...}`

Pseudoclassi: si applicano ad un sottoinsieme degli elementi di un tipo identificato da una proprietà

`tipo_elemento:proprietà{...}`

▪ Es. stato di un'ancora: link e visited

`a:link{...}, a:visited{...}`

▪ Es: condizione di un elemento: active, focus e hover

`h1:hover{...}, ...`

Pseudoelementi: si applicano ad una parte di un elemento

`tipo_elemento:parte{...}`

▪ Es. solo la prima linea o la prima lettera di un paragrafo:

`p:first-line{...}, p:first-letter{...}`

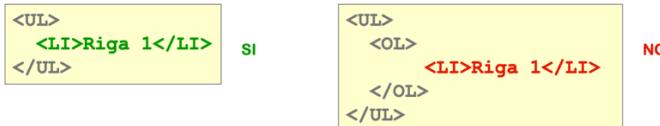
Selettori gerarchici: si applicano a tutti gli elementi di un dato tipo che hanno un determinato legame gerarchico (discendente, figlio, fratello) con elementi di un altro tipo

▪ `tipo1 tipo2{...}` tipo2 discende da tipo1

▪ `tipo1>tipo2{...}` tipo2 è figlio di tipo1

▪ `tipo1+tipo2{...}` tipo2 è fratello di tipo1

▪ Ad esempio: `UL>LI{...}` si applica solo agli elementi contenuti direttamente in liste non ordinate:



Se la stessa dichiarazione si applica a più tipi di elemento si scrive una regola in forma raggruppata:

`H1, H2, H3 {font-family: sans-serif}`

5.2 Proprietà e valori

Nelle dichiarazioni è possibile far uso di proprietà singole o in forma abbreviata:

- Le **proprietà singole** permettono di definire un singolo aspetto di stile
- Le **shorthand properties** consentono invece di definire un insieme di aspetti, correlati fra di loro usando una sola proprietà.

Per esempio, ogni elemento permette di definire un margine rispetto a quelli adiacenti usando quattro proprietà: *margin-top*, *margin-right*, *margin-bottom*, *margin-left*.

Sono equivalenti le scritture:

proprietà singole	shorthand properties
<pre>P { margin-top: 10px; margin-right: 8px; margin-bottom: 10px; margin-left: 8px; }</pre>	<pre>P {margin: 10px 8px 10px 8px;}</pre>

Notiamo che per i valori...

- **Numeri**: interi e reali: “.” come separatore decimale
- **Grandezze**: usate per lunghezze orizzontali e verticali
- **Unità di misura relative**:
 - **em**: è relativa alla dimensione del font in uso
(es. se il font ha corpo 12pt, 1em varrà 12pt, 2em varranno 24pt)
 - **px**: pixel, sono relativi al dispositivo di output e alle impostazioni del computer dell'utente
- **Unità di misura assolute**:
 - **in**: pollici; (1 in = 2.54 cm)
 - **cm**: centimetri
 - **mm**: millimetri
 - **pt**: punti tipografici (1/72 di pollice)
 - **pc**: pica = 12 punti
- **Percentuali**: percentuale del valore che assume la proprietà stessa nell'elemento padre; un numero seguito da %
- **URL**: assoluti o relativi; si usa la notazione: *url(percorso)*
- **Stringhe**: testo delimitato da apici singoli o doppi
- **Colori**: possono essere identificati con tre metodi differenti:
 - In forma esadecimale **#RRGGBB**
 - Tramite la funzione **rgb(rosso,verde,blu)**
 - Usando una serie di **parole chiave**

5.3 Ereditarietà e cascade

Per poter rappresentare una pagina HTML il browser deve riuscire ad applicare ad ogni elemento uno stile. Ogni browser ha un foglio stile di default che contiene stili per ogni tipologia di elemento HTML (tag). L'attribuzione può essere diretta se l'elemento contiene uno stile inline o esistono una o più regole il cui selettori rimanda all'elemento; oppure può essere indiretta se l'elemento "eredita" lo stile dall'elemento che lo contiene.

L'**ereditarietà** è un meccanismo di tipo differenziale simile per certi aspetti all'ereditarietà nei linguaggi ad oggetti. Si basa sui blocchi annidati di un documento HTML, uno stile applicato ad un blocco esterno si applica anche ai blocchi in esso contenuti.

In un blocco interno si possono definire stili aggiuntivi e si possono ridefinire stili già definiti.

Non tutte le proprietà sono soggette al meccanismo dell'ereditarietà!

In generale non vengono ereditate quelle che riguardano la formattazione del box model ovvero il riquadro che circonda ogni elemento.

Il CSS definisce un insieme di regole di risoluzione dei conflitti che prende il nome di **cascade**.

La logica di risoluzione si basa su tre elementi:

- **Origine del foglio stile:**

- Autore: stile definito nella pagina
- Browser: foglio stile predefinito
- Utente: in alcuni browser si può editare lo stile

- **Specificità:** esiste una formula che misura il grado di specificità attribuendo, ad es., un punteggio maggiore ad un selettori di ID rispetto ad uno di Classe

- **Dichiarazione !important:** è possibile aggiungere al valore di una dichiarazione la clausola !important (*p.myClass {text-color: red !important}*)

In caso di conflitto vince quella con peso maggiore

Per determinare il peso si applicano in sequenza una serie di regole:

→ **Origine:** l'ordine di prevalenza è autore, utente, browser

→ **Specificità del selettori:** ha la precedenza il selettori con specificità maggiore

→ **Ordine di dichiarazione:** se esistono due dichiarazioni con ugual specificità e origine
vince quella fornita per ultima

→ Una regola marcata come **!important** ha sempre precedenza sulle altre

5.4 Colori e Testi

CSS definisce una sessantina di proprietà che ricadono grosso modo nei seguenti gruppi:

- Colori e sfondi
- Caratteri e testo
- Box model
- Liste
- Display e gestione degli elementi floating
- Posizionamento
- Tabelle

Color -----

Per ogni elemento si possono definire almeno tre colori:

- il colore di primo piano (**foreground color**)
- il colore di sfondo (**background color**)
- il colore del bordo (**border color**)

La proprietà **color** definisce esclusivamente:

- il colore di primo piano, ovvero quello del testo
- il colore del bordo di un elemento, quando non viene impostato esplicitamente con border-color

La sintassi di color è:

selettore { color: <valore> }

La definizione dello sfondo (*background*) può essere applicata a due soli elementi: body e tabelle

Proprietà singole e valori:

- **background-color**: colore oppure transparent
- **background-image**: url di un'immagine o none
- **background-repeat**: {repeat|repeat-x|repeat-y|no-repeat}
- **background-attachment**: {scroll|fixed}
- **background-position**: x,y in % o assoluti o parole chiave (top|left|bottom|right)

Proprietà in forma breve: **background**
selettore {background: background-color background-image background-repeat background-attachment background-position;}

Font -----

Esistono proprietà per definire tutti gli elementi classici della tipografia.

Un font è una serie completa di caratteri (lettere, cifre, segni di interpunkzione) con lo stesso stile.

Possono essere classificati sulla base di diversi criteri come, ad esempio, la presenza (*serif*) o non di grazie (*sans serif*).

La proprietà che ci permette di definire il tipo di carattere è **font-family**:

p {font-family: Verdana;}

I font pongono un problema di compatibilità tra piattaforme piuttosto complesso

Le 5 famiglie generiche sono e hanno una corrispondenza specifica che dipende dalla piattaforma (fra parentesi i valori utilizzati da Windows):

- **serif** (Times New Roman)
- **sans-serif** (Arial)
- **cursive** (Comic Sans)
- **fantasy** (Allegro BT)
- **monospace** (Courier)

È possibile anche una dichiarazione multipla tale che il browser provi a caricare i font per una pagina per ordine.

```
p {font-family: Verdana, Helvetica, sans-serif;}
```

La proprietà **font-size** permette di definire le dimensioni del testo.

La dimensione può essere espressa in forma assoluta:

- con una serie di parole chiave: *xx-small, x-small, small, medium, large, x-large, xx-large*
- con unità di misura assolute: tipicamente pixel (*px*) e punti (*pt*)

Oppure in forma relativa:

- con le parole chiave *smaller* e *larger*
- con l'unità *em, ex, %*

La proprietà **font-weight** definisce il peso del carattere (la grossezza dei tratti che lo compongono).

Il peso si può esprimere in diversi modi:

- Valori numerici: *100, 200, ... 800, 900*
- Parole chiave assolute: *normal, bold*
- Parole chiave relative: *bolder, lighter*

La proprietà **font-style** permette di definire varianti del testo rispetto al normale:

- *normal*: valore di default (tondo)
- *italic*: testo in corsivo
- *oblique*: testo obliquo, simile a italic

La proprietà **font** è una proprietà sintetica che consente di definire tutti gli attributi dei caratteri in un colpo solo, nell'ordine:

font-style font-variant font-weight font-size font-family font di sistema

I font di sistema permettono di adattare le pagine all'aspetto del sistema operativo, sono 6 valori:

- *caption*: font usato per bottoni e combo-box
- *icon*: font usato per il testo delle icone
- *menu*: carattere usato nei menu delle varie finestre
- *message-box*: carattere usato per i popup
- *small-caption*: carattere per i controlli più piccoli
- *status-bar*: font usato per la barra di stato

line-height permette di definire l'altezza di una riga di testo all'interno di un elemento blocco.

Con **text-align** possiamo definire l'allineamento: *left, right, center, justify*

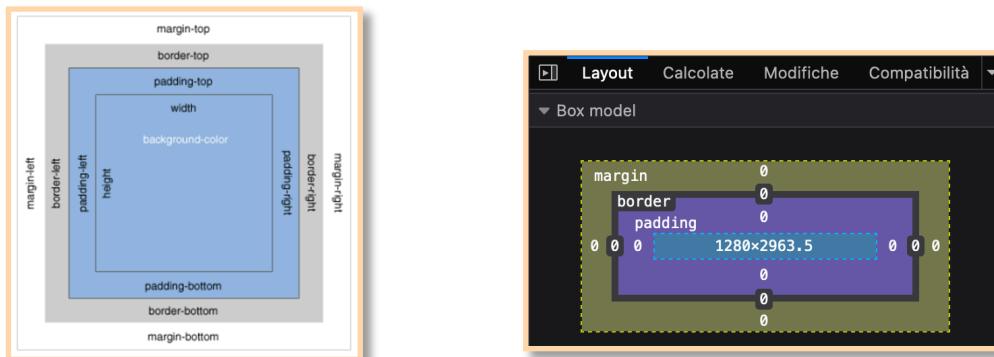
text-decoration permette invece di definire alcune decorazioni: *underline, overline, line-through*

text-indent definisce l'indentazione della prima riga in ogni elemento contenente del testo.

text-transform serve a cambiare gli attributi del testo relativamente a maiuscole e minuscole.

5.5 Box model

Per **box model** si intende l'insieme delle regole per la definizione degli stili per gli elementi blocco. I modello comprende cinque elementi base rappresentati nella figura:



Elementi:

- **Area del contenuto:** testo, immagine, ecc... di cui è possibile definire altezza e larghezza (**width** e **height**)
- **Padding:** spazio vuoto tra il contenuto e il bordo dell'elemento (**padding**)
- **Bordo (border):** di cui possiamo definire colore, stile e spessore
- **Margine (margin):** spazio tra l'elemento e gli altri elementi adiacenti

La larghezza complessiva è data da:

$$\text{margin sx} + \text{bordo sx} + \text{padding sx} + \text{width} + \text{padding dx} + \text{bordo dx} + \text{margin dx}$$

Se non si impone specificamente il valore **width** (o si specifica il valore **auto**) la dimensione del contenuto viene stabilita automaticamente dal browser.

Margini

Quattro proprietà singole: **margin-top**, **margin-right**, **margin-bottom**, **margin-left**.

Una proprietà sintetica: **margin**.

Padding

Quattro proprietà singole: **padding-top**, **padding-right**, **padding-bottom**, **padding-left**.

Una proprietà sintetica: **padding**.

Bordi

Dodici proprietà singole (3 per ogni bordo): **border-top-color**, **border-top-style**, **border-top-width**, **border-right-color**, ...

Se i quattro bordi hanno lo stesso stile: **border**.

Dimensioni del contenuto del Box Model:

- **height:** altezza, si applica a tutti gli elementi blocco escluse le colonne delle tabelle
- **min-height** e **max-height:** permettono di fissare l'altezza minima o quella massima anziché un valore esatto (**min-height** non funziona con IE)
- **width:** larghezza
- **min-width** e **max-width:** permettono di fissare la larghezza minima o quella max
→ **Valori ammessi:** *auto*, valore numerico con unità di misura, valore percentuale.

La proprietà **overflow** permette di definire il comportamento da adottare quando il contenuto (tipicamente testo) deborda dalle dimensioni fissate.

→ *visible, hidden, scroll, auto*.

5.6 Liste

Css definisce alcune proprietà che agiscono sulle liste puntate `` e numerate ``, o meglio sugli elementi delle liste `<i>`. Se applichiamo una proprietà alle liste la applichiamo a tutti gli elementi.

- **list-style-image**: definisce l'immagine da utilizzare come “punto elenco” e ammette i valori:
→ `url()`, `none`.
- **list-style-position**: indica la posizione del punto e ammette i valori
→ `inside`: il punto fa parte del testo, `outside`: il punto è esterno al testo.
- **list-style-type**: aspetto del punto-elenco.

5.7 Display

HTML classifica gli elementi in tre categorie: blocco, inline e lista.

Ogni elemento appartiene per default ad una di queste categorie ma la proprietà **display** permette di cambiare questa appartenenza.

I valori più comuni sono:

- **inline**: l'elemento diventa di tipo inline
- **block**: l'elemento diventa di tipo blocco
- **list-item**: l'elemento diventa di tipo lista
- **none**

Con **float** è possibile estrarre un elemento dal normale flusso del documento e spostarlo su uno dei lati del suo elemento contenitore.

```
<html>
<head>
<style type="text/css">
    img.sx { border-width: none; float: left}
    img.dx { border-width: none; float: right}
</style>
</head>
<body>
<p>Domenica 27 dicembre 2009 alle ore 15.30, il Museo del Patrimonio Industriale propone su prenotazione (massimo 25 partecipanti) il laboratorio "Making toons" dedicato ai bambini dai 7...</p>
<p>Domenica 27 dicembre 2009 alle ore 15.30, il Museo del Patrimonio Industriale propone su prenotazione (massimo 25 partecipanti) il laboratorio "Making toons" dedicato ai bambini dai 7 ai 10 anni. Dai personaggi di Walt Disney ai Gormiti, dalle Winx a Ben 10, sono tanti i cartoni animati che affascinano i ragazzi di tutte le età, ma ancora più affascinante è poter diventare per un giorno un disegnatore, creando i propri personaggi e scoprendo i segreti dell'animazione. Tutto questo sarà possibile al Museo del Patrimonio industriale con "Making Toons", un laboratorio dedicato ai ragazzi per imparare a fare semplici cartoni animati, approfondendone la scienza e la storia.


Domenica 27 dicembre 2009 alle ore 15.30, il Museo del Patrimonio Industriale propone su prenotazione (massimo 25 partecipanti) il laboratorio "Making toons" dedicato ai bambini dai 7 ai 10 anni. Dai personaggi di Walt Disney ai Gormiti, dalle Winx a Ben 10, sono tanti i


```

La proprietà **clear** serve a disattivare l'effetto della proprietà float sugli altri elementi che lo seguono, ovvero a impedire che al fianco di un elemento float (sx, dx o entrambi) compaiano altri elementi.

5.8 Posizionamento

position è la proprietà fondamentale per la gestione della posizione degli elementi, di cui determina la modalità di presentazione sulla pagina.

Non è ereditata e ammette i seguenti valori:

- **static**: (default) posizionamento naturale nel flusso
- **absolute**: il box dell'elemento viene rimosso dal flusso ed è posizionato rispetto al box contenitore del primo elemento antenato «posizionato», ovvero non static
- **relative**: l'elemento viene posizionato relativamente al box che l'elemento avrebbe occupato nel normale flusso del documento
- **fixed**: posizionamento rispetto al viewport (browser window)

left, top, right, bottom sono le coordinate del posizionamento.

visibility determina la visibilità e ammette due valori: **visible, hidden**.

z-index: permette di stabilire quale “layer” sta sopra e quale sta sotto.

5.9 Tabelle

- **table-layout**: non è ereditata e ammette due valori:

auto: layout trattato automaticamente dal browser, **fixed**: layout controllato dal CSS

- **border-collapse**: definisce il trattamento dei bordi interni e degli spazi fra celle
e ammette due valori: **collapse**: se viene impostato un bordo, le celle della tabella lo condividono,
separate: se viene impostato un bordo, ogni cella ha il suo, separato dalle altre

(Se si usa separate lo spazio tra le celle e tra i bordi si imposta con la proprietà **border-spacing**)

6 Web Dinamico

Il modello che abbiamo analizzato finora, basato sul concetto di ipertesto distribuito, ha una natura essenzialmente statica. Anche se l'utente può percorrere dinamicamente l'ipertesto in modi molto diversi, l'insieme dei contenuti è prefissato staticamente.

Immaginiamo di costruire un'enciclopedia dei Dinosauri consultabile via Web.

Il modello va in crisi se proviamo ad aggiungere una funzionalità molto semplice: ricerca per nome. È quindi necessaria un'estensione specifica... un programma scritto appositamente per l'enciclopedia che interpreti i parametri passati nel GET del form, cerchi nel file system la pagina specificata e la restituisca al Web server per l'invio al client.

La prima soluzione proposta per risolvere questo problema prende il nome di Common Gateway Interface **CGI**, già presente fino da HTTPv1.0. Le applicazioni che usano questo standard prendono il nome di **programmi CGI** (scritti in un qualunque linguaggio), essi vengono eseguiti dinamicamente in risposta alla chiamata.



I programmi CGI e il server comunicano in quattro modi:

- **Variabili di ambiente** del sistema operativo
- **Parametri** sulla linea di comandi: programma CGI viene lanciato in un processo pesante (si pensi a shell di sistema operativo che interpreta i parametri passati, ad esempio in metodo GET)
- **Standard Input** (usato con il metodo POST)
- **Standard Output** per restituire al server la pagina HTML da inviare al client

→ Metodo GET ←

Con il metodo GET il server passa il contenuto della form al programma CGI come se fosse da linea di comando di una shell. Linea di comando ha una lunghezza finita dipendente da SO (massimo 256 caratteri su SO UNIX).

Esempio:

www.dino.it/cerca?nomeTxt=diplodocus&cercaBtn=Cerca

Dove:

- www.dino.it è l'indirizzo del server web
- cerca è il nome del programma CGI
- nomeTxt=diplodocus&cercaBtn=Cerca è la riga di comandi

→ Metodo POST ←

usando POST non viene aggiunto nulla alla URL specificata da *action*. I dati del form, contenuti nell'header HTTP, vengono inviati al programma CGI tramite standard input; in questo modo si possono inviare dati lunghi.

Esempio:

www.dino.it/cerca

Dove cerca corrisponde un comando di linea senza parametri.

In C per accedere ai dati si apre aprire un file su stdin e si leggono i campi del post con fgetc():
nome = fgetc(stdin); // nomeTxt=diplodocus
btn = fgetc(stdin); // cercaBtn=Cerca

Prima di chiamare il programma CGI, il Web server imposta alcune variabili di sistema corrispondenti ai principali header http...

In C, si può usare getenv():

```
utente = getenv(REMOTE_HOST);
```

- **REQUEST_METHOD:** metodo usato dalla form
- **QUERY_STRING:** parte di URL che segue il "?"
- **REMOTE_HOST:** host che ha inviato la richiesta
- **CONTENT_TYPE:** tipo MIME dell'informazione contenuta nel body della richiesta (nel POST)
- **CONTENT_LENGTH:** lunghezza dei dati inviati
- **HTTP_USER_AGENT:** nome e versione del browser usato dal client

Il programma elabora i dati in ingresso ed emette un output per il client in attesa di risposta.

Per passare i dati al server il programma CGI usa stdout.

Il server preleva i dati dallo standard output e li invia al client incapsulandoli in messaggio http.

NOTA:

Il Web server deve riconoscere che una volta all'interno del nostro url non è specificata una pagina html ma un programma CGI. Per far ciò è necessario che i programmi siano tutti in un'apposita directory e che nella configurazione del server sia specificato il path ove trovare i programmi.

...MA la nostra enciclopedia presenta **altri problemi...**

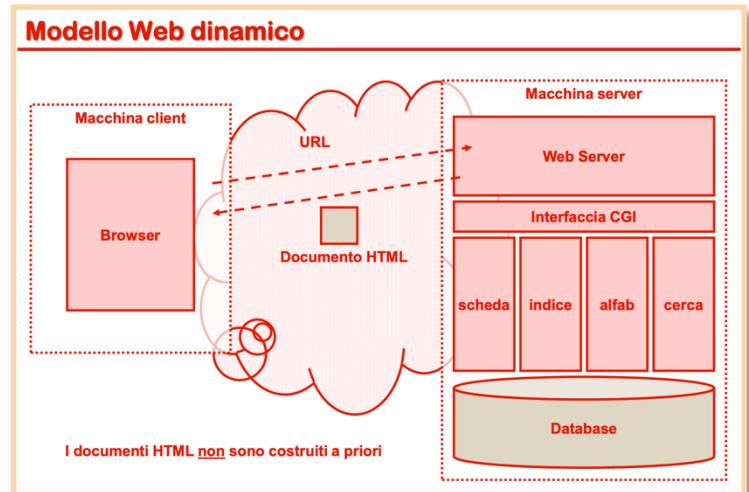
Ad esempio, la manutenibilità se volessimo aggiungere un altro dinosauro. Se poi volessimo cambiare l'aspetto grafico della nostra enciclopedia? dovremmo rifare una per una tutte le pagine (a meno di CSS). Una soluzione ragionevole è quella di separare gli aspetti di contenuto da quelli di presentazione sfruttando la possibilità di usufruire di un database.

In questo modo la gestione dell'enciclopedia è sicuramente più semplice...

Basta inserire un record nel database per aggiungere una nuova specie; l'indice tassonomico e quello alfabetico si aggiornano automaticamente.

Il modello è cambiato in modo significativo:

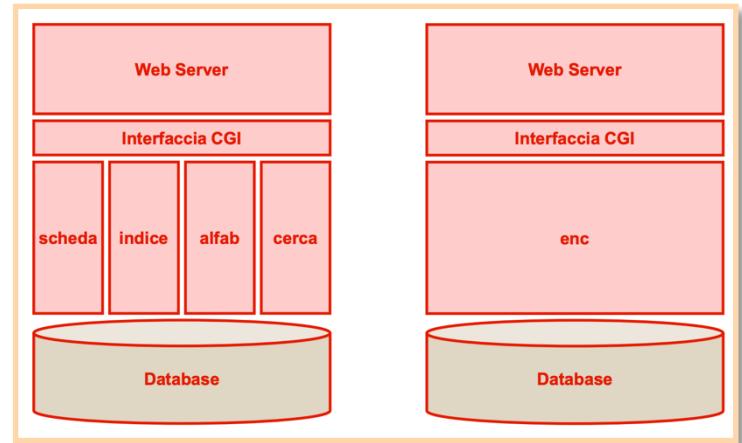
- Abbiamo aggiunto una forma di elaborazione lato server
- L'insieme delle CGI che gestiscono l'enciclopedia costituisce un'applicazione distribuita
- Ogni CGI può essere vista come una procedura remota invocata tramite HTTP dal cliente



L'architettura che abbiamo appena visto soffre comunque di diversi problemi:

- Ci sono problemi di prestazioni: ogni volta che viene invocata una CGI si crea un processo che viene distrutto alla fine dell'elaborazione
- Le CGI, soprattutto se scritte in C, possono essere poco robuste (che cosa succede se errore bloccante?)
- Ogni programma CGI deve reimplementare tutta una serie di parti comuni
- Abbiamo scarse garanzie sulla sicurezza

Per ovviare al penultimo punto si potrebbe realizzare una sola CGI (*enc*) che implementa tutte e quattro le funzionalità (vedi schema sottostante). In questo modo però si ha un'applicazione monolitica e si perdono i vantaggi della modularità.



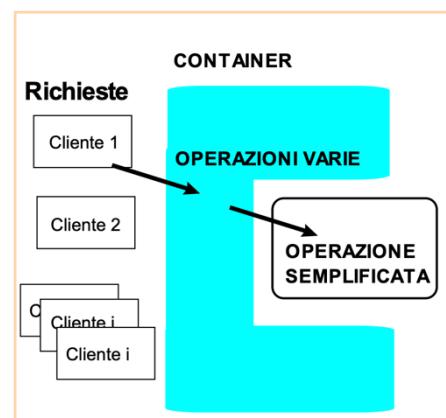
6.1 Modello a contenimento

La soluzione migliore è quella di realizzare un contenitore in cui far “vivere” le funzioni server-side. Il contenitore si preoccupa di fornire i servizi di cui le applicazioni hanno bisogno:

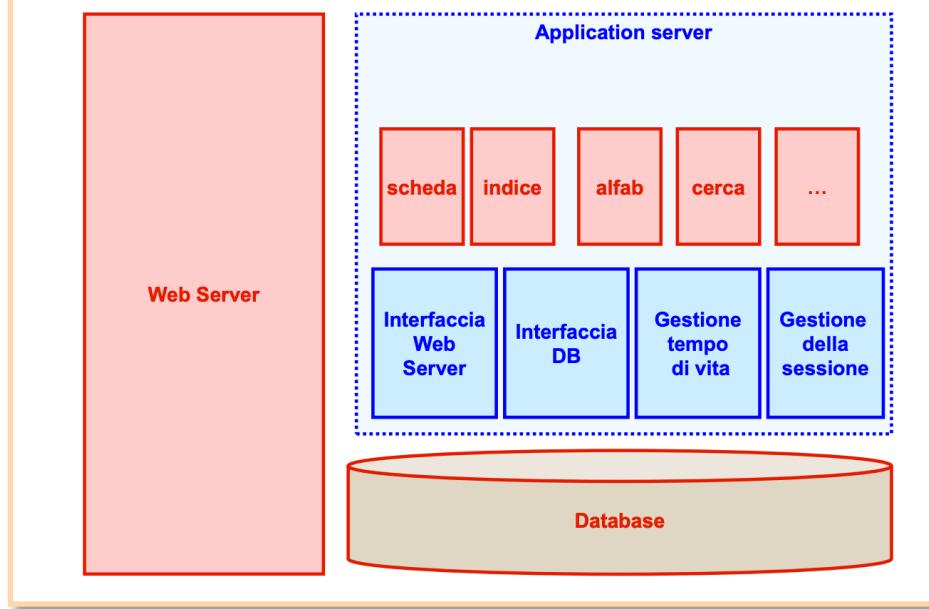
- Interfacciamento con il Web Server
- Gestione del tempo di vita (attivazione on-demand delle funzioni)
- Interfacciamento con il database
- Gestione della sicurezza

Si ha così una soluzione modulare in cui le funzionalità ripetitive vengono portate a fattor comune. Un ambiente di questo tipo prende il nome di **application server**.

Molte funzionalità possono essere non controllate ma lasciate come responsabilità ad un'entità delegata che prende il nome di **Container** (Engine, Middleware, ...); esso si occupa di azioni automatiche introducendo politiche di default. Queste funzioni comprendono supporto al ciclo di vita (attivazione/deattivazione del servitore, mantenimento dello stato), persistenza, recupero delle informazioni da DB, supporto al sistema di nomi e sicurezza.



Architettura basata su application server



L'interazione tra Client e Server può essere di due tipi:

- **Stateful:**

Esiste stato dell'interazione e quindi l'n-esimo messaggio può essere messo in relazione con gli n-1 precedenti. Tipico delle pagine in cui le richieste Web hanno bisogno di personalizzazione.

- **Stateless:**

Non si tiene traccia dello stato, ogni messaggio è indipendente dagli altri.

Un'interazione è stateless se progettata con operazioni idempotenti, ovvero stessa richiesta genera sempre stessa risposta.

6.2 Stato e sessione

Parlando di applicazioni Web è possibile classificare lo stato in modo più preciso...

- **Stato di esecuzione:**

Insieme dei dati parziali per una elaborazione.

Rappresenta un avanzamento in una esecuzione; per sua natura è uno stato volatile; può essere mantenuto in memoria lato server come stato di uno o più oggetti.

- **Stato di sessione:**

Insieme dei dati che caratterizzano una interazione con uno specifico utente.

Rappresenta la storia delle interazioni di UN cliente con UN servitore.

Una sessione viene gestita di solito in modo unificato attraverso l'uso di istanze di oggetti specifici.

- **Stato informativo persistente:**

Insieme dei dati che non devono essere persi anche dopo una sessione (tipicamente mantenuti in un DB).

La **sessione** rappresenta lo stato associato ad una sequenza di pagine visualizzate da un utente.

Contiene informazioni quali IP di provenienza, nome, cognome, username... ecc.

Lo **scope di sessione** è dato da un **tempo di vita** (lifespan) dell'interazione con l'utente, al termine del quale le variabili di sessione vengono eliminate, e dall'**accessibilità** (riconoscimento del client).

Definiamo infine **conversazione** il flusso di pagine web per ottenere la pagina voluta e terminare l'operazione desiderata (ad esempio l'insieme di pagine visitate per comprare su Amazon).

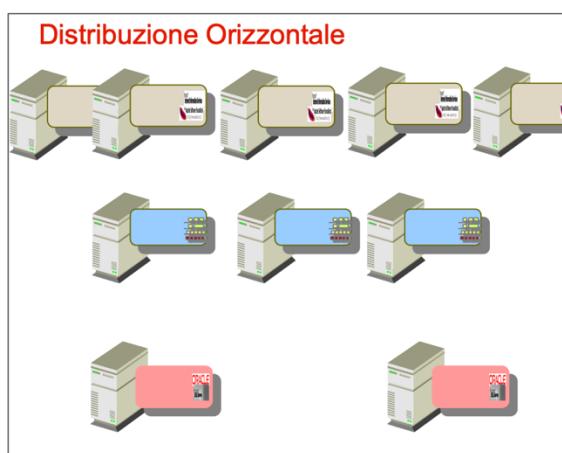
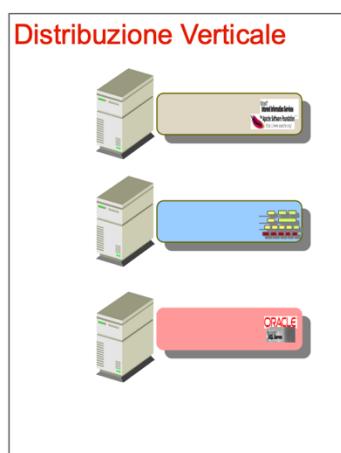
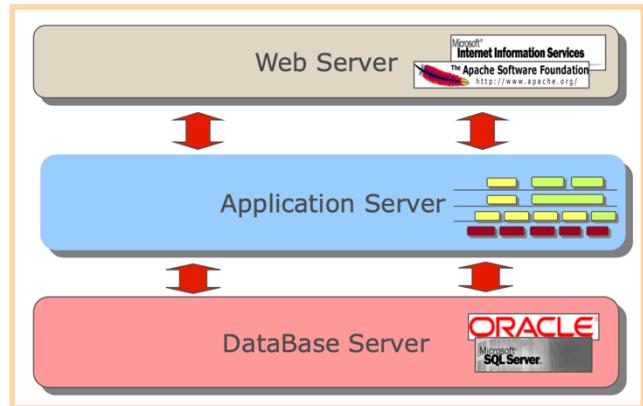
Le tecniche di base per la gestione dello stato sono:

- Utilizzo di **cookie** (con storage lato client)
- Gestione **stato sul server**

6.3 Architettura nei sistemi Web

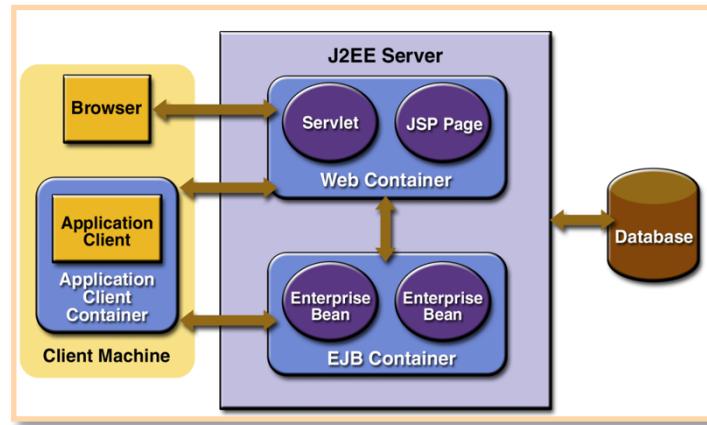
La struttura generale per la distribuzione di sistemi Web è dunque a **3 tier**: Web Server, Application Server, DataBase Server. Questi tre servizi possono risiedere su stesso HW o su macchine diverse. Orizzontalmente ad ogni livello è possibile replicare il servizio su diverse macchine; in questo caso si parla di **distribuzione orizzontale**, altrimenti **verticale**.

Notiamo che il Web server è stateless (per la natura del protocollo HTTP), dunque facile da replicare e a cui si possono applicare politiche di load balancing. Il Database è invece stateful la replicazione è molto delicata in quanto è necessario mantenere i principi sincronizzazione e di atomicità delle operazioni.



7 Servlet

I **Web Client** hanno sostituito, in molte situazioni di applicazioni client-server, i più tradizionali “fat-client”. Sono spesso costituiti dal semplice browser Web senza bisogno di alcuna installazione ad hoc e comunicano via HTTP e HTTPS con il server (come sapete, browser è, tra le altre cose, un client HTTP). Effettuano inoltre autonomamente il rendering della pagina in HTML.

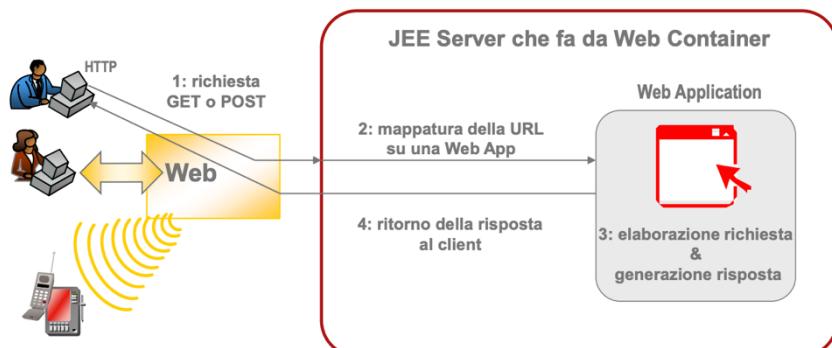


Una **Web Application** è un gruppo di risorse server-side che nel loro insieme creano una applicazione interattiva fruibile via Web. Le risorse server-side includono:

- **Classi server-side** (Servlet e classi standard Java)
- **Java Server Pages** (le vedremo in seguito)
- **Risorse statiche** (documenti HTML, immagini, css, ...)
- **Applet, Javascript** e/o altri componenti che diventeranno attivi client-side
- **Informazioni di configurazione**

I **Web Container** forniscono un ambiente di esecuzione per Web Application.

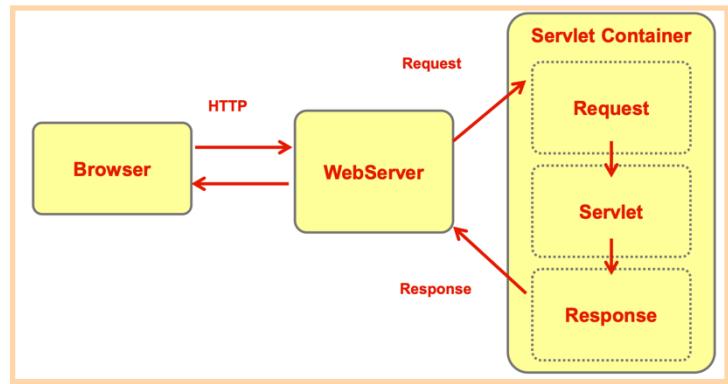
L'accesso ad una Web Application è un processo multi-step...



Una **Servlet** è una classe Java che fornisce risposte a richieste HTTP. Comunica dunque con il client tramite protocolli request/response. Le servlet estendono le funzionalità di un Web server generando contenuti dinamici e superando i classici limiti delle applicazioni CGI.

Eseguono direttamente in un Web Container, in termini pratici, sono classi che derivano dalla classe **HttpServlet** la quale implementa vari metodi che possiamo ridefinire. Le classi interessate sono contenute nel package `javax.servlet.http.*`.

All'arrivo di una richiesta HTTP il Web Server riconosce se la richiesta interessa una pagina statica o una servlet; nel secondo caso il Servlet Container crea un oggetto request e un oggetto response e li passa alla servlet. [Se necessario, alla prima richiesta, il Servlet Container carica la classe, crea una istanza e inizializza la servlet] leggi alla prossima pagina!



Gli oggetti di tipo **Request** rappresentano la chiamata al server effettuata dal client.

Sono caratterizzati da varie informazioni...

→ Chi ha effettuato la Request

→ Quali parametri sono stati passati nella Request

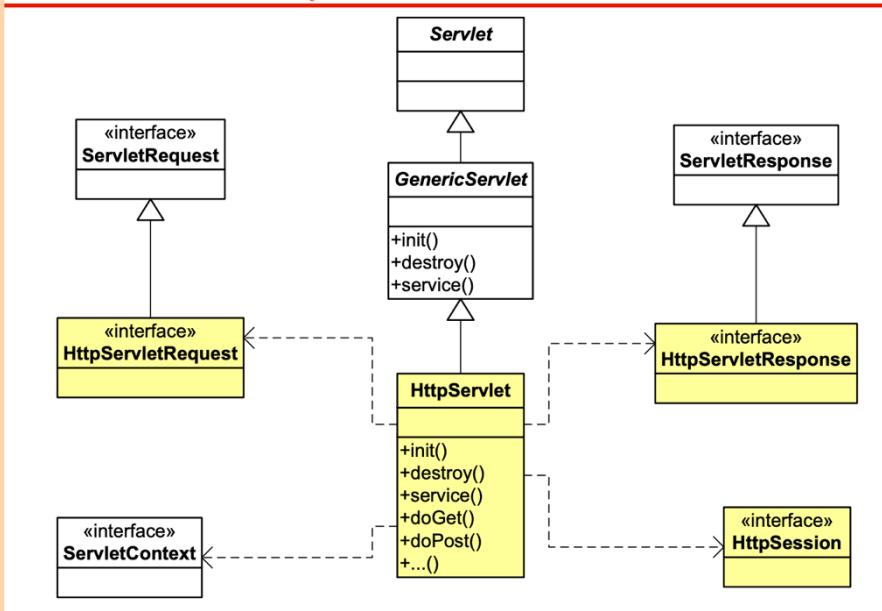
→ Quali header sono stati passati

Gli oggetti di tipo **Response** rappresentano le informazioni restituite al client ad una Request

→ Dati in forma testuale (es. html, text) o binaria (es. immagini)

→ HTTP header, cookie, ...

Classi e interfacce per Servlet



7.1 Ciclo di vita

Servlet container controlla e supporta automaticamente il ciclo di vita di una servlet.

Se non esiste una istanza della servlet nel container...

→ Carica la classe della servlet

→ Crea una istanza della servlet

→ Inizializza la servlet (invoca il metodo `init()`)

Poi, a regime...

→ Invoca la servlet (`doGet()` o `doPost()` a seconda del tipo di richiesta ricevuta) passando come parametri due oggetti di tipo `HttpServletRequest` e `HttpServletResponse`

Esistono due modelli di threading:

Modello normale o multithreading -----

UNA sola istanza di servlet indipendentemente dal numero di richieste ricevute...

Dunque, per **N richieste → 1 oggetto → N Thread**

Nella modalità normale più thread condividono la stessa istanza di una servlet e quindi si crea una situazione di concorrenza. Il metodo `service()` (`doGet()` e `doPost()`) può essere invocato da numerosi client in modo concorrente ed è quindi **necessario gestire le sezioni critiche** tramite **blocchi synchronized** (consigliati), semafori, mutex,...

Modello single-threaded (deprecated) -----

In questo caso il Server Container genera per ogni istanza un thread...

N richieste → N oggetti → N thread

Nonostante l'assenza di concorrenza il consumo di memoria risulta troppo alto.

7.2 Metodi

I metodi per il controllo del **ciclo di vita** sono nella classe astratta `GenericServlet`.

- **init()**: viene chiamato una sola volta al caricamento della servlet.

In questo metodo si può inizializzare l'istanza: ad esempio si crea la connessione con un database

- **service()**: viene chiamato ad ogni HTTP Request

→ Chiama `doGet()` o `doPost()` a seconda del tipo di HTTP Request ricevuta

- **destroy()**: viene chiamato una sola volta quando la servlet deve essere disattivata

NOTA: `HTTPServlet` fornisce una implementazione di `service()` che delega l'elaborazione della richiesta ai metodi `doGet()`, `doPost()`, `doPut()`, `doDelete()`.

Usiamo l'esempio Hello World per affrontare i vari aspetti della realizzazione di una servlet...

→ metodo GET ←

```
import java.io.*  
import javax.servlet.*  
import javax.servlet.http.*;  
public class HelloServlet extends HttpServlet {  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException {  
    ...  
}  
...  
}
```

Dobbiamo tener conto che in `doGet()` possono essere sollevate eccezioni di due tipi. Decidiamo di non gestirle per semplicità e quindi ricorriamo alla clausola `throws`.

---- OGGETTO RESPONSE -----

Nel nostro primo esempio ci è di interesse solamente l'oggetto `response`, esso contiene i dati restituiti dalla Servlet al Client:

- Status line (status code, status phrase)
- Header della risposta HTTP
- Response body: il contenuto (ad es. pagina HTML)

Ha come tipo l'interfaccia **HttpServletResponse** che espone metodi per:

- Specificare lo status code della risposta HTTP
- Indicare il content type (tipicamente `text/html`)
- Ottenere un output stream in cui scrivere il contenuto da restituire
- Indicare se l'output è bufferizzato
- Gestire i cookie



Per definire lo **status code** `HttpServletResponse` fornisce il metodo
public void setStatus(int statusCode)

Esempi di status Code: 200 OK, 404 Page not found

Per inviare **errori** possiamo anche usare

```
public void sendError(int sc)  
public void sendError(int code, String message)
```

Per la gestione degli **header**

public void setHeader(String headerName, String headerValue)
imposta un header arbitrario

public void setDateHeader(String name, long millisecs)
imposta la data

public void setIntHeader(String name, int headerValue)
imposta un header con un valore intero (evita la conversione intero-stringa)

addHeader, addDateHeader, addIntHeader

aggiungono una nuova occorrenza di un dato header

setContentType

configura il content-type (si usa sempre)

setContentLength

utile per la gestione di connessioni persistenti

addCookie

consente di gestire i cookie nella risposta

sendRedirect

imposta location header e cambia lo status code in modo da forzare una ridirezione

Per definire il response **body** possiamo operare in due modi utilizzando due metodi

public PrintWriter out = response.getWriter()
mette a disposizione uno stream di caratteri (un'istanza di PrintWriter)

public ServletOutputStream out = response.getOutputStream()

mette a disposizione uno stream di byte (un'istanza di ServletOutputStream)

```
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>Hello</title></head>"); 
    out.println("<body>Hello World!</body>"); 
    out.println("</html>");
```

Risposta generata

```
HTTP/1.1 200 OK
Content-Type: text/html
<html>
<head><title>Hello</title></head>
<body>Hello World!</body>
</html>
```

Supponiamo di voler aggiungere un elemento variabile...

Usando GET ricordiamo che nell'URL viene usata una query string con la seguente sintassi:

`<path>?<nome1>=<valore1>&<nome2>=<valore2>&...`

---- OGGETTO REQUEST -----

La richiesta contiene i dati inviati dal client HTTP al server. Viene creata dal Servlet Container e passata alla servlet.

È un'istanza di una classe che implementa l'interfaccia **HttpServletRequest** che fornisce metodi per accedere a varie informazioni:

- HTTP Request URL
- HTTP Request header
- Tipo di autenticazione e informazioni su utente
- Cookie
- Session



`http://[host]:[port]/[request path]?[query string]`

String getParameter(String parName)

restituisce il valore di un parametro individuato per nome

String getContextPath()

restituisce informazioni sulla parte dell'URL che indica il contesto della Web application

String getQueryString()

restituisce la stringa di query

String getPathInfo()

per ottenere il path

String getPathTranslated()

per ottenere informazioni sul path nella forma risolta

Per accedere agli **header**

String getHeader(String name)

restituisce il valore di un header individuato per nome sotto forma di stringa

Enumeration getHeaders(String name)

restituisce tutti i valori dell'header individuato da name sotto forma di enumerazione di stringhe (utile ad esempio per Accept che ammette n valori)

Enumeration getHeaderNames()

elenco dei nomi di tutti gli header presenti nella richiesta

int getIntHeader(name)

valore di un header convertito in intero

long getDateHeader(name)

valore della parte Date di header, convertito in long

Per autenticazione, sicurezza e cookie

String getRemoteUser()

nome di user se la servlet ha accesso autenticato, null altrimenti

String getAuthType()

nome dello schema di autenticazione usato per proteggere la servlet

boolean isUserInRole(java.lang.String role)

restituisce true se l'utente è associato al ruolo specificato

Cookie[] getCookies()

restituisce un array di oggetti cookie che il client ha inviato alla request

http://.../HelloServlet?to=Mario

```
public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
{
    String toName = request.getParameter("to");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>")
    out.println("<head><title>Hello to</title></head>");
    out.println("<body>Hello to " + toName + "!</body>");
    out.println("</html>");
}
```

HTTP/1.1 200 OK
Content-Type: text/html
<html>
<head><title>Hello</title></head>
<body>Hello to Mario!</body>
</html>

→ **metodo POST** ←

```
public void doPost(HttpServletRequest request,  
                    HttpServletResponse response)  
                    throws ServletException, IOException {  
    ...  
}
```

I form dichiarano i campi utilizzando l'attributo *name*. Quando il form viene inviato al server, nome dei campi e loro valori sono inclusi nella request.

```
<form action="myServlet" method="post">  
    First name: <input type="text" name="firstname"/><br/>  
    Last name: <input type="text" name="lastname"/>  
</form>
```

```
public class MyServlet extends HttpServlet  
{  
    public void doPost(HttpServletRequest rq, HttpServletResponse rs)  
    {  
        String firstname = rq.getParameter("firstname");  
        String lastname = rq.getParameter("lastname");  
    }  
}
```

HttpRequest espone anche il metodo **InputStream getInputStream()** che consente di leggere il body della richiesta.

Osservazione:

Se non viene ridefinito, il metodo **service()** effettua il dispatch delle richieste ai metodi **doGet**, **doPost**, ... a seconda del metodo HTTP usato nella request. Se si vuole trattare in modo uniforme get e post, si può ridefinire il metodo **service**.

7.3 Deployment

Ma... Come facciamo a far funzionare il nostro esempio?

Un'applicazione Web deve essere installata e questo processo prende il nome di **deployment**.

Il deployment comprende:

- La definizione del runtime environment di una Web Application
- La mappatura delle URL sulle servlet
- La definizione delle impostazioni di default di un'applicazione
- La configurazione delle caratteristiche di sicurezza dell'applicazione

Gli **Archivi Web (Web Archives)** sono file con estensione “.war”.

Rappresentano la modalità con cui avviene la distribuzione/deployment delle applicazioni Web.

Sono file jar con una struttura particolare...

```
jar {ctxu} [vf] [jarFile] files  
  
-ctxu: create, get the table of content, extract, update content  
-v: verbose  
-f: il JAR file sarà specificato con jarFile option  
-jarFile: nome del JAR file  
-files: lista separata da spazi dei file da includere nel JAR
```

Esempio

```
jar -cvf newArchive.war myWebApp/*
```

La struttura di directory delle Web Application è basata sulle Servlet 2.4 specification:

	MyWebApplication	Root della Web Application
	META-INF	Informazioni per i tool che generano archivi (manifest)
	WEB-INF	File privati (config) che non saranno serviti ai client
	classes	Classi server side: servlet e classi Java std
	lib	Archivi .jar usati dalla Web app
	web.xml	Web Application deployment descriptor

web.xml è in sostanza un file di configurazione (XML) che contiene una serie di elementi descrittivi. Contiene l'elenco delle servlet attive sul server, il loro mapping verso URL, e per ognuna di loro permette di definire una serie di parametri come coppie nome-valore.

Esempio di descrittore con mappatura:

```
<web-app>
  <servlet>
    <servlet-name>myServlet</servlet-name>
    <servlet-class>myPackage.MyServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>myServlet</servlet-name>
    <url-pattern>/myURL</url-pattern>
  </servlet-mapping>
</web-app>
```

Esempio di URL che viene mappato su myServlet:

```
http://MyHost:8080/MyWebApplication/myURL
```

Una servlet accede ai propri parametri di configurazione mediante l'interfaccia **ServletConfig**.

Ci sono 2 modi per accedere a oggetti di questo tipo:

- Il parametro di tipo **ServletConfig** passato al metodo **init()**
- il metodo **getServletConfig()** della servlet, che può essere invocato in qualunque momento

ServletConfig espone un metodo per ottenere il valore di un parametro in base al nome:

String getInitParameter(String parName).

```
<web-app>
  <servlet>
    <servlet-name>HelloServ</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
    <init-param>
      <param-name>title</param-name>
      <param-value>Hello page</param-value>
    </init-param>
    <init-param>
      <param-name>greeting</param-name>
      <param-value>Ciao</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloServ</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```

Ridefiniamo quindi anche il metodo `init()`:
memorizziamo i valori dei parametri in due attributi

```
import java.io.*  
import java.servlet.*  
import javax.servlet.http.*;  
  
public class HelloServlet extends HttpServlet  
{  
    private String title, greeting;  
  
    public void init(ServletConfig config)  
        throws ServletException  
    {  
        super.init(config);  
        title = config.getInitParameter("title");  
        greeting = config.getInitParameter("greeting");  
    }  
    ...
```

http://.../hello?to=Mario

Notare l'effetto della
mappatura tra l'URL hello e la servlet

```
public void doGet(HttpServletRequest request,  
    HttpServletResponse response)  
    throws ServletException, IOException  
{  
    String toName = request.getParameter("to");  
    response.setContentType("text/html");  
    PrintWriter out = response.getWriter();  
    out.println("<html>");  
    out.println("<head><title>" + title + "</title></head>");  
    out.println("<body>" + greeting + " " + toName + "!</body>");  
    out.println("</html>");  
}
```

HTTP/1.1 200 OK
Content-Type: text/html
<html>
<head><title>Hello page</title></head>
<body>Ciao Mario!</body>
</html>

Ogni Web application esegue in un contesto.

L'interfaccia **ServletContext** è la vista della Web application (del suo contesto) da parte della servlet. Si può ottenere un'istanza di tipo *ServletContext* all'interno della servlet utilizzando il metodo `getServletContext()` che consente di accedere ai parametri di inizializzazione, agli attributi del contesto e alle risorse statiche della Web application.

**Il servlet context viene condiviso tra tutti gli utenti, tutte le richieste
e tutti i componenti serverside!**

I parametri di inizializzazione del contesto definiti all'interno di elementi di tipo *context-param* in `web.xml` → sono accessibili in lettura a tutte le servlet della Web application.

Gli attributi di contesto sono accessibili a tutte le servlet e funzionano come variabili “globali”.

7.4 Gestione dello stato di sessione

Applicazioni Web hanno spesso bisogno di stato. Sono state definite due tecniche per mantenere traccia delle informazioni di stato...

- uso dei cookie: meccanismo di basso livello
- uso della sessione (session tracking): meccanismo di alto livello

Cookie -----

Il **cookie** è un'unità di informazione che Web server deposita sul Web browser lato cliente.

Sono parte dell'header HTTP, trasferiti in formato testuale; vengono infatti mandati avanti e indietro nelle richieste e nelle risposte.

Attenzione però! Possono essere rifiutati dal browser (tipicamente perché disabilitati) o essere considerati un fattore di rischio proprio perché visibili nell'header del protocollo.

Un cookie contiene un certo numero di informazioni, tra cui:

- una coppia nome/valore
- il dominio Internet dell'applicazione che ne fa uso
- path dell'applicazione
- una expiration date espressa in secondi
(-1 indica che il cookie non sarà memorizzato su file associato)
- un valore booleano per definirne il livello di sicurezza

La classe **Cookie** modella il cookie HTTP

- Si recuperano i cookie dalla request utilizzando il metodo **getCookies()**
- Si aggiungono cookie alla response utilizzando il metodo **addCookie()**

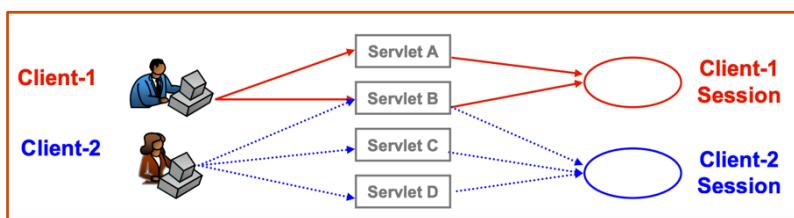
```
creazione  
Cookie c = new Cookie("MyCookie", "test");  
c.setSecure(true);  
c.setMaxAge(-1);  
c.setPath("/");  
response.addCookie(c);  
  
lettura  
Cookie[] cookies = request.getCookies();  
if(cookies != null)  
{  
    for(int j=0; j<cookies.length(); j++)  
    {  
        Cookie c = cookies[j];  
        out.println("Un cookie: " +  
            c.getName() + "=" + c.getValue());  
    }  
}
```

HttpSession

La sessione Web è un'entità gestita dal Web container.

È condivisa fra tutte le richieste provenienti dallo stesso client.

Può contenere dati di varia natura ed è identificata in modo univoco da un **session ID**.



Browser diversi → Sessioni diverse

Dispositivi diversi → Sessioni diverse

IP diversi MA stesso session ID → Stessa sessione

L'accesso avviene mediante l'interfaccia **HttpSession**.

Per ottenere un riferimento ad un oggetto di questo tipo si usa il metodo dell'interfaccia **HttpServletRequest**:

HttpSession getSession(boolean createNew);

Valori di createNew:

- **true**: ritorna la sessione esistente o, se non esiste, ne crea una nuova
- **false**: ritorna, se possibile, la sessione esistente, altrimenti ritorna null

```
HttpSession session = request.getSession(true);
```

Si possono memorizzare dati specifici dell'utente negli attributi della sessione (coppie nome/valore). Sono simili agli attributi di contesto, ma con scope fortemente diverso.

```
Cart sc = (Cart) session.getAttribute("shoppingCart");
sc.addItem(item);

session.setAttribute("shoppingCart", new Cart());
session.removeAttribute("shoppingCart");

Enumeration e = session.getAttributeNames();
while(e.hasMoreElements())
    out.println("Key: " + (String)e.nextElement());
```

Altre operazioni con le sessioni...

- **String getID()** restituisce l'ID di una sessione
- **boolean isNew()** dice se la sessione è nuova
- **void invalidate()** permette di invalidare (distruggere) una sessione
- **long getCreationTime()** dice da quanto tempo è attiva la sessione (in millisecondi)
- **long getLastAccessedTime()** dà informazioni su quando è stata utilizzata l'ultima volta

Come identificare una sessione?

Una tecnica per trasmettere l'ID è quella di includerlo in un cookie (session cookie): sappiamo però che non sempre i cookie sono attivati nel browser. Un'alternativa è rappresentata dall'inclusione del session ID nella URL, in questo caso si parla di **URL rewriting**.

È buona prassi codificare sempre le URL generate dalle servlet usando il metodo encodeURL() di *HttpServletResponse*. Dovrebbe essere usato per hyperlink e form.

7.5 Scoped objects

Gli oggetti di tipo *ServletContext*, *HttpSession*, *HttpServletRequest* forniscono metodi per immagazzinare e ritrovare oggetti nei loro rispettivi ambiti (scope).

Lo **scope** è definito dal tempo di vita (*lifespan*) e dall'accessibilità da parte delle servlet.

Ambito	Interfaccia	Tempo di vita	Accessibilità
Request	HttpServletRequest	Fino all'invio della risposta	Servlet corrente e ogni altra pagina interrogata tramite include o forward
Session	HttpSession	Durata della sessione utente	Ogni richiesta dello stesso cliente
Application	ServletContext	Lo stesso dell'applicazione	Ogni richiesta alla stessa Web app anche da clienti diversi e per servlet diverse

Gli oggetti scoped forniscono metodi per immagazzinare e ritrovare oggetti nei rispettivi ambiti:

- **void setAttribute(String name, Object o)**
- **Object getAttribute(String name)**
- **void removeAttribute(String name)**
- **Enumeration.getAttributeNames()**

7.6 Inclusione di risorse

Per includere una risorsa si ricorre a un oggetto di tipo **RequestDispatcher** che può essere richiesto al contesto indicando la risorsa da includere (statica o dinamica).

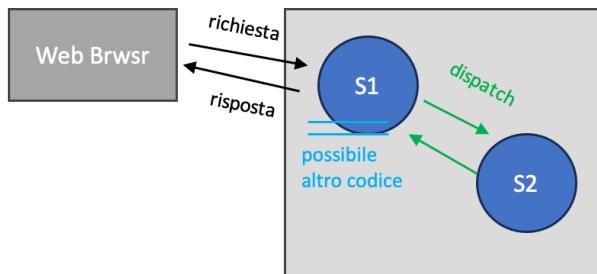
Esistono due tecniche di inclusione:

Include

Si invoca quindi il metodo `include` passando come parametri `request` e `response` che vengono così condivisi con la risorsa inclusa.

```
RequestDispatcher dispatcher =  
    getServletContext().getRequestDispatcher("/inServlet");  
dispatcher.include(request, response);
```

Gli stessi oggetti della servlet1 vengono passati alla servlet2 la quale ritorna il controllo alla prima per l'invio della risposta.

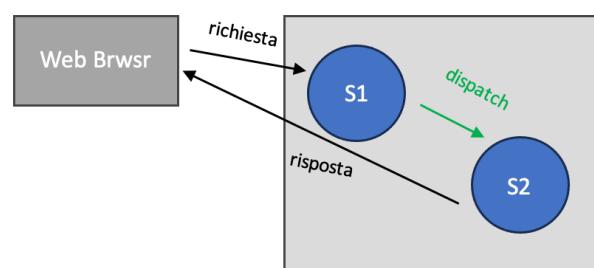


Forward

Si invoca quindi il metodo `forward` passando anche in questo caso `request` e `response`.

```
RequestDispatcher dispatcher =  
    getServletContext().getRequestDispatcher("/inServlet");  
dispatcher.forward(request, response);
```

L'oggetto di risposta è lavorato dalla servlet2.



APACHE TOMCAT

Un semplice Web server, sviluppato e distribuito dalla fondazione Apache, internamente scritto in Java. Permette di pubblicare siti Web e fornisce l'ambiente di esecuzione per applicazioni Web scritte secondo le specifiche JSP e Servlet.

Download da sito ufficiale → .zip

→ impostare variabile d'ambiente JRE_HOME o JAVA_HOME indicando un'installazione JDK

export JRE_HOME=... (unix)

set JRE_HOME=... (windows)

/Library/Java/JavaVirtualMachines/jdk-19.jdk

→ lanciare il server

TOMCAT_HOME/bin/startup.sh (unix)

TOMCAT_HOME/bin/startup.bat (windows)

→ accedere: <http://localhost:8080/>

→ deploy: Export > WAR file

→ <http://localhost:8080/nome-progetto/nome-servlet-jsp>

Eclps > Window > Show View > Servers → File > New > Server > Apache...

Struttura di TOMCAT su file system:

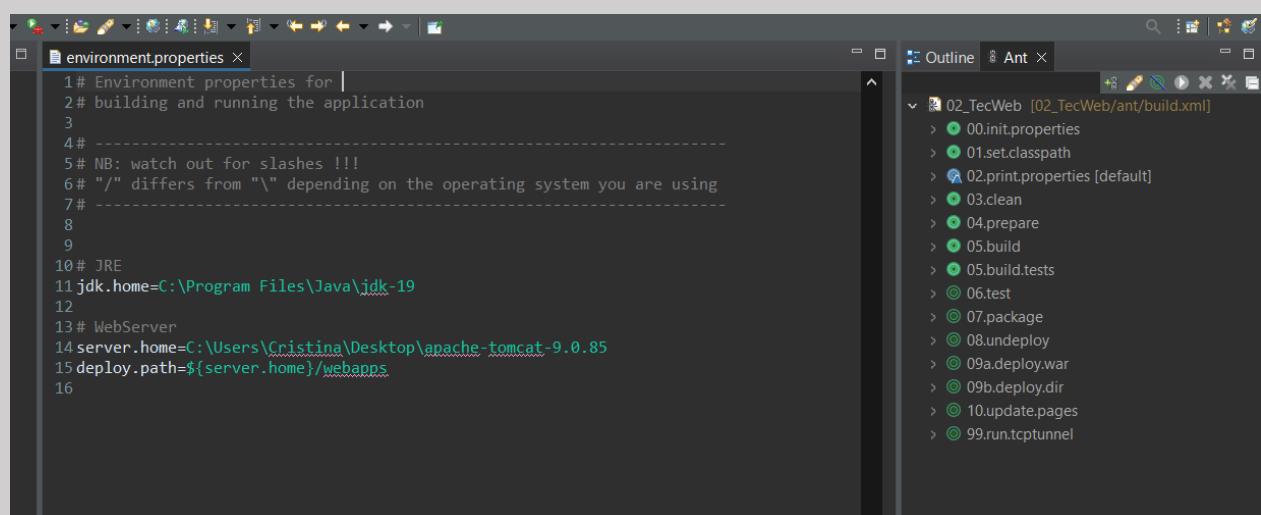
- bin: script e comandi di avvio
- common: librerie Java visibili e condivise da tutte le applicazioni Web in esecuzione sul server
- conf: configurazione di porte, permessi e altre risorse
- logs: file di log (da creare a mano se non esiste)
- server: codice del server
- webapps: pubblicazione delle applicazioni Web (file .war)
- temp, work: directory per le operazioni del server (salvataggio dei dati di sessione, compilazione delle pagine JSP, ...)

ANT

L'obbiettivo di ANT è facilitare il processo di deployment e compilazione di un'applicazione Java.

Eclipse → Window > Show View > Ant

Cartella di progetto "ant" → build.xml nella apposita view



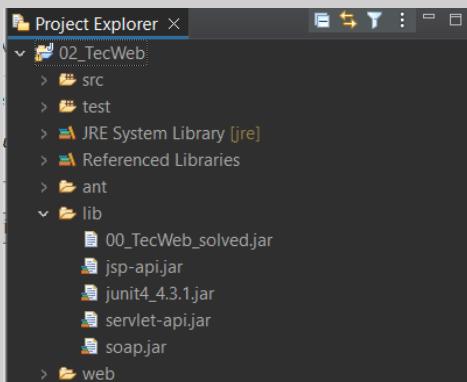
The screenshot shows the Eclipse IDE interface with the Ant view open. On the left, there is a code editor window titled "environment.properties" containing the following content:

```
1# Environment properties for |
2# building and running the application
3#
4# -----
5# NB: watch out for slashes !!!
6# "/" differs from "\" depending on the operating system you are using
7#
8#
9#
10# JRE
11jdk.home=C:\Program Files\Java\jdk-19
12
13# WebServer
14server.home=C:\Users\Cristina\Desktop\apache-tomcat-9.0.85
15deploy.path=${server.home}/webapps
16
```

On the right, the "Outline" view shows a tree structure of build steps for a project named "02_TecWeb". The steps include:

- > 00.init.properties
- > 01.set.classpath
- > 02.print.properties [default]
- > 03.clean
- > 04.prepare
- > 05.build
- > 05.build.tests
- > 06.test
- > 07.package
- > 08.undeploy
- > 09a.deploy.war
- > 09b.deploy.dir
- > 10.update.pages
- > 99.run.tcptunnel

SERVLET e descrittori XML



(Nota: Impostare correttamente il Build-path, aggiungere i jar delle cartelle /lib)

L'archivio .WAR (creato tramite ANT) contiene le nostre Servlet.

Il Servlet Container (TOMCAT) non conosce a quali URL devono essere associate!

Inoltre, l'utente non può sapere da quale pagina iniziare la navigazione.

Si usano i descrittori XML

→ web > WEB-INF > web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" version="2.4">

  <!-- 1) General -->
  <!-- Name the application -->
  <display-name>02_TecWeb</display-name>
  <description> A servlet-based project to use
    as a template for your owns </description>

  <!-- 2) Servlets -->
  <servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>
      it.unibo.tw.web.HelloWorldServlet
    </servlet-class>
  </servlet>

  <!-- Map some URL's to the servlet -->
  <servlet-mapping>
    <servlet-name>HelloWorld</servlet-name>
    <url-pattern>/helloworld</url-pattern>
  </servlet-mapping>

  <!-- 3) Welcome Files -->
  <!-- Define, in order of preference, which file to
    show when no filename is defined in the path -->
  <welcome-file-list>
    <welcome-file>test.html</welcome-file>
    <welcome-file>home.html</welcome-file>
  </welcome-file-list>

  <!-- 4) Error Handler -->
  <!-- Define an error handler for 404 pages -->
  <error-page>
    <error-code>404</error-code>
    <location>/errors/notfound.html</location>
  </error-page>
  <!-- Define an handler for java.lang.Exception -->
  <error-page>
    <exception-type>
      java.lang.Exception
    </exception-type>
    <location>/errors/exception.html</location>
  </error-page>
</web-app>
```

2 → Mappo la servlet ad un URL specifico

3 → Se nel URL web non viene specificato alcun nome

ES: http://localhost:8080/02_TecWeb/

Di default viene caricata la pagina web > home.html

http://localhost:8080/02_TecWeb/ > http://localhost:8080/02_TecWeb/home.html

SERVLET esempio GET / POST

Le richieste che giungono alla nostra Servlet possono essere di tipo GET o POST.

Creiamo un progetto che realizza una Servlet in grado di rispondere come segue:

http GET:

- Presentazione di form per l'invio di testo al Server mediante POST
- Valorizzazione del campo di input del form con l'eventuale testo già inviato dall'utente

http POST:

- Visualizzazione del testo ricevuto nella pagina di risposta
- Memorizzazione e mantenimento del testo (stato)

Nota: due possibili soluzioni per lo stato: Sessione o Cookie

Creiamo una pagina **home.html** (*raggiunta di default > XML*)

che raggiunge la servlet tramite GET:

```
home.html
1<html>
2  <head>
3    <title>Start Web Application</title>
4    <!--
5      we redirect user to the faces page, but it takes time for the
6      server to instantiate the framework, at first...
7    -->
8    <meta http-equiv="Refresh" content= "0; URL=esession"/>
9
10   <link type="text/css" href="styles/default.css" rel="stylesheet"></link>
11
12 </head>
13 <body>
14   <p>
15     Please wait for the web application to start... &nbsp;
16     
17   </p>
18
19 </body>
20 </html>
```

home.html non viene visualizzato a meno di ritardi

Node	Content
xml	version="1.0" encoding="ISO-8859-1"
web-app	((description*, display-name*, icon*)) distributable context-param filter filter-mapping listener servlet...
xmlns	http://java.sun.com/xml/ns/j2ee
xmlnsxsi	http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation	http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd
version	2.4
!	1) General
!	Name the application
!	02_TecWeb
!	A servlet-based project to use as a template for your own
!	2) Servlets
!	((description*, display-name*, icon*), servlet-name, (servlet-class jsp-file), init-param*, load-on-startup?, ru...
!	Identification
!	EsSession
!	it.unibo.tw.web.EsSessionServlet
!	Map some URL's to the servlet
!	(servlet-name, url-pattern)
!	EsSession
!	/esession
!	3) Welcome Files
!	Define, in order of preference, which file to show when no filename is defined in the path
!	(welcome-file+)
!	home.html
!	4) Error Handler
!	Define an error handler for 404 pages
!	((error-code exception-type), location)
!	Define an error handler for java.lang.Exception
!	((error-code exception-type), location)

SERVLET esempio GET / POST

Creiamo la Servlet (classe Java) EsSessionServlet

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class EsSessionServlet extends HttpServlet{

    private static final long serialVersionUID = 1L;

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {
        String testo = (String) request.getSession().getAttribute("testo");
        if(testo == null) testo = "";

        PrintWriter out = response.getWriter();
        out.println("<html>");

        out.println("<head>");
        out.println("<title>Servlet e Sessione</title>");
        out.println("<link rel=\"stylesheet\" href=\"styles/default.css\" type=\"text/css\"></link>");
        out.println("</head>");

        out.println("<body>");
        // default action targets the current URL
        out.println("<form method=\"post\" />");
        out.println("Scrivi qualcosa da ricordare:<br/><br/>");
        out.println("<input type=\"text\" name=\"testo\" value=\"" + testo + "\" />");
        out.println("<input type=\"submit\" value=\"Invia!\" />");
        out.println("</form>");
        out.println("</body>");

        out.println("</html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        // retrieve former value
        String testo = request.getParameter("testo");
        if ( testo == null ) testo = "";

        PrintWriter out = response.getWriter();

        out.println("<html>");

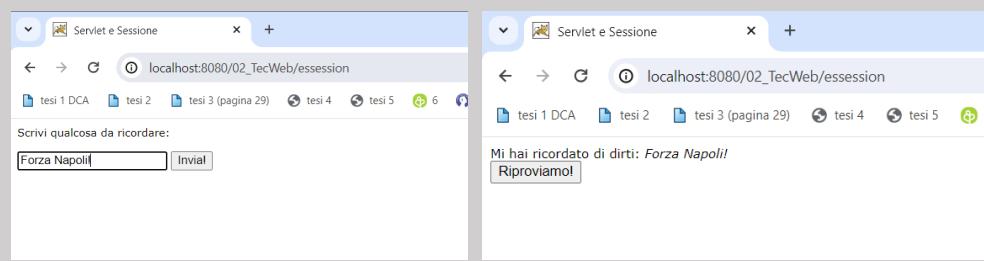
        out.println("<head>");
        out.println("<title>Servlet e Sessione</title>");
        out.println("<link rel=\"stylesheet\" href=\"styles/default.css\" type=\"text/css\"></link>");
        out.println("</head>");

        out.println("<body>");
        out.println("Mi hai ricordato di dirti: <i>" + testo + "</i>");
        // default action targets the current URL
        out.println("<form method=\"get\" />");

        // store the value as a session attribute
        request.getSession().setAttribute("testo", testo);

        out.println("<input type=\"submit\" value=\"Riproviamo!\" />");
        out.println("</form>");

        out.println("</body>");
        out.println("</html>");
    }
}
```



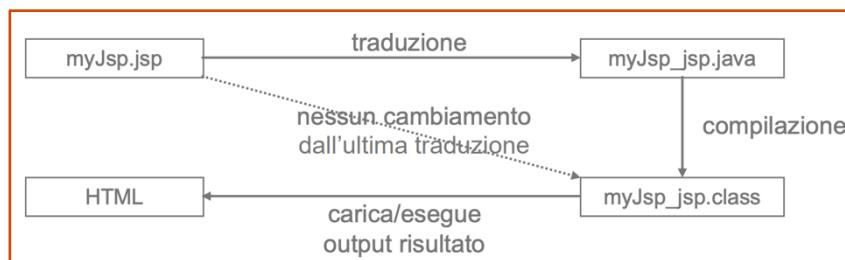
8 JSP

Le **Java Server Pages** sono uno dei due componenti di base della tecnologia delle pagine web dinamiche: estendono HTML con codice Java custom.

Oltre al contenuto HTML presentano tag JSP che includono blocchi di codice Java eseguiti runtime.

Le richieste verso JSP sono gestite da una particolare servlet (in Tomcat si chiama JspServlet) che effettua le seguenti operazioni:

- traduzione della JSP in una servlet
- compilazione della servlet risultante in una classe
- esecuzione della JSP



Il codice risultante è equivalente a quello di una servlet.

Il loro ciclo di vita è controllato sempre dal medesimo Web Container.

Perché usare JSP?

- *NOTA: nessuna differenza di potenza di espressione da una servlet*
- *Facilitano la progettazione*
- *Web designer possono produrre pagine senza dover conoscere i dettagli della logica server-side*
- *La generazione di codice dinamico è implementata sfruttando il linguaggio Java*
- *Rendono molto semplice presentare documenti HTML o XML (o loro parti) all'utente*
- *Non necessitano di compilazione (codice dentro un .war)*

Le servlet forniscono agli sviluppatori un completo controllo dell'applicazione. Se si vogliono fornire contenuti differenziati a seconda di diversi parametri quali l'identità dell'utente, condizioni dipendenti dalla business logic, etc. è conveniente continuare a lavorare con le servlet.

8.1 Sintassi e tag

Ogni volta che arriva una request, il server compone dinamicamente il contenuto della pagina.
Ogni volta che incontra un tag JSP valuta l'espressione Java contenuta al suo interno e inserisce al suo posto il risultato dell'espressione.



Sono possibili due tipi di sintassi per questi tag:

- **Scripting-oriented tag**
- **XML-Oriented tag**

→ **XML-Oriented tag** ←

- <jsp:declaration>declaration</jsp:declaration>
- <jsp:expression>expression</jsp:expression>
- <jsp:scriptlet>java_code</jsp:scriptlet>
- <jsp:directive.dir_type dir_attribute />

→ **Scripting-oriented tag** ←

- **DICHIARAZIONI <%! %>**

Si usano per dichiarare variabili e metodi; questi possono poi essere referenziati in qualsiasi punto del codice. Quando la pagina viene tradotta diventano metodi della servlet.

```
<%!
String name = "Paolo Rossi";
double[] prices = {1.5, 76.8, 21.5};

double getTotal() {
    double total = 0.0;
    for (int i=0; i<prices.length; i++)
        total += prices[i];
    return total;
}
%>
```

▪ ESPRESSIONI <%= %>

Tra questi delimitatori un'espressione viene valutata e il risultato, convertito in stringa, inserito al posto del tag.

JSP

```
<p>Sig. <%=name%>, </p>
<p>l'ammontare del suo acquisto è: <%=getTotal()%> euro. </p>
<p>La data di oggi è: <%=new Date()%></p>
```



Pagina HTML risultante

```
<p>Sig. Paolo Rossi, </p>
<p>l'ammontare del suo acquisto è: 99.8 euro. </p>
<p>La data di oggi è: Tue Feb 20 11:23:02 2010</p>
```

▪ SCRIPTLET <% %>

Sono usate per aggiungere un frammento di codice Java eseguibile dalla JSP. Sono tipicamente usate per il controllo di flusso, la combinazione di tutti gli scriptlet deve definire un blocco logico completo di codice Java.

```
<% if (userIsLogged) { %>
    <h1>Benvenuto Sig. <%=name%></h1>
<% } else { %>
    <h1>Per accedere al sito devi fare il login</h1>
<% } %>
```

▪ DIRETTIVE <%@ %>

Sono comandi JSP valutati a tempo di compilazione.

Le più importanti sono:

- **page**: permette di importare package, dichiarare pagine d'errore, definire modello di esecuzione JSP relativamente alla concorrenza (ne discuteremo a breve), ecc.
- **include**: include un altro documento.
- **taglib**: carica una libreria di custom tag implementate dallo sviluppatore.

(Non producono nessun output visibile)

```
<%@ page info="Esempio di direttive" %>
<%@ page language="java" import="java.net.*" %>
<%@ page import="java.util.List, java.util.ArrayList" %>
<%@ include file="myHeaderFile.html" %>
```

NOTA: direttiva page

La direttiva page definisce una serie di attributi che si applicano all'intera pagina.

```
<%@ page
[ language="java" ]
[ extends="package.class" ]
[ import="{package.class | package.*}, ..." ]
[ session="true | false" ]
[ buffer="none | 8kb | sizekb" ]
[ autoFlush="true | false" ]
[ isThreadSafe="true | false" ]
[ info="text" ]
[ errorPage="relativeURL" ]
[ contentType="mimeType [ ;charset=characterSet ]" |
  "text/html ; charset=ISO-8859-1" ]
[ isErrorPage="true | false" ]
%>
```

N.B. valori sottolineati sono quelli di default

- language="java" linguaggio di scripting utilizzato nelle parti dinamiche, allo stato attuale l'unico valore ammesso è "java"
- import="{package.class|package.*},..." lista di package da importare. Gli import più comuni sono impliciti e non serve inserirli (java.lang.*, javax.servlet.*, javax.servlet.jsp.*, javax.servlet.http.*)
- session="true|false" : indica se la pagina fa uso della sessione (altrimenti non si può usare session)
- buffer="none|8kb|sizekb" dimensione in KB del buffer di uscita
- autoFlush="true|false" dice se il buffer viene svuotato automaticamente quando è pieno.
- isThreadSafe="true|false" indica se il codice contenuto nella pagina è thread-safe. Se vale false le chiamate alla JSP vengono serializzate. Ricordate il modello single-threaded per le servlet?
- info="text" testo di commento. Può essere letto con il metodo Servlet.getServletInfo()
- errorPage="relativeURL" indirizzo della pagina a cui vengono inviate le eccezioni
- isErrorPage="true|false" indica se JSP corrente è una pagina di errore. Si può utilizzare l'oggetto eccezione solo se l'attributo è true
- contentType="mimeType [;charset=charSet]" | "text/html; charset=ISO-8859-1" indica il tipo MIME e il codice di caratteri usato nella risposta

NOTA: direttiva include

Serve ad includere il contenuto del file specificato.

È possibile nidificare un numero qualsiasi di inclusioni. L'inclusione viene fatta a tempo di compilazione: eventuali modifiche al file incluso non determinano una ricompilazione della pagina che lo include.

(Esempio: <%@ include file="/shared/copyright.html">)

NOTA: direttiva taglib

Una taglib è una collezione di tag non standard, realizzata mediante una classe Java.

(Sintassi: <%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>)

8.2 Built-in objects

Le specifiche JSP definiscono 9 oggetti built-in (o impliciti) utilizzabili senza dover creare istanze. Rappresentano utili riferimenti ai corrispondenti oggetti Java veri e propri presenti nella tecnologia servlet.

Oggetto	Classe/Interfaccia
page	javax.servlet.jsp.HttpJspPage
config	javax.servlet.ServletConfig
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
out	javax.servlet.jsp.JspWriter
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
pageContext	javax.servlet.jsp.PageContext
exception	Java.lang.Throwable

L'oggetto **page** rappresenta l'istanza corrente della servlet.

Ha come tipo l'interfaccia *HTTPJspPage* che discende da JSP page, la quale a sua volta estende *Servlet*. Può quindi essere utilizzato per accedere a tutti i metodi definiti nelle servlet.

JSP

```
<%@ page info="Esempio di uso page." %>
<p>Page info:
  <%=page.getServletInfo() %>
</p>
```

Pagina HTML

```
<p>Page info: Esempio di uso di page</p>
```

L'oggetto **config** contiene la configurazione della servlet (parametri di inizializzazione).

Metodi utili:

- **getInitParameterName()** restituisce tutti i nomi dei parametri di inizializzazione
- **getInitParameter(name)** restituisce il valore del parametro passato per nome

L'oggetto **request** rappresenta la richiesta alla pagina JSP.

È il parametro request passato al metodo *service()* della servlet. Consente l'accesso a tutte le informazioni relative alla richiesta HTTP: indirizzo, URL, headers, cookie, parametri, ecc...

Metodi utili:

- **String getParameter(String parName)**
restituisce valore di un parametro individuato per nome
- **Enumeration getParameterNames()**
restituisce l'elenco dei nomi dei parametri
- **String getHeader(String name)**
restituisce il valore di un header individuato per nome sotto forma di stringa
- **Enumeration getHeaderNames()**
elenco nomi di tutti gli header presenti nella richiesta
- **Cookie[] getCookies()**
restituisce un array di oggetti cookie che client ha inviato alla request

L'oggetto **response** è quello legato all'I/O della pagina JSP.

Rappresenta la risposta che viene restituita al client, consente di inserire nella risposta diverse informazioni: content type ed encoding, eventuali header di risposta, URL Rewriting, i cookie...

Metodi utili:

- **public void setHeader(String headerName, String headerValue)**

imposta header

- **public void setDateHeader(String name, long millisecs)**

imposta data

- **addHeader, addDateHeader, addIntHeader**

aggiungono nuova occorrenza di un dato header

- **setContentType**

determina content-type

- **addCookie**

consente di gestire i cookie nella risposta

- **public PrintWriter getWriter**

restituisce uno stream di caratteri (un'istanza di PrintWriter)

- **public ServletOutputStream getOutputStream()**

restituisce uno stream di byte (un'istanza di ServletOutputStream)

L'oggetto **out** è uno stream di caratteri e rappresenta lo stream di output della pagina.

Metodi utili:

- **isAutoFlush()** dice se output buffer è stato impostato in modalità autoFlush o meno
- **getBufferSize()** restituisce dimensioni del buffer
- **getRemaining()** quanti byte liberi nel buffer
- **clearBuffer()** ripulisce il buffer
- **flush()** forza l'emissione del contenuto del buffer
- **close()** fa flush e chiude stream

```
<p>Conto delle uova
<%
    int count = 0;
    while (carton.hasNext())
    {
        count++;
        out.print(".");
    }
%>
<br/>
Ci sono <%= count %> uova.
</p>
```

L'oggetto **session** fornisce informazioni sul contesto di esecuzione della JSP in termini di sessione utente. L'attributo session della direttiva page deve essere true affinché JSP partecipi alla sessione.

```
<% UserLogin userData = new UserLogin(name, password);
   session.setAttribute("login", userData);
%>
<%UserLogin userData=(UserLogin)session.getAttribute("login");
   if (userData.isGroupMember("admin")) {
       session.setMaxInactiveInterval(60*60*8);
   } else {
       session.setMaxInactiveInterval(60*15);
   }
%>
```

Metodi utili:

- **String getId()** restituisce ID di una sessione

- **boolean isNew()** dice se sessione è nuova

- **void invalidate()** permette di invalidare (distruggere) una sessione

- **long getCreationTime()** ci dice da quanto tempo è attiva la sessione (in ms)

- **long getLastAccessedTime()** ci dice quando è stata utilizzata l'ultima volta

L'oggetto **application** fornisce informazioni su contesto di esecuzione della JSP con scope di visibilità comune a tutti gli utenti. Rappresenta la Web application a cui JSP appartiene, consente di interagire con l'ambiente di esecuzione.

L'oggetto **pageContext** fornisce informazioni sul contesto di esecuzione della pagina JSP. Rappresenta l'insieme degli oggetti built-in di una JSP, consente accesso a tutti gli oggetti impliciti e ai loro attributi.

L'oggetto **exception** è connesso alla gestione degli errori.

Rappresenta l'eccezione che non viene gestita da nessun blocco catch; non è automaticamente disponibile in tutte le pagine ma solo nelle Error Page.

```
<%@ page isErrorPage="true" %>
<h1>Attenzione!</h1>
E' stato rilevato il seguente errore:<br/>
<b><%= exception %></b><br/>
<%
    exception.printStackTrace(out);
%>
```

8.3 Azioni

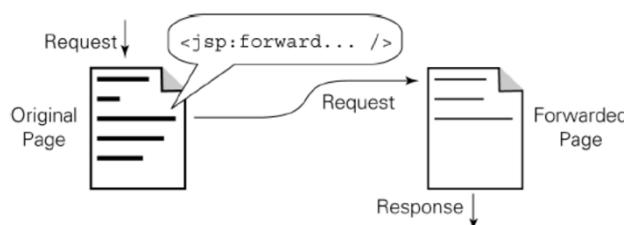
Le **azioni** sono comandi JSP tipicamente per l'interazione con altre pagine JSP, servlet, o componenti JavaBean. Sono espresse usando sintassi XML e prevedono 6 tipi di azioni definite dai seguenti tag:

- **useBean**: istanzia JavaBean e gli associa un identificativo
- **getProperty**: ritorna property indicata come oggetto
- **setProperty**: imposta valore della property indicata per nome
- **include**: include nella JSP contenuto generato dinamicamente da un'altra pagina locale
- **forward**: cede controllo ad un'altra JSP o servlet
- **plugin**: genera contenuto per scaricare plug-in Java se necessario

```
<html>
<body>
<jsp:useBean id="myBean" class="it.unibo.deis.my.HelloBean"/>
<jsp:setProperty name="myBean" property="nameProp" param="value"/>
Hello, <jsp:getProperty name="myBean" property="nameProp"/>!
</body>
</html>
```

→ **forward** sintassi: *<jsp:forward page="local URL" />*

Consente trasferimento del controllo dalla pagina JSP corrente ad un'altra pagina sul server locale. L'attributo page definisce l'URL della pagina a cui trasferire il controllo, la request viene completamente trasferita in modo trasparente per il client.



È possibile generare dinamicamente l'attributo page:

`<jsp:forward page='<%="message"+statusCode+".html"%>' />`

Oggetti request, response e session della pagina d'arrivo sono gli stessi della pagina chiamante, ma viene istanziato un nuovo oggetto *pageContext*.

Attenzione: forward è possibile soltanto se non è stato emesso alcun output!

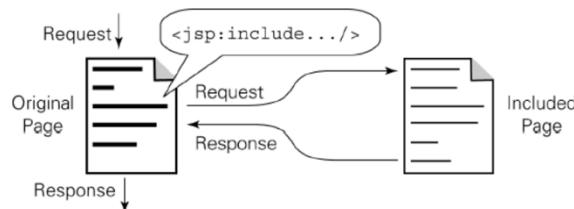
È possibile aggiungere parametri all'oggetto request della pagina chiamata utilizzando il tag `<jsp:param>`.

```
<jsp:forward page="localURL">
  <jsp:param name="parName1" value="parValue1"/>
  ...
  <jsp:param name="parNameN" value="parValueN"/>
</jsp:forward>
```

→ **include** sintassi: `<jsp:include page="local URL" flush="true" />`

Consente di includere il contenuto generato dinamicamente da un'altra pagina locale all'interno dell'output della pagina corrente. Trasferisce temporaneamente controllo ad un'altra pagina.

L'attributo page definisce l'URL della pagina da includere, mentre l'attributo flush stabilisce se sul buffer della pagina corrente debba essere eseguito flush prima di effettuare l'inclusione.



Gli oggetti session e request per pagina da includere sono gli stessi della pagina chiamante, ma viene istanziato un nuovo contesto di pagina.

È possibile aggiungere parametri all'oggetto request utilizzando il tag `<jsp:param>`.

8.4 Modello a componenti

Scriptlet ed espressioni consentono uno sviluppo centrato sulla pagina.

Questo modello non consente una forte separazione tra logica applicativa e presentazione dei contenuti. Applicazioni complesse necessitano di maggiore modularità ed estensibilità, tramite una architettura a più livelli.

A tal fine, JSP consente anche uno sviluppo basato su un **modello a componenti**...

→ maggiore separazione fra logica dell'applicazione e contenuti

→ costruire architetture molto più articolate

Reminder:

Un **JavaBean**, o semplicemente **bean**, non è altro che una classe Java dotata di alcune caratteristiche particolari:

- Classe public
- Ha un costruttore public di default (senza argomenti)
- Espone proprietà, sotto forma di coppie di metodi di accesso (accessors) costruiti secondo le regole che abbiamo appena esposto (get... set...)
- Espone eventi con metodi di registrazione che seguono regole precise

La proprietà *prop* è definita da due metodi *getProp()* e *setProp()*.

Il tipo del parametro di *setProp()* e del valore di ritorno di *getProp()* devono essere uguali e rappresentano il tipo della proprietà.

Le proprietà di tipo boolean seguono una regola leggermente diversa: metodo di lettura ha la forma *isProp()*.

I componenti vivono all'interno di contenitori (component container) che gestiscono:

- tempo di vita dei singoli componenti
- collegamenti fra componenti e resto del sistema

Un *bean container* è in grado di interfacciarsi con i bean utilizzando Java Reflection che fornisce strumenti di introspezione e di dispatching.

JSP prevedono una serie di tag per agganciare un bean e utilizzare le sue proprietà all'interno della pagina. Questa architettura è denominata **Model 1**. I tag possono essere di tre tipi:

Tag per creare un **riferimento al bean** (creazione di un'istanza)

Sintassi: `<jsp:useBean id="beanName" class="class" scope="page|request|session|application"/>`

- **id** è il nome con cui l'istanza del bean verrà indicata nel resto della pagina
- **class** è classe Java che definisce il bean
- **scope** definisce ambito di accessibilità e tempo di vita dell'oggetto (default = page)

Con l'attributo scope è possibile estendere la vita del bean oltre la singola richiesta

Tag per **impostare il valore delle proprietà** del bean

Sintassi: `<jsp:setProperty name="beanName" property="propName" value="propValue"/>`

Consente di modificare il valore delle proprietà del bean

I tag per JavaBean non supportano proprietà indicizzate.

Tag per **leggere il valore delle proprietà del bean e inserirlo nel flusso della pagina**

Sintassi: `<jsp:getProperty name="beanName" property="propName"/>`

- **name** nome del bean a cui si fa riferimento
- **property** nome della proprietà di cui si vuole leggere il valore

Produce come output il valore della proprietà del bean

Esempio:

```
Bean
import java.util.*
public class CurrentTimeBean {
    private int hours;
    private int minutes;
    public CurrentTimeBean() {
        Calendar now = Calendar.getInstance(); this.hours =
                                                now.get(Calendar.HOUR_OF_DAY);
        this.minutes = now.get(Calendar.MINUTE);
    }
    public int getHours() { return hours; }
    public int getMinutes() { return minutes; }
}
```

JSP

```
<jsp:useBean id="time" class="CurrentTimeBean"/>
<html>
<body>
<p> Sono le ore
    <jsp:getProperty name="time" property="hours"/> e
    <jsp:getProperty name="time" property="minutes"/> minuti.
</p>
</body>
</html>
```

[*Discorso duale in caso di set:*

```
<jsp:setProperty name="user" property="nome" value="Alessandro"/>]
```

NOTA:

Un bean è un normale oggetto Java: è quindi possibile accedere a variabili e metodi:

```
<jsp:useBean id="weather" class="weatherForecasts"/>

<p><b>Previsioni per domani:</b>:
   <%= weather.getForecasts(0) %>
</p>
<p><b>Resto della settimana:</b>
<ul>
   <% for (int index=1; index < 5; index++) { %>
      <li><%= weather.getForecasts(index) %></li>
   <% } %>
</ul>
</p>
```

JSP e Servlet

Date due pagine .jsp (pagine html con codice Java JSP)

gestisciCliente.jsp

statistiche.jsp

gestisciCliente permette ad un cassiere di inserire gli ordini effettuati in una classe java (bean) InsiemeDiArticoli formata da un vettore di Articolo (che tiene conto: prezzo, quantità, data...).

InsiemeDiArticoli.java

private Vector<Articolo> merce

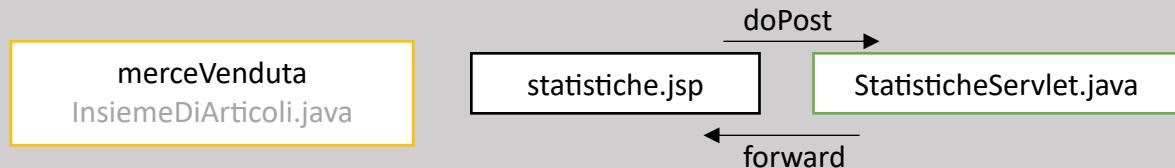
Articolo.java

private String id

private float price

...

statistiche.jsp interroga l'istanza merceVenduta tramite una servlet raggiunta attraverso un form (POST). La Servlet calcola gli incassi di un articolo, se specificato, o di tutti gli articoli.



Codice 03a_TecWeb, nota:

Uso del ContextPath tramite jsp + Istanziare classi JavaBean tramite useBean /gestisciClienti

```
<body>
    <h3>Gestisci Cliente</h3>
    <a href="<%=>request.getContextPath()%>/statistiche.jsp">Vai a statistiche.jsp</a><br />
    <br />

    <jsp:useBean id="merceSelezionata" class="it.unibo.tw.es1.beans.InsiemeDiArticoli" scope="session" />
    <jsp:useBean id="merceVenduta" class="it.unibo.tw.es1.beans.InsiemeDiArticoli" scope="application" />
    ...
<%>
```

La Servlet riprende il bean merceVenduta /Servlet

```
String richiesta = req.getParameter("req");
if( richiesta!=null && richiesta.equals("calcola") ){
    InsiemeDiArticoli vendite = (InsiemeDiArticoli)this.getServletContext().getAttribute("merceVenduta")

    // recupero i parametri della ricerca
    Vector<Articolo> articoliVenduti = vendite.getMerce();
    String id = req.getParameter("id");
    int firstDay = Integer.parseInt(req.getParameter("firstDay"));

    ...
}
```

Torniamo il controllo alla pagina statistiche.jsp tramite forward

Il risultato finale va inserito come un attributo alla richiesta che verrà fatta alla jsp

```
// inserisco il risultato nella richiesta che viene passata alla JSP
req.setAttribute("guadagnoRichiestaAttuale", guadagno);
}

// passo il controllo alla JSP
RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/statistiche.jsp");
dispatcher.forward(req, resp);
```

JSP e Servlet, muoversi tra le pagine 03b_TecWeb

Definiamo tramite web.xml una “welcome-page” che reindirizza su una Servlet il nostro utente.



La servlet ha il solo scopo di indirizzare nuovamente l'utente alla vera home page.

Viene richiesto l'URL come parametro di inizializzazione...
questo è configurabile tramite web.xml!

```
<servlet>
    <servlet-name>StillAServlet</servlet-name>
    <servlet-class>it.unibo.tw.web.servlets.StillAServlet</servlet-class>
    <init-param>
        <param-name>homeURL</param-name>
        <param-value>/pages/home.jsp</param-value>
    </init-param>
</servlet>
```

```
public class StillAServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    private String homeURL = null;

    @Override
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        this.homeURL = config.getInitParameter("homeURL");
    }

    @Override
    public void service(ServletRequest req, ServletResponse resp)
    throws ServletException, IOException {
        // tempo di attesa, qui inserito artificiosamente
        try {
            Thread.sleep(5000);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        // un altro forward eseguito lato servlet
        req.getRequestDispatcher(homeURL).forward(req, resp);
    }
}
```

La vera home fa uso di altre pagine jsp includendole nella propria

```
home.jsp × checkout.jsp catalogue.jsp cart.jsp
1<html>
2    <head>
3        <meta name="Author" content="pisi79">
4        <title>Home JSP</title>
5        <link rel="stylesheet" href="<%= request.getContextPath() %>/styles/default.css" type="text/css"/>
6    </head>
7
8    <body>
9
10       <%@ include file="../fragments/header.jsp" %>
11       <%@ include file="../fragments/menu.jsp" %>
12
13       <div id="main" class="clear">
14           <p>Welcome to the 'TemplateJSP' project.</p>
15           <br/>
16           <p>You can have lot of fun, here!</p>
17       </div>
18
19       <%@ include file="../fragments/footer.jsp" %>
20
21   </body>
22 </html>
```

9 JavaScript

JavaScript è un linguaggio di scripting sviluppato per dare interattività LATO CLIENTE alle pagine HTML. Al di là del nome, Java e JavaScript sono due linguaggi completamente diversi...

- JavaScript è **interpretato** e non compilato
- JavaScript è **object-based** ma non class-based
 - (Esiste il concetto di oggetto, non esiste il concetto di classe)
- JavaScript è **debolmente tipizzato**

Il codice JavaScript viene eseguito da un interprete contenuto all'interno del browser; nasce infatti per dare dinamicità alle pagine Web (pagine attive). Consente dunque di accedere/modificare elementi HTML, reagire ad eventi, validare dati ed interagire con il browser.

Uno script viene inserito all'interno di una pagina HTML tramite tag: **<script>**

La sintassi di JavaScript è modellata su quella del C con alcune varianti significative

- È un linguaggio case-sensitive
- Le istruzioni sono terminate da ';' ma il terminatore può essere omesso se si va a capo
- Sono ammessi sia commenti multilinea (delimitati da /* e */) che mono-linea (iniziano con //)
- Gli identificatori possono contenere lettere, cifre e i caratteri '_' e '\$' ma non possono iniziare con una cifra

9.1 Variabili e Oggetti

Le **variabili** vengono dichiarate usando la parola chiave **var**:

- Non hanno un tipo
- Possono essere inizializzate contestualmente alla dichiarazione
- Possono essere dichiarate in linea
- Esiste scope globale, locale (ad una funzione) ma non di blocco

var nomevariabile;

Esistono due valori speciali:

null come in SQL (diverso da 0 o "")

undefined variabile non inizializzata

Javascript prevede pochi tipi primitivi:

Numeri – number

Rappresentati in formato floating point 8 byte, non presentano distinzione tra interi e reali.

Esistono i valori speciali **Nan** e **infinite**.

Booleani – boolean

true o **false**.

Le variabili assumono in realtà un "tipo" dinamicamente in base al dato a cui vengono agganciate.

Gli **oggetti** sono tipi composti che contengono un certo numero di attributi.

- Ogni attributo ha nome e valore
- Si accede agli attributi tramite .
- Le proprietà possono essere aggiunte dinamicamente
(Object() è un costruttore, non una classe)

var o = new Object()

Costruire un oggetto:

```
var o = new Object();
o.x = 5;
o.y = 2;
o.tot = o.x + o.y;
```

Costanti oggetto:

```
var o = {prop1:val1, prop2:val2, ...}
```

Gli **array** sono tipi composti i cui elementi sono accessibili mediante un indice numerico.

- Non hanno dimensione prefissata
- Espongono metodi e attributi
- Possono essere inizializzati usando costanti delimitate da []
- Possono contenere elementi di tipo eterogeneo

new Array([dimensione])

opzionale

Osservazione:

Gli oggetti sono in realtà array associativi.

o.x = 7 == o["x"] = 7

Le **stringhe** sono sequenze arbitrarie di caratteri in formato UNICODE a 16 bit e immutabili come in Java. Esiste la possibilità di definire costanti stringa (*string literal*) delimitate da apici singoli ('ciao') o doppi ("ciao").

- Ne è possibile inoltre la concatenazione tramite operatore + e la comparazione (<, >, <=, >=, !=)
- Possiamo invocarne i metodi (ES s.length;)
- Accedere agli attributi (ES s.charAt(1);)

Non sono però oggetti e la possibilità di trattarli come tali nasce da due caratteristiche:

- Esiste un tipo wrapper **String** che è un oggetto
- JavaScript fa il boxing in automatico

9.2 Funzioni e Costruttori

JavaScript ha un supporto per le **espressioni regolari** (regular expressions) che sono un tipo di dato nativo del linguaggio. Esistono le costanti di tipo espressione (`/ expression /`) oppure possono essere create mediante costruttore (`RegExp()`).

Una **funzione** è un frammento di codice JavaScript che viene definito una volta e usato in più punti.

→ Ammette parametri che sono privi di tipo

→ Restituisce un valore il cui tipo non viene definito

Definite usando la parola chiave **function**, possono essere assegnate ad una variabile.

```
function sum(x,y)
{
    return x+y;
}
var s = sum(2,4);
```

NOTA:

Esistono **costanti funzione** (function literal) che permettono di definire una funzione e poi di assegnarla ad una variabile con una sintassi decisamente inusuale:

```
var sum =
    function(x,y) { return x+y; }
```

Una funzione può essere anche creata usando un costruttore denominato **Function**:

```
var sum =
    new Function("x","y","return x+y;");
```

Quando una funzione viene assegnata ad una proprietà di un oggetto viene chiamata **metodo** dell'oggetto. In questo caso all'interno della funzione si può utilizzare la parola chiave **this** per accedere all'oggetto di cui la funzione è una proprietà

```
var o = new Object();
o.x = 7;
o.y = 8;
o.tot = function() { return this.x + this.y; }
alert(o.tot());
```

Un **costruttore** è una funzione che ha come scopo quello di costruire un oggetto.

Se viene invocato con **new** riceve l'oggetto appena creato e può aggiungere proprietà e metodi. L'oggetto da costruire è accessibile con **this**.

```
function Rectangle(w, h)
{
    this.w = w;
    this.h = h;
    this.area = function()
        { return this.w * this.h; }
    this.perimeter = function()
        { return 2*(this.w + this.h); }
}
var r = new Rectangle(5,4);
alert(r.area());
```

JavaScript ammette l'esistenza di proprietà e metodi statici con lo stesso significato di Java. Per esempio, se abbiamo definito il costruttore `Circle()` che serve per creare oggetti di tipo cerchio, possiamo aggiungere l'attributo PI in questo modo:

```
function Circle(r)
{
    this.r = r;
}
Circle.PI = 3.14159;
```

Osservazione:

Anche in Javascript esiste l'oggetto **Math** che definisce solo metodi statici corrispondenti alle varie funzioni matematiche

9.3 Operatori ed istruzioni

JavaScript ammette tutti gli operatori presenti in C e in Java.

Esistono alcuni operatori tipici:

- **delete**: elimina una proprietà di un oggetto
- **void**: valuta un'espressione senza restituire alcun valore
- **typeof**: restituisce il tipo di un operando
- **==**: identità o uguaglianza stretta (diverso da == che verifica l'egualità)
- **!=**: non identità (diverso da !=)

if/else, switch, while, do/while e for funzionano come in C e Java.

La struttura for/in permette di scorrere le proprietà di un oggetto (e quindi anche un array) con la sintassi: `for (variable in object) statement`

In JavaScript esiste un oggetto globale隐式的. Tutte le variabili e le funzioni definite in una pagina appartengono all'oggetto globale che espone alcune **funzioni predefinite**:

- **eval(expr)**

valuta la stringa expr (che contiene un'espressione Javascript)

- **isFinite(number)**

dice se il numero è finito

- **isNaN(testValue)**

dice se il valore è NaN

- **parseInt(str [, radix])**

converte la stringa str in un intero (in base radix - opzionale)

- **parseFloat(str)**

converte la stringa str in un numero

9.4 Note su <script>

```
<script> <!--script-text //--> </script>
```

Il commento HTML (`<!-- //-->`) che racchiude il testo dello script serve per gestire la compatibilità con i browser che non gestiscono JavaScript. La sintassi completa prevede anche la definizione del tipo di script definito:

```
<script type="text/javascript"> HTML4 (deprecato)
```

o

```
<script type="application/javascript"> HTML5
```

Script ESTERNO:

il tag contiene il riferimento ad un file con estensione .js che contiene lo script

```
<script type="application/javascript" scr="filescript.js">
```

Script INTERNO:

lo script è contenuto direttamente nel tag

Osservazione:

Una pagina HTML viene eseguita in ordine sequenziale, dall'alto verso il basso, per cui:

- gli script di intestazione vengono caricati prima di tutti gli altri
- quelli nel body vengono eseguiti secondo l'ordine di caricamento

Una variabile o qualsiasi altro elemento Javascript può essere richiamato solo se caricato in memoria. Quello che si trova nel body è visibile solo agli script che lo seguono.

Ci sono browser che non gestiscono JavaScript.

HTML prevede un tag **<noscript>** da inserire in testata per gestire contenuti alternativi in caso di non disponibilità di Javascript.

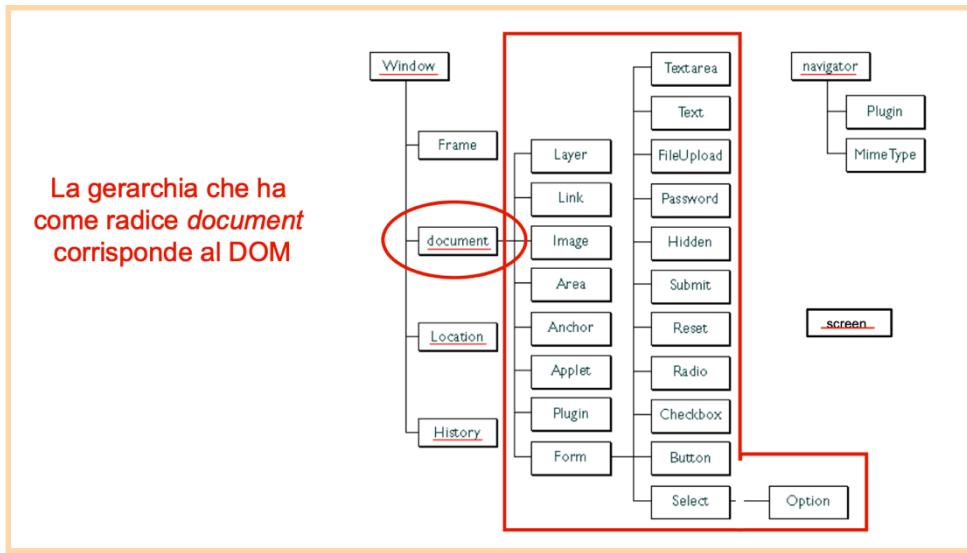
Ad esempio:

```
<noscript>
<meta http-equiv="refresh" content=
    "0;url=altrapagina.htm">
</noscript>
```

9.5 Utilità di JavaScript

- **Costruire dinamicamente parti della pagina** in fase di caricamento
- **Rilevare informazioni sull'ambiente** (tipo di browser, dimensione dello schermo, ecc.)
- **Rispondere ad eventi** generati dall'interazione con l'utente
- **Modificare dinamicamente il DOM** (si parla in questo caso di Dynamic HTML o DHTML)

Javascript utilizza una gerarchia di oggetti predefiniti denominati Browser Objects e DOM Objects:



La più semplice modalità di utilizzo di JavaScript consiste nell'inserire nel corpo della pagina script che generano dinamicamente parti della pagina HTML. Bisogna tener presente che questi script vengono eseguiti solo una volta durante il caricamento della pagina e quindi non si ha interattività con l'utente.

L'uso più comune è quello di generare pagine diverse in base al tipo di browser o alla risoluzione dello schermo. La pagina corrente è rappresentata dall'oggetto **document**; Per scrivere nella pagina si utilizzano i metodi `document.write()` e `document.writeln()`.

Per accedere ad informazioni sul browser si utilizza l'oggetto **navigator** che espone una serie di proprietà:

Proprietà	Descrizione
appCodeName	Nome in codice del browser (poco utile)
appName	Nome del browser (es. Microsoft Internet Explorer)
appVersion	Versione del Browser (es. 5.0 (Windows))
cookieEnabled	Cookies abilitati o no
platform	Piattaforma per cui il browser è stato compilato (es. Win32)
userAgent	Stringa passata dal browser come header user-agent (es. "Mozilla/5.0 (compatible; MSIE 9.0;)") È possibile esplorare la proprietà userAgent per mobile browser quali iPhone, iPad, o Android

L'oggetto **screen** permette di ricavare informazioni sullo schermo, espone alcune utili proprietà, tra cui segnaliamo **width** e **height** che permettono di ricavarne le dimensioni.

9.6 Eventi

Per avere una reale interattività bisogna utilizzare il meccanismo degli eventi. JavaScript consente di associare script agli eventi causati dall'interazione dell'utente con la pagina HTML; questi prendono il nome di gestori di eventi (*event handlers*).

Nelle risposte agli eventi si può intervenire sul DOM modificando dinamicamente la struttura della pagina: **DHTML = JavaScript + DOM + CSS**.

Evento	Applicabilità	Occorrenza	Event handler
Abort	Immagini	L'utente blocca il caricamento di un'immagine	onAbort
Blur	Finestre e tutti gli elementi dei form	L'utente toglie il focus a un elemento di un form o a una finestra	onBlur
Change	Campi di immissione di testo o liste di selezione	L'utente cambia il contenuto di un elemento	onChange
Click	Tutti i tipi di bottoni e i link	L'utente 'clicca' su un bottone o un link	onClick
DragDrop	Finestre	L'utente fa il drop di un oggetto in una finestra	onDragDrop
Error	Immagini, finestre	Errore durante il caricamento	onError
Focus	Finestre e tutti gli elementi dei form	L'utente dà il focus a un elemento di un form o a una finestra	onFocus
KeyDown	Documenti, immagini, link, campi di immissione di testo	L'utente preme un tasto	onKeyDown
KeyPress	Documenti, immagini, link, campi di immissione di testo	L'utente digita un tasto (pressione + rilascio)	onKeyPress
KeyUp	Documenti, immagini, link, campi di immissione di testo	L'utente rilascia un tasto	onKeyUp

Evento	Applicabilità	Occorrenza	Event handler
Load	Corpo del documento	L'utente carica una pagina nel browser	onLoad
MouseDown	Documenti, bottoni, link	L'utente preme il bottone del mouse	onMouseDown
MouseMove	Di default nessun elemento	L'utente muove il cursore del mouse	onMouseMove
MouseOut	Mappe, link	Il cursore del mouse esce fuori da un link o da una mappa	onMouseOut
MouseOver	Link	Il cursore passa su un link	onMouseOver
MouseUp	Documenti, bottoni, link	L'utente rilascia il bottone del mouse	onMouseUp
Move	Windows	La finestra viene spostata	onMove
Reset	Form	L'utente resetta un form	onReset
Resize	Finestre	La finestra viene ridimensionata	onResize
Select	Campi di immissione di testo (input e textarea)	L'utente seleziona il campo	onSelect
Submit	Form	L'utente sottomette il form	onSubmit
Unload	Corpo del documento	L'utente esce dalla pagina	onUnload

La sintassi di gestione è:

`<tag eventHandler="JavaScript Code">`

Ad esempio:

```
<input type="button" value="Calculate" onClick='alert("Calcolo")' />
```

NOTA: È sempre necessario alternare doppi apici e apice singolo!

9.7 Muoversi nel DOM

Il punto di partenza per accedere al Documento Object Model (DOM) della pagina è l'oggetto **document** che espone 4 collezioni di oggetti che rappresentano gli elementi di primo livello: **anchors[], forms[], images[], links[]**.

L'accesso agli elementi delle collezioni può avvenire per indice (ordine di definizione nella pagina) o per nome (attributo *name* dell'elemento):

```
document.links[0]  
document.links["nomelink"]  
document.nomelink
```

I metodi più utilizzati sono:

- **getElementById()**
restituisce un riferimento al primo oggetto della pagina avente l'id specificato come argomento
- **write()**
scrive un pezzo di testo nel documento
- **writeln()**
come write() ma aggiunge un a capo

Proprietà:

- **bgcolor**: colore di sfondo
- **fgcolor**: colore di primo piano
- **lastModified**: data e ora di ultima modifica
- **cookie**: tutti i cookie associati al document rappresentati da una stringa di coppie: nome-valore
- **title**: titolo del documento
- **URL**: url del documento

Esempio:

```
<!DOCTYPE html>
<html>
<body>

<p>Prompt Box e uso di document:</p>

<button onclick="myFunction()">Clicca qui</button>

<p id="demo"></p>

<script>
function myFunction() {
    var person = prompt("Inserisci il tuo nome:", "Harry Potter");
    if (person != null) {
        document.getElementById("demo").innerHTML =
        "Ciao " + person;
    }
}
</script>

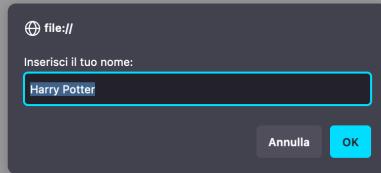
</body>
</html>
```

Prompt Box e uso di document:

Clicca qui

Prompt Box e uso di document:

Clicca qui



Prompt Box e uso di document:

Clicca qui

Ciao Harry Potter

9.8 Form

Un oggetto **form** può essere referenziato con il suo nome o mediante il vettore **forms []**.

Gli elementi possono essere referenziati con il loro nome o mediante il vettore **elements []**:

document.nomeForm.nomeElemento

document.forms[n].elements[m]

document.forms["nomeForm"].elements["nomeElem"]

Ogni elemento ha una proprietà **form** che permette di accedere al form che lo contiene:

this.form

Per ogni elemento del form esistono proprietà corrispondenti ai vari attributi:

id, name, value, type, className...

```
<form name="myForm">
  Form name:
  <input type="text" name="text1" value="test">
  <br/>
  <input name="button1" type="button"
    value="Mostra il nome del form"
    onclick="document.myForm.text1.value=
      document.myForm.name">
</form>
```

In alternativa potevamo scrivere:

```
onclick="this.form.text1.value=
  this.form.name">
```



Proprietà:

- **action**: riflette l'attributo action
- **elements**: vettore contenente gli elementi della form
- **length**: numero di elementi nella form
- **method**: riflette l'attributo method
- **name**: nome del form
- **target**: riflette l'attributo target

Metodi:

- **reset()**: resetta il form
- **submit()**: esegue il submit

Eventi:

- **onreset**: quando il form viene resettato
- **onsubmit**: quando viene eseguito il submit del form

Ogni tipo di controllo (**widget**) che può entrare a far parte di un form è rappresentato da un oggetto:

```
Text: <input type = "text">  
Checkbox: <input type="checkbox">  
Radio: <input type="radio">  
Button: <input type="button"> o <button>  
Hidden: <input type="hidden">  
File: <input type="file">  
Password: <input type="password">  
Textarea: <textarea>  
Submit: <input type="submit">  
Reset: <input type="reset">
```

Proprietà:

- **form**: riferimento al form che contiene il widget
- **name**: nome del widget (o controllo)
- **type**: tipo del controllo
- **value**: valore dell'attributo value
- **disabled**: disabilitazione/abilitazione del controllo

Metodi:

- **blur()** toglie il focus al controllo
- **focus()** dà il focus al controllo
- **click()** simula il click del mouse sul controllo

Eventi:

- **onblur**: quando il controllo perde il focus
- **onfocus**: quando il controllo prende il focus
- **onclick**: quando l'utente clicca sul controllo

NOTA – Text e Password

Proprietà (get/set):

- **defaultValue**: valore di default
- **disabled**: disabilitazione / abilitazione del campo
- **maxLength**: numero massimo di caratteri
- **readOnly**: sola lettura / lettura e scrittura
- **size**: dimensione del controllo

Metodi:

- **select()**: seleziona una parte di testo

NOTA – Checkbox e Radio

Proprietà (get/set):

- *checked*: dice se il box è spuntato
- *defaultChecked*: impostazione di default

Uno degli utilizzi più frequenti di JavaScript è nell'ambito della **validazione dei campi di un form**.

Riduce il carico delle applicazioni server side filtrando l'input, riduce il ritardo in caso di errori di inserimento dell'utente, semplifica le applicazioni server side.

Generalmente si valida un form in due momenti:

→ Durante l'inserimento utilizzando l'evento **onChange()** sui vari controlli.

La funzione si attiva ad ogni modifica del form

→ Al momento del submit utilizzando l'evento **onClick()** del bottone di submit
o l'evento **onSubmit()** del form.

Notiamo che *onClick()* viene eseguito prima del submit di un form (e relativa action).

Se *onSubmit()* ritorna *false* il form non è valido e non viene inviato.

JavaScript 04_TecWeb

I browser sono soliti a fare cache dei contenuti statici, per evitare ciò inserire le intestazioni:

```
<meta http-equiv="Pragma" content="no-cache" />
<meta http-equiv="Expires" content="-1" />
```

I messaggi di **popup** possono interrompere il normale flusso di rendering della pagina.

I più usati sono...

```
alert("Hello World!");
```

```
var capito = confirm("Capito?");
```

Restituisce un risultato true o false utilizzabile!

```
if(capito) {...} else {...}
```

```
var nome = prompt("Inserisci il tuo nome:");
```

Restituisce la stringa inserita in input!

L'evento **onload** della pagina può essere sfruttato per attaccare funzioni Javascript a parti del

DOM, in corrispondenza di eventi supportati (come .onmouseover/out).

```
window.onload = function(){ ... }
```

Creare una pagina html che calcoli la media voti inseriti da uno studente.

Usare un oggetto "Statistica", non è necessario un controllo dei valori inseriti.

The screenshot shows a code editor with two tabs: 'media.html' and 'media.js'. The 'media.html' tab contains the following HTML code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta http-equiv="Pragma" content="no-cache" />
5 <meta http-equiv="Expires" content="-1" />
6 <title>MediaVoti</title>
7 <link rel="stylesheet" href="../styles/default.css" type="text/css"/>
8 </head>
9
10 <body>
11 <script type="text/javascript" src="../scripts/media.js"></script>
12 <h1>Calcolo media voti</h1>
13 <p>Inserisci i tuoi voti separati da virgola e spazio<br>
14 Ad esempio: 30, 18, 25
15 </p>
16 <input type="button" value="Iniziamo!" onclick="calcola()"/>
17 <hr/>
18 <div id="res"></div>
19 </body>
20 </html>
```

The 'media.js' tab contains the following JavaScript code:

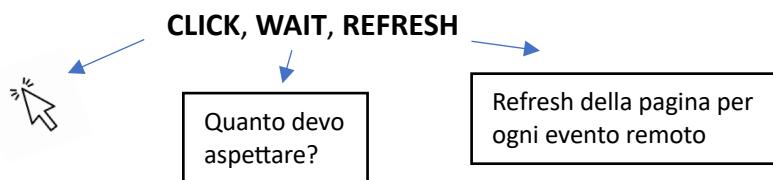
```
1 function calcola(){
2     var statistica = new Object();
3     var voti = prompt("inserisci i tuoi voti:");
4
5     statistica.voto = voti.split(", ");
6     statistica.media = function(){
7         var somma = 0;
8         var i;
9         for(i=0; i<statistica.voto.length; i++){
10             somma += parseInt(statistica.voto[i]);
11         }
12         var media = (somma/statistica.voto.length);
13         return media;
14     }
15
16     //Scrivo i risultati sul documento
17     //NOTA: il documento viene SOVRASCRITTO
18     document.writeln("Media: " + statistica.media());
19
20 }
```

10 Ajax

Arrivati a questo punto abbiamo individuato a livello concettuale due livelli di eventi:

<u>Eventi Locali</u>	<u>Eventi Remoti</u>
Implicano una modifica locale e diretta del DOM (Javascript)	Implicano il ricaricamento della pagina che viene modificata lato server: "postback" (GET/POST)

Essendo abituati ad un alto livello di interazione, le applicazioni tradizionali espongono un modello rigido e limitante, il modello...

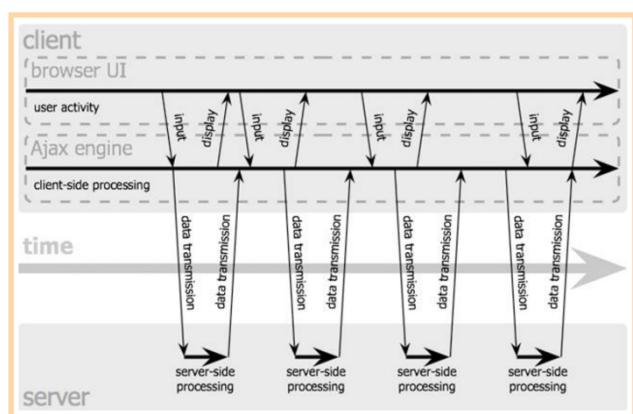


Il modello **AJAX** (Asynchronous Javascript And Xml) nasce con lo scopo di superare queste limitazioni date da un modello sincrono. Punta a supportare applicazioni user friendly con elevata interattività; l'idea di base è quella di consentire agli script JavaScript di interagire direttamente con il server.

L'elemento centrale è l'utilizzo dell'oggetto JavaScript **XMLHttpRequest**.

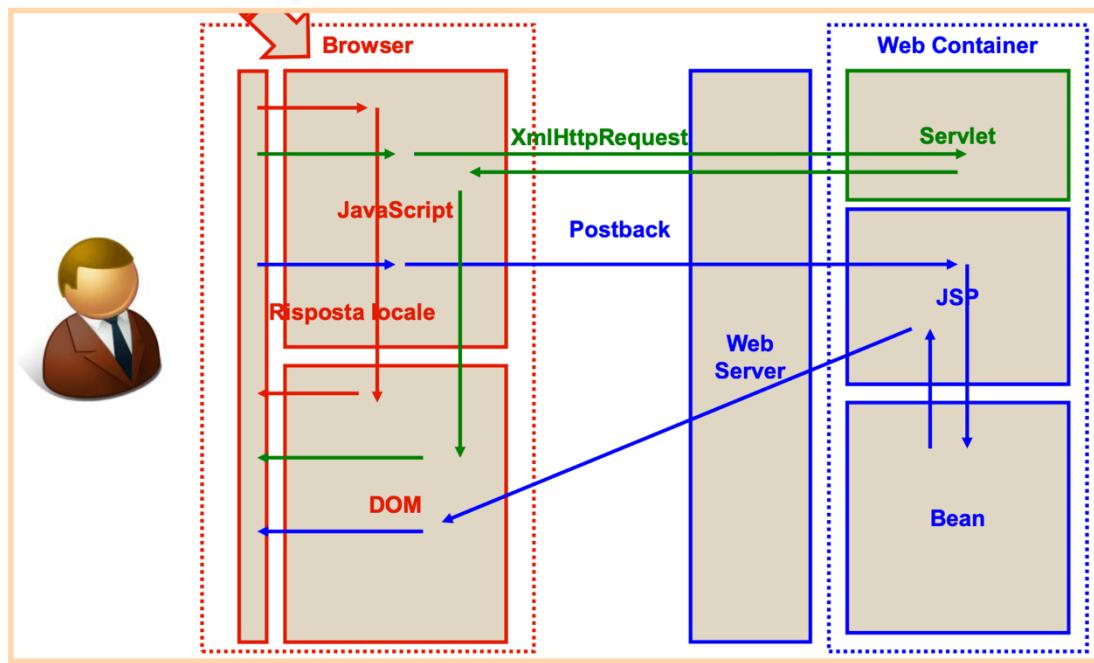
Consente di ottenere dati dal server senza necessità di ricaricare l'intera pagina realizzando una nuova comunicazione di tipo **asincrona**.

L'uso di request/response avviene in background in modo tale che il browser non si blocchi. [ATTENZIONE: questo implica controllare cosa succede durante questo continuo]



Sequenza Ajax:

- evento utente-paginaWeb
- esecuzione di funzione JavaScript
- si istanzia un oggetto XMLHttpRequest e si associa una funzione di "callback"
- chiamata asincrona al server che risponde al client
- il browser invoca una funzione di callback che elabora il risultato e aggiorna il DOM



10.1 XMLHttpRequest

Per istanziare un oggetto di questo tipo si usa la sintassi:

```
var xhr = new XMLHttpRequest();
```

La lista dei **metodi** disponibili è diversa da browser a browser... in genere si usano solo quelli presenti in Safari (sottoinsieme più limitato, ma comune a tutti i browser)

→ `open()`

Ha lo scopo di inizializzare la richiesta da formulare al server.

L'uso più comune prevede tre parametri: `open(method, uri, true)`

- **method**: stringa che assume il valore get o post
- **uri**: identifica la risorsa da ottenere
- **async** = `true` [se = false modalità: click, wait, refresh]

→ **setRequestHeader(nomeheader, valore)**

Consente di impostare gli header HTTP della richiesta.

Viene invocata per ogni header da impostare; notiamo che mentre per la GET sono opzionali, per la POST sono necessari.

- !Impostare setRequestHeader(connection, close)

→ **send(body)**

Consente di inviare la richiesta al server. Prende come parametro una stringa che costituisce il body della richiesta HTTP.

Esempi:

GET

```
var xhr = new XMLHttpRequest();
xhr.open("get", "pagina.html?p1=v1&p2=v2", true);
xhr.setRequestHeader("connection", "close");
xhr.send(null);
```

POST

```
var xhr = new XMLHttpRequest();
xhr.open("post", "pagina.html", true);
xhr.setRequestHeader("Content-type",
                     "application/x-www-form-urlencoded");
xhr.setRequestHeader("connection", "close"); //
xhr.send("p1=v1&p2=v2");
```

Stato e risultati della richiesta vengono memorizzati dall'interprete Javascript all'interno dell'oggetto. Le proprietà comunemente supportate sono...

→ **readyState**

Proprietà in sola lettura di tipo intero che consente di leggere lo stato della richiesta.

- **0**: uninitialized - l'oggetto esiste, ma non è stato ancora richiamato *open()*
- **1**: open - è stato invocato il metodo *open()*, ma *send()* non ha ancora effettuato l'invio dati
- **2**: sent - metodo *send()* è stato eseguito e ha effettuato la richiesta
- **3**: receiving – la risposta ha cominciato ad arrivare
- **4**: loaded - l'operazione è stata completata

[L'unico stato supportato da tutti i browser è il 4]

→ **onreadystatechange**

Dato che dopo una send l'esecuzione del codice non si blocca, per gestire la risposta si deve adottare un approccio a eventi. Occorre registrare una funzione di **callback** che viene richiamata in modo asincrono ad ogni cambio di stato della proprietà *ReadyState*.

```
xhr.onreadystatechange = nomefunzione
xhr.onreadystatechange = function(){istruzioni}
[assegnamento fatto prima di send()]
```

→ `status`

Contiene un valore intero corrispondente al codice HTTP dell'esito (200, 404, 500...).

→ `statusText`

Contiene una descrizione testuale del codice di stato

→ `responseText`

Stringa che contiene il body della risposta HTTP (disponibile solo se ReadyState = 4).

→ `responseXML`

Body della risposta convertito in documento XML (può essere *null*).

→ `getHttpRequestHeader(nomeheader) / getAllRequestHeader()`

Consentono di leggere gli header HTTP che descrivono la risposta del server.

Sono utilizzabili **solo nella funzione di callback**.

→ `abort()`

Consente l'interruzione delle operazioni di invio o ricezione. termina immediatamente la trasmissione dati. Notiamo che non ha senso invocarlo dentro la funzione di callback poiché se readyState non cambia, il metodo non viene richiamato quando la risposta si fa attendere.

Risulta tipico l'utilizzo di un time-out...

`setTimeOut(funzionePerAbortire, timeOut)`

Osservazione:

Applicazione asincrona... cosa succede a runtime?

Un utente può avere un approccio "strano"...

- *Difficile il debug*
- *Scrivere codice compatto*
- *Usare AJAX quando serve*

10.2 JSON

Spesso i dati scambiati fra client e server sono codificati in XML.

L'utilizzo di XML come formato di scambio fra client e server porta a generazione e utilizzo di quantità di byte piuttosto elevate e non ottimizzate.

Esiste un formato più efficiente per scambiare informazioni tramite AJAX? Si, JSON.

JSON (JavaScript Object Notation) è un formato di dati, NON UN LINGUAGGIO!

Si basa sulla notazione usata per le costanti oggetto e le costanti array in JavaScript.

----Notazioni JS----

[costante oggetto/oggetto]

```
var Beatles = { "Paese" :  
    "Inghilterra", "AnnoFormazione" :  
    1959,  
    "TipoMusica" : "Rock" }
```

```
var Beatles = new Object();  
Beatles.Paese = "England";  
Beatles.AnnoFormazione = 1959;  
Beatles.TipoMusica = "Rock";
```

[costante array/array]

```
var Membri =  
    ["Paul", "John", "George", "Ringo"];
```

```
var Membri = new Array  
    ("Paul", "John", "George", "Ringo");
```

----[si possono combinare array di oggetti o array dentro ad oggetti]----

Una **stringa JSON** è una stringa di caratteri “**nome**” : “**valore**”

```
'{"Paese" : "Inghilterra", "AnnoFormazione" : 1959, "TipoMusica" :  
    "Rock'n'Roll", "Membri" : ["Paul", "John", "George", "Ringo"] }'
```

Ho una stringa JSON, come la processo in un oggetto Java/JavaScript?

→ parser **eval()** [da non utilizzare]

```
var obj = eval ('(' + s + ')')
```

è un valutatore di espressioni, se esistesse e venisse passata una stringa contenente la funzione “deleteAll” eval forzerebbe l'espressione... cosa succede? MALE MALE MALE!!!!

→ parser **jabsorb** [funziona ma poco usato]

→ parser **JSON** [il più usato... è una libreria di Google da includere nel nostro progetto]

Inizializzazione: Gson g = new Gson();

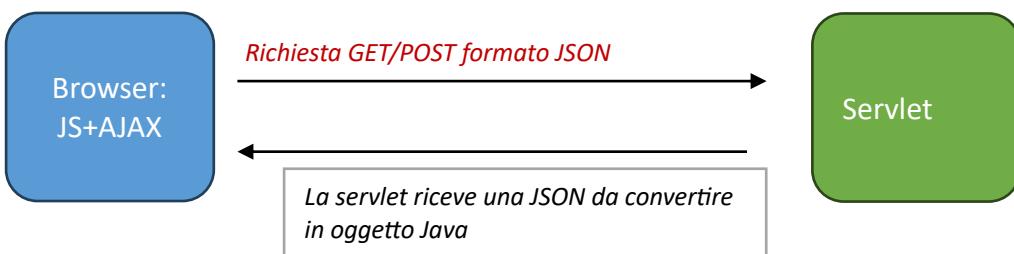
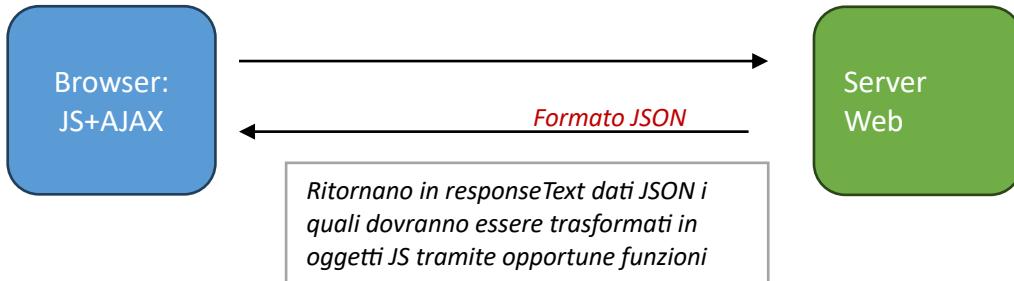
Da oggetto a JSON:

```
Person santa = new Person("Santa", "Claus");  
g.toJson(santa);
```

Da JSON a oggetto:

```
Person santa = g.fromJson(json, Person.class);
```

10.3 Schemi di interazione



AJAX

L'oggetto XMLHttpRequest effettua la richiesta di una risorsa via HTTP ad un server.

Non provoca né cambio di pagina, né sostituzione URI!

NOTA: L'evento onload è essenziale per le interazioni asincrone, finchè non si è certi di avere gli elementi del DOM caricati è inutile effettuare modifiche!

Molti browser supportano XMLHttpRequest come oggetto nativo, altri come oggetto ActiveX, altri ancora non lo supportano proprio...

Controllo del supporto:

```
// ad esempio invocata in corrispondenza dell'evento onload
function myAjaxApp() {
  const xhr = myGetXMLHttpRequest();
  if ( xhr ) { /* applicazione in versione AJAX */ }
  else { /* versione non AJAX o avviso all'utente */ } }

function myGetXMLHttpRequest() {
let xhr = false;
const activeXOptions = new Array( "Microsoft.XmlHttp",
"MSXML4.XmlHttp", "MSXML3.XmlHttp", "MSXML2.XmlHttp", "SXML.XmlHttp" );
);
// prima come oggetto nativo
try { xhr = new XMLHttpRequest(); }
catch (e) { }
// poi come oggetto activeX dal più al meno recente
if ( ! xhr ) {
let created = false;
for ( let i = 0 ; i < activeXOptions.length && !created ; i++ ) {
try { xhr = new ActiveXObject( activeXOptions[i] );
created = true; }
catch (e) { }
}
return xhr;
}
```

Funzione di callback:

L'esecuzione di codice non si blocca dopo una send()... la funzione di callback:

```
const xhr = myGetXMLHttpRequest();
xhr.open( "post", "sottoconto/pagina.html?", "post", true );
xhr.setRequestHeader( "content-type", "application/x-www-form-urlencoded" );
xhr.setRequestHeader( "connection", "close" );
```

```
xhr.send( "p1=v1&p2=v2" );
```

```
xhr.onreadystatechange = function() {/*callback*/}
```

va assegnata prima della send()

AJAX e callback, utils.js

La callback...

→ legge lo stato di avanzamento della richiesta

```
xhr.readyState (0, 1, 2, 3, 4) //4 = operazione completata
```

→ verifica la richiesta attraverso il codice di stato (200 successo, 404 page not found, ...)

```
xhr.status
```

```
// disponibile una descrizione testuale: alert(xhr.statusText)
```

→ ha accesso agli header di risposta (meglio se ha risposta conclusa)

→ può leggere il contenuto della risposta (se readyState=4)

```
xhr.responseText
```

```
xhr.responseXML
```

```
var textHolder = new Object();
textHolder.testo = "La risposta del server: ";

xhr.onreadystatechange = function(){
  if(xhr.readyState == 4 && xhr.status == 200){
    //...
    alert( textHolder.testo + xhr.responseText );
  }
};
```

Nota: se definite come funzione esterna può accettare parametri formali e riferirne i valori

```
function myPopup(oggettoAjax, contenitoreTesto){ ... }

//... var xhr = ...

// LA CALLBACK VA UTILIZZATA COSÌ:
xhr.onreadystatechange = function() { myPopup(xhr, textHolder); }
//E NON xhr.onreadystatechange = myPopup(xhr, textHolder);
```

utils.js: 05_TecWeb – Effettua opportuni controlli sulle funzioni

myGetElementById(*idElemento*) → restituisce un elemento html per id

ES: *myGetElementById('input').value, myGetElementById('div')*

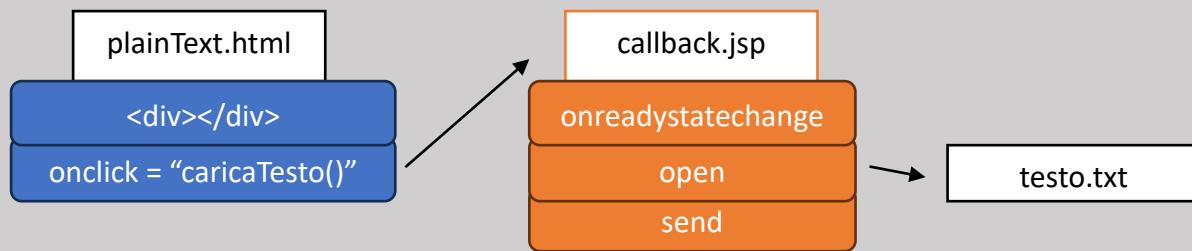
myGetXmlHttpRequest() → restituisce l'oggetto XMLHttpRequest effettuando gli opportuni controlli visti prima; ritorna *false* se il browser non supporta AJAX.

myGetRequestParamter(*parameterName*) → ritorna il valore del parametro richiesto per nome; ritorna *null* se non lo trova.

myGetChildByName(*theElement*, *name*) → recupera per nome l'elemento FIGLIO (*name*) di un element dato (*theElement*)

AJAX esempi: Scaricamento di un testo 05_TecWeb

Scaricamento di dati da un file di testo (testo.txt) in modo asincrono (tramite AJAX).



- Ricordiamo che la funzione di callback va definita prima della `send()`!!
- `open()` inizializza la richiesta: (metodo_get/post, uri_della_risorsa, true)
- `send(null)` invia la richiesta, il body è null in quanto ci interessa solo la risorsa testo.txt che verrà fornita tramite `responseText`

```
/*
 * Scarica un contenuto testuale dall'uri fornito
 * e lo aggiunge al contenuto dell'elemento e del dom
 * Gestisce sia AJAX che il mancato supporto ad AJAX
 */
function caricaTesto(uri,e) {

    // variabili di funzione
    var
        // assegnazione oggetto XMLHttpRequest
        xhr = myGetXmlHttpRequest();

    // try to place a breakpoint here and change xhr value to false!
    if ( xhr )
        caricaTestoAJAX(uri,e,xhr);
    else
        caricaTestoIframe(uri,e);

} // caricaTesto()
function caricaTestoAJAX(theUri, theElement, theXhr) {
    theXhr.onreadystatechange = function() {

        // designiamo la formattazione della zona dell'output
        theElement.class = "content";

        // verifica dello stato
        if ( theXhr.readyState === 2 ) {
            theElement.innerHTML = "Richiesta inviata...";
        } // if 2
        else if ( theXhr.readyState === 3 ) {
            theElement.innerHTML = "Ricezione della risposta...";
        } // else if 3
        else if ( theXhr.readyState === 4 ) {
            // verifica della risposta da parte del server
            if ( theXhr.status === 200 ) {
                // operazione avvenuta con successo
                theElement.innerHTML = theXhr.responseText;
            } // if 200
            else {
                // errore di caricamento
                theElement.innerHTML = "Impossibile effettuare l'operazione richiesta.<br />";
                theElement.innerHTML += "Errore riscontrato: " + theXhr.statusText;
            } // else (if !200)

        } // else if 4
        // -----
    } // callback function
    // impostazione richiesta asincrona in GET del file specificato
    try {
        theXhr.open("get", theUri, true);
    }
    catch(e) {
        // Exceptions are raised, for instance, when trying to access cross-domain URIs
        alert(e);
    }
    // invio richiesta
    theXhr.send(null);
} // caricaTestoAJAX()
```

(Guarda anche logica external)

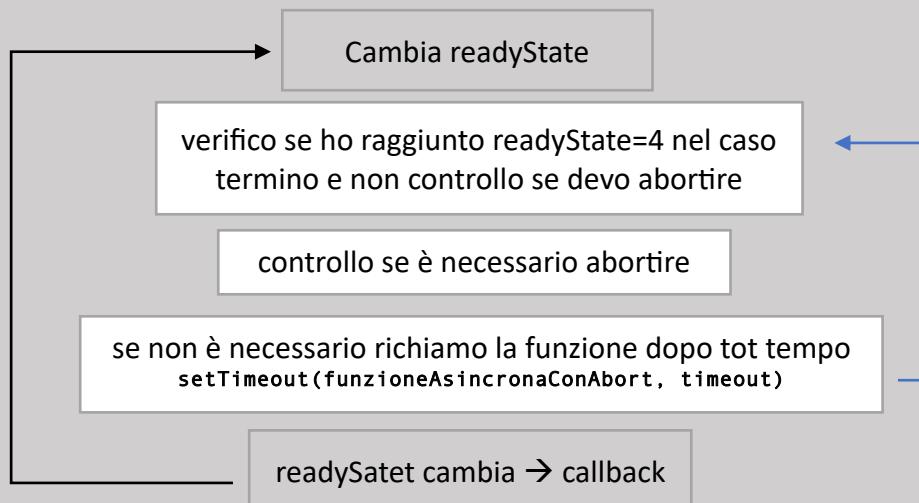
AJAX esempi: abortire 05_TecWeb loadabort.js

Uno smodato uso di AJAX genera un numero elevato di richieste verso il server, ecco perché è meglio usare connection=close.

Il metodo abort() non deve essere richiamato all'interno di una callback in quanto se readyState non cambia esso non viene mai invocato!

Si crea una funzione da far richiamare in modo asincorno:

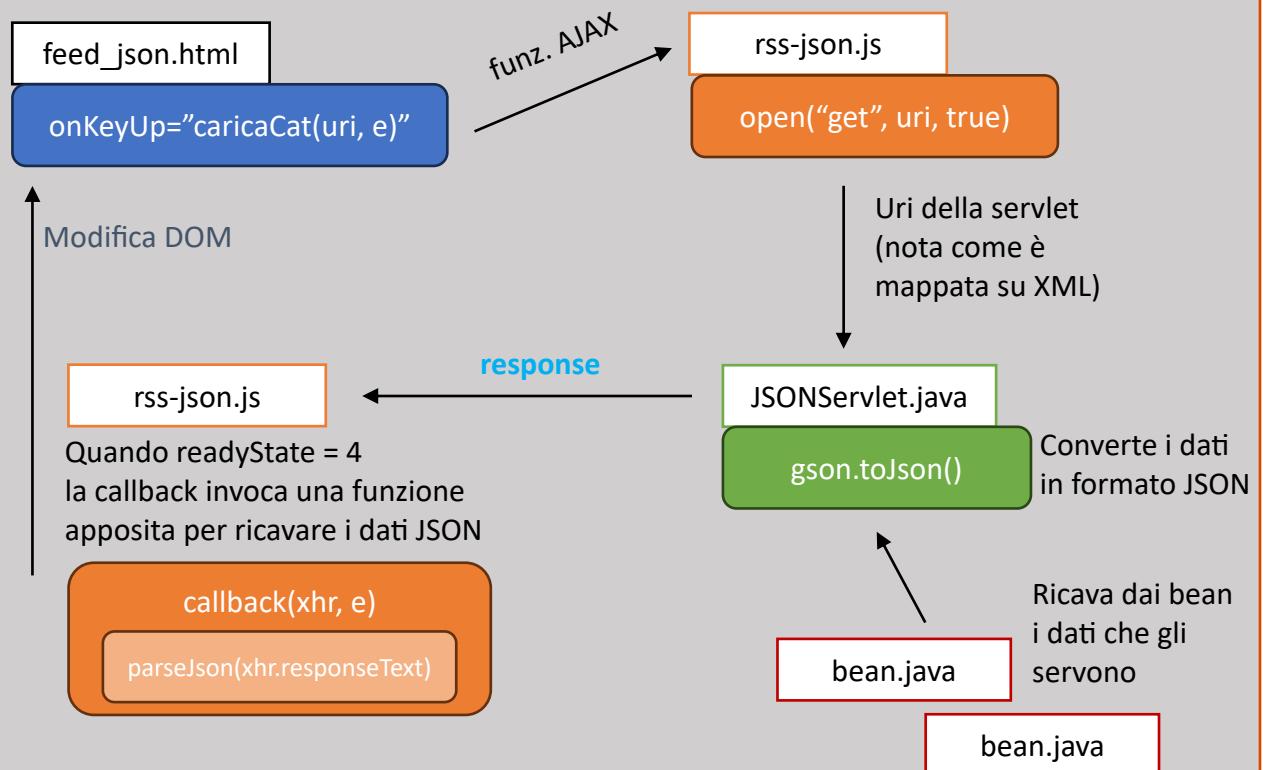
```
setTimeout(funzioneAsincronaConAbort, timeout_ms)
```



AJAX esempi: JSON 05_TecWeb

Ricorda di importare le librerie GSON.

Esempio con Servlet: Suggerimento di una categoria per ricerca.



Struttura ideale di una callback AJAX

```
function callback(theXhr, theElement){  
    if(theXhr.readyState == 2){/*...*/}  
    else if(theXhr.readyState == 3){/*...*/}  
    else if(theXhr.readyState == 4){  
        if(theXhr.status==200){  
            //operazione avvenuta con successo  
            /* if(theXhr.responseText && theXhr.responseText != ""){  
                theElement.value = theXhr.responseText;  
            }  
            */  
        }  
        else{/*errore di caricamento*/}  
    }  
}  
  
function caricaIframe(theUri, theHolder){  
    theHolder.innerHTML = '<iframe src="'+theUri+'></iframe>';  
}  
  
function caricaAJAX(uri, e, xhr){  
  
    xhr.onreadystatechange= function(){callback(xhr, e);}  
    try{ xhr.open("get", uri, true);}  
    catch(e){alert(e);}  
    xhr.send(null);  
  
}  
  
//FUNZIONE INVOCATA  
function carica(uri, e){  
    var xhr = myGetXmlHttpRequest();  
    if(xhr){  
        caricaAJAX(uri, e, xhr);  
    }  
    else{ caricaIframe(uri, e); }  
}
```

11 React.js

React.js è una libreria JS, ideata da Facebook, per la creazione di interfacce utente Web.
È una tecnologia **front-end** basata su linguaggio Javascript; esegue dunque all'interno del browser dell'utente.

Si ispira al concetto di “Single Page Application” (*SPA*) che funge da contenitore all'interno del quale la pagina web evolve dinamicamente attraverso l'utilizzo di **componenti** che interagiscono tra loro (*immaginiamoli come “pezzi di pagina” che interagiscono con le API della libreria React*).

React.js introduce il concetto di **Virtual DOM**, ovvero una copia esatta del DOM più leggera e facile da modificare. Risulta utile in caso di renderizzazione di porzioni di pagina (componenti).
[Sono inviate al DOM solo le modifiche strettamente necessarie.]

Ne notiamo subito i vantaggi:

- *Livello di astrazione: possibilità di lavorare sui singoli componenti*
- *Modifica del DOM fatta in modo trasparente da React*
- *Poco overhead, Virtual DOM alleggerisce il processo di rendering*

11.1 Concetti base

In HTML, nella sezione **<head>** è necessario importare le tre librerie Javascript di React...

- Per i metodi react
- Per il Virtual Dom
- Per ottenere il precompilatore JSX

```
<script src="https://unpkg.com/react@15/dist/react.js"></script>
<script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/
    babel-core/5.8.24/browser.js"></script>
```

Nella sezione **<body>** il codice è racchiuso tra tag di tipo **<script type="text/babel">**

Primo sguardo a elementi, metodi e classi componenti attraverso due esempi...

```
<body>
  <div id="root"></div>
  <script type="text/babel">
    const elem = <p>Hello</p>
    ReactDOM.render(elem,
      Document.getElementById('root'));
  </script>
</body>
```

Abbiamo definito un **React Element**, una costante che contiene un frammento di HTML. Questa istruzione è data in JSX.

Attraverso il metodo *render* abbiamo indicato **COSA** visualizzare e **DOVE**.

Notiamo che il paragrafo "Hello" è prodotto dinamicamente

```
<body>
  <div id="root"></div>
  <script type="text/babel">
    class HelloWorld extends React.Component{
      render(){ return <p>Hello</p> }
    };
    ReactDOM.render(<HelloWorld/>,
      Document.getElementById('root'));

  </script>
</body>
```

Abbiamo creato una classe componente estendendo **React.Component**. Questi possono accettare in input dati arbitrari e restituire *React Element* da far apparire sullo schermo.

Notiamo che è sempre necessario definire il metodo *render()*

11.2 JSX

Il linguaggio JSX nasce per semplificare un equivalente scrittura JS pura.

Permette allo sviluppatore di scrivere facilmente tag all'interno di codice JS.

Risulta dunque molto semplice...ad esempio... creare una lista...

```
const listElement = <ul className="list-of-items">
  <li className="item-1" key="key-1">Item 1</li>
  <li className="item-2" key="key-2">Item 2</li>
  <li className="item-3" key="key-3">Item 3</li>
</ul>;
```

o... quello che vogliamo usando i tag HTML e il metodo *ReactDOM.render(COSA, DOVE)*

All'interno di { } è possibile inserire qualsiasi tipo di espressione o funzione che restituisca un valore (*costanti, metodi, 2+2, ecc...*)

Esempio:

```
const nome = "Ale"
const element = <p>Hello {nome}</p>
```

Hello Ale

L'interprete di linguaggio Babel JSX è specificato tramite la sua libreria vista precedentemente.

11.3 Componenti

I **React Components** sono i “mattoncini” fondamentali che consentono di passare da una pagina statica a un’applicazione Web dinamica la cui interfaccia è in grado di rispondere agli eventi che si verificano nella pagina.



- La sezione più esterna, con bordo giallo, è il componente React che rappresenta l’applicazione e contiene tutti gli altri componenti
- A livello più basso, ogni riquadro di diverso colore è un componente React innestato.

I componenti sono pezzi di codice indipendenti e riusabili.

Esistono **due tipi di components**; entrambi restituiscono codice HTML tramite *return*:

function

```
function Car(){ return <h2>I'm a Car</h2> }
ReactDOM.render(<Car/>, document.getElementById('root'));
```

Il nome del componente deve cominciare con una lettera maiuscola!

La funzione deve restituire l’elemento da renderizzare tramite *return*!

ReactDOM.render() è l’istruzione che attiva la manipolazione del DOM

class

```
class Car extends React.Component{
  render(){ return <h2>I'm a Car</h2> }
}
ReactDOM.render(<Car/>, document.getElementById('root'));
```

Un tipo class deve estendere *React.Component*!

Deve implementare obbligatoriamente il metodo *render()* che restituisce tramite *return*!

La creazione può avvenire anche secondo logica factory:

```
var Car = React.createClass({
  render:function(){ return <h2>I'm a Car</h2> }
});
```

→ Concetto di **props** ←

Sia per le funzioni che per le classi è possibile specificare delle proprietà ed assegnare a queste determinati valori. Le props assumono valori immutabili per i quali non è prevista alcuna alterazione.

L'oggetto che contiene queste proprietà prende il nome di **props** (è una parola chiave riservata). In fase di rendering, è possibile accedere alle *props* di un componente richiamandole come fossero attributi di un tag HTML.

```
function Car(props){ return <h2>I'm a {props.colore} Car</h2> }
ReactDOM.render(<Car colore="red"/>,
                document.getElementById('root'));
```



```
class Car extends React.Component{
  render(){ return <h2>I'm {this.props.colore} a Car</h2> }
}
ReactDOM.render(<Car colore="red"/>,
                document.getElementById('root'));
```

→ Concetto di **state** ←

Esiste un altro modo per rappresentare le proprietà di un componente di tipo classe. Tutti i componenti di **tipo classe** possiedono un oggetto built-in che prende il nome di **state**. Queste proprietà non sono immutabili, quando cambiano viene invocata la ri-renderizzazione.

(NOTA: le componenti di tipo function sono stateless, ovvero non hanno un oggetto state)

Per il tipo di componente class è possibile definire un costruttore che viene invocato prima del rendering e funge da inizializzatore.

```
class Car extends React.Component{
  constructor(){
    super();
    this.state={color:"red"};
  }

  render(){ return <h2>I'm {this.state.color} a Car</h2> }
}
ReactDOM.render(<Car/>, document.getElementById('root'));
```

Se non si invoca il metodo *super()*, non è possibile usare la keyword *this* all'interno del costruttore.

È possibile imbattersi anche in questa sintassi:

```
constructor(props){  
  super(props);  
  this.state={...};  
}
```

L'oggetto state di un componente classe può essere modificato attraverso la funzione **setState()** ereditata da React.Component. L'invocazione provvederà a modificare lo stato e a re-invocare la funzione *render()*.

NOTA: l'oggetto state è encapsulato all'interno di un componente, il quale è l'unico ad avere diritto e responsabilità di modificarlo.

Esempio: *lancioDado.html* – SOLO codice componente

```
class Dado extends React.Component{  
  constructor(props){  
    super(props);  
    this.state = {numeroEstratto:0};  
  }  
  
  randomNumber(){  
    return Math.round(Math.random()*5)+1;  
  }  
  
  lanciaDado(){  
    this.setState({numeroEstratto:this.randomNumber()});  
  }  
  
  render(){  
    let valore;  
    if(this.state.numeroEstratto==0){  
      valore = <span>Lancia il dado cliccando sul pulsante</span>;  
    }  
    else{ valore = <span>{this.state.numeroEstratto}</span>;}  
  
    return(  
      <div className="card">  
        <p className="card_numb">{valore}</p>  
        <button className="card_but" onClick={()=>this.lanciaDado()}>  
          Lancia</button>  
      </div>  
    )  
  }  
}
```

Invocazione in modalità “arrow”

È consigliato costruire componenti senza stato (stateless). La tipica applicazione React è realizzata come una gerarchia di componenti: di solito, ci sono alcuni componenti ai vertici che saranno responsabili di mantenere lo stato della applicazione e di passare le informazioni giù ai componenti figli tramite *props*.

11.4 Gestione degli eventi

Sintassi:

Html/javascript	JSX
<pre><button onclick="attivaLaser()"> Attiva Laser</button></pre>	<pre><button onClick={attivaLaser}> Attiva Laser</button></pre>

Analizziamo il seguente esempio per capire come funziona la gestione di eventi...

```
class App extends React.Component{  
  constructor(props){  
    super(props);  
  }  
  handleClick(e){  
    console.log("Pulsante premuto - Evento ${e.type}");  
  }  
  render(){  
    return(<button onClick={this.handleClick}>Pulsante</button>)  
  }  
}
```

Stampato nella console dello strumento di ispezione browser

Il parametro *e* è un evento sintetico, questi sono definiti in base alle specifiche W3C (standard).

Per una corretta invocazione dell'handler dell'evento, occorre che l'oggetto invocante sia il componente e non il documento. A tal fine, faremo ricorso alla keyword *this*....

Due possibilità:

Forzare la bind	Modalità arrow
<pre>constructor(props){ super(props); this.handleClick = this.handleClick.bind(this); }</pre>	<pre><button onClick={()=>this.handleClick()}> Pulsante </button></pre>

11.5 Form

In React, gli elementi HTML di cui è composto un form funzionano in un modo leggermente differente. Gli elementi di un form quali `<input>` `<select>` `<textarea>`, mantengono e aggiornano il proprio stato in base all'input dell'utente... in React, come è già noto, lo stato mutabile viene mantenuto nella proprietà *state* del componente e viene poi aggiornato solo mediante *setState()*.

A ciascun evento deve essere associato uno specifico handler:

<u>evento onChange</u>	<u>evento onSubmit</u>
<pre><input type="text" onChange={this.handleChange}></pre>	<pre><form onSubmit={this.handleSubmit}></pre>

L'invocazione di una risorsa tramite request HTTP è possibile tramite l'uso delle Fetch API fornite da javascript. Mettono a disposizione metodi per recuperare risorse in modo asincrono e per manipolare parti della pipeline HTTP. [VEDI SLIDE]

React.js

Innanzitutto, è necessaria l'esistenza di un ambiente Node.js

- 1- Scaricare ed installare Node.js
- 2- Da terminale creare una directory di lavoro e lanciare il comando:
npx create-react-app my-app
- 3- La directory appena creata (my-app) è l'area di lavoro di un Progetto React
- 4- Rimuovere (o modificare) i file di esempio (App.js) per iniziare un nuovo progetto
- 5- Per lanciare l'applicazione React eseguire:
npm start
all'interno della root del progetto

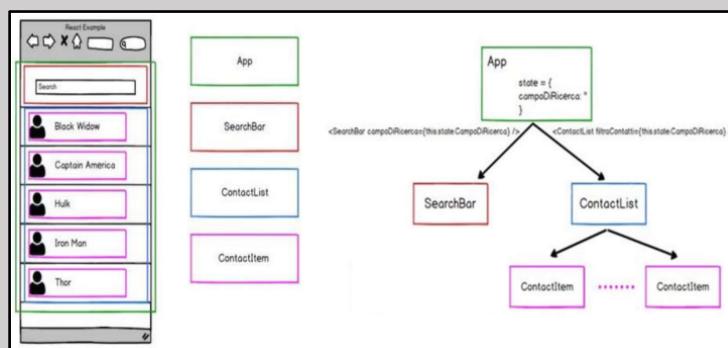
→ per ogni progetto è necessario creare una opportuna my-app (cambiare il nome)

→ tomcat: localhost:3000

Spostarsi da eclipse a VisualStudio per lavorare meglio!!

Una tipica applicazione React è formata da una gerarchia di componenti.

Ci sono alcuni componenti ai vertici che sono responsabili di mantenere lo stato e passare le informazioni ai figli tramite props.



Dato un evento, se l'handler deve fare accesso allo state del componente occorre o forzare la bind all'interno del costruttore o invocare l'handler in modalità arrow.

React.js: calcolatrice my-app06a

Quando si progetta è importante disgregare l'applicazione in modo tale che i singoli componenti abbiano una propria identità. Creiamo dunque i componenti...

Keyboard.js popolato da bottoni, ovvero tasti della calcolatrice

Display.js visualizzerà espressione e risultato

App.js componente contenitore che ISTANZA gli altri componenti e GESTISCE gli eventi.

Keyboard e Display non hanno state ma solo props inizializzati da App (che ha state).

Il file .html all'interno dei quali vengono richiamati gli script dei componenti non funziona qualora si provi ad accedervi direttamente da file system.

Usare npm start o fare il deploy su Tomcat copiando la cartella dentro la dir webapps.

html: il div racchiuderà le nostre componenti react!

```
<!DOCTYPE html>

<html>
  <head>
    <meta charset="UTF-8" />
    <title>Calcolatrice con due display</title>
    <link rel="StyleSheet" href=".//styles/style.css" type="text/css" media="all" />
    <script src=".//libs/react.js"></script>
    <script src=".//libs/react_dom.js"></script>
    <script src=".//libs/babel.js"></script>

  </head>

  <body>
    <!-- Placeholder -->
    <div id="root"></div>

    <!-- React components -->
    <script src=".//react-components/Display.js" type="text/babel"></script>
    <script src=".//react-components/KeyBoard.js" type="text/babel"></script>
    <script src=".//react-components/App.js" type="text/babel"></script>
    <!-- <script src="index.js" type="text/babel"></script>-->

    <script type="text/babel">
      ReactDOM.render(
        <React.StrictMode>
          <App/>
        </React.StrictMode>,
        document.getElementById('root')
      );
    </script>
  </body>
</html>
```

IMPORTANTE!

Le classi estendono React.Component necessario:

```
import React from 'react';
per usare altri componenti
import KeyBoard from './KeyBoard';
esportare i componenti come ultima riga di codice
export default App;
```

Display.js

```
'use strict';

import React from 'react';

class Display extends React.Component {

  render() {
    let result = this.props.result;
    return (
      <div className="result">
        <p>{result}</p>
      </div>
    );
  }
}

export default Display;
```

NOTA: la props è un parametro inizializzato da App.js

KeyBoard.js

```
'use strict';

import React from 'react';

class KeyBoard extends React.Component {

  render() {
    return (
      <div className="keypad">
        <button name="(" onClick={this.props.onClick}>(</button>
        <button name="CE" onClick={this.props.onClick}>CE</button>
        <button name=")" onClick={this.props.onClick}>)</button>
        <button name="C" onClick={this.props.onClick}>C</button><br/>

        <button name="1" onClick={this.props.onClick}>1</button>
        <button name="2" onClick={this.props.onClick}>2</button>
```

NOTA: la props "onClick" passata da App.js è in realtà una funzione su App.js

App.js

```
1  'use strict';
2
3  import React from 'react';
4  import Display from './Display';
5  import KeyBoard from './KeyBoard';
6
7  class App extends React.Component {
8    constructor(){
9      super();
10     this.state = {
11       result: '',
12       input: ''
13     }
14     this.onClick = this.onClick.bind(this);
15   }
16
17   onClick(e) {
18     let button = e.target.name
19     if(button === "="){
20       this.calculate()
21     }
22
23     else if(button === "C"){
24       this.reset()
25     }
26     else if(button === "CE"){
27       this.backspace()
28     }
29     else {
30       this.setState({
31         input: this.state.input + button
32       })
33     }
34   };
35   calculate(){
36     try {
37       this.setState({
38         result: eval(this.state.input) || "" ) + ""
39     })
40     } catch (e) {
41       this.setState({
42         result: "error"
43     })
44     }
45   };
46
47   reset(){
48     this.setState({
49       result: "",
50       input: ""
51     });
52
53   backspace(){
54     this.setState({
55       input: this.state.input.slice(0, -1)
56     });
57   };
58
59   render() {
60     return (
61       <div className="calculator-body">
62         <h1>Calcolatrice</h1>
63         <Display result={this.state.input}/>
64         <Display result={this.state.result}/>
65         <KeyBoard onClick={this.onClick}/>
66       </div>
67     );
68   }
69 }
70
71 export default App;
```

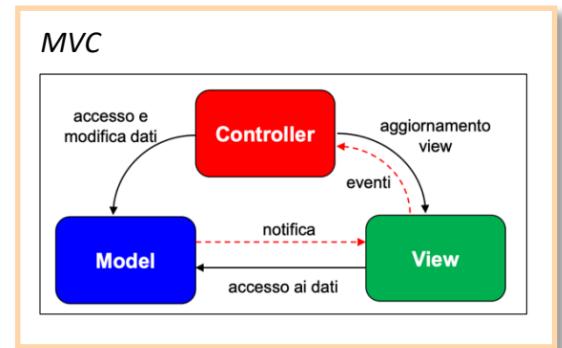
12 Tecnologie Server-side

Con “**Model 2**” indichiamo un progetto di applicazione Web in Java che separa nettamente la logica di business da quella di presentazione. Spesso, anche se in modo inappropriato, Model 2 e MVC (Model-View-Controller) sono usati come sinonimi; questo accade perché il modello MVC si adatta per architettura ad applicazioni Web interattive.

Model: rappresenta livello dei dati, incluse operazioni per accesso e modifica. *Model* deve notificare *view* associate quando il modello viene modificato.

View: si occupa di rendering dei contenuti di model e gira input utente verso controller.

Controller: definisce comportamento dell'applicazione:
→ fa dispatching di richieste utente e seleziona view per presentazione.
→ interpreta input utente e lo mappa su azioni che devono essere eseguite da model.



Possible mapping Web...

Controller → deve ottenere il contenuto da mostrare

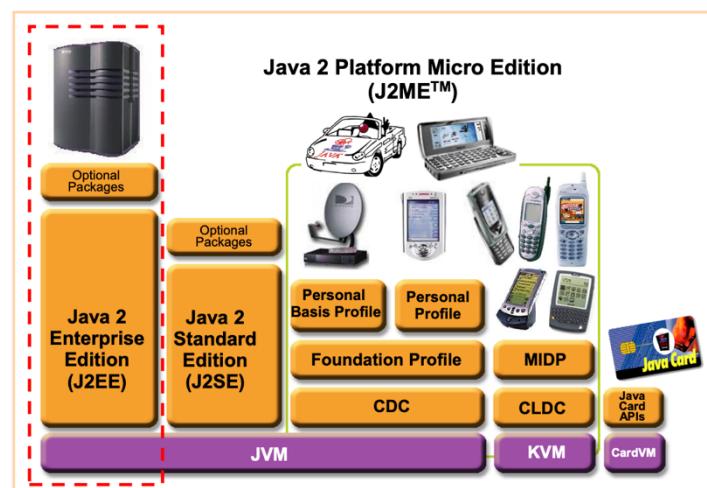
→ richieste del browser cliente passate ad un controller (Servlet / EJB SB)

View → renderizza il contenuto (JSP)

12.1 J2EE

Java Enterprise Edition è una piattaforma open e standard per lo sviluppo, il deployment e la gestione di applicazioni enterprise n-tier, Web-enabled, server-centric e basate su componenti.

Analizziamo la logica per capire alcune direzioni di implementazione di Java Model 2, di che cosa si occupa il container J2EE, e le differenze rispetto ad approcci a container leggero come Spring. Il container può fornire “automaticamente” molte delle funzioni per supportare il servizio applicativo verso l’utente...
→ Supporto al ciclo di vita
→ Supporto al sistema di nomi
→ Supporto al servizio
→ Sicurezza
→ ...

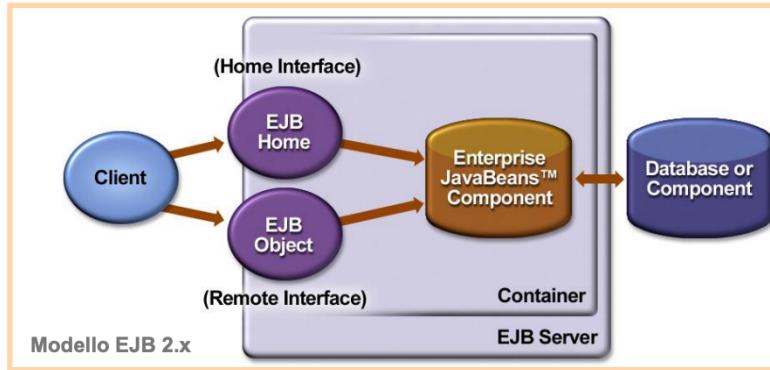


12.2 EJB

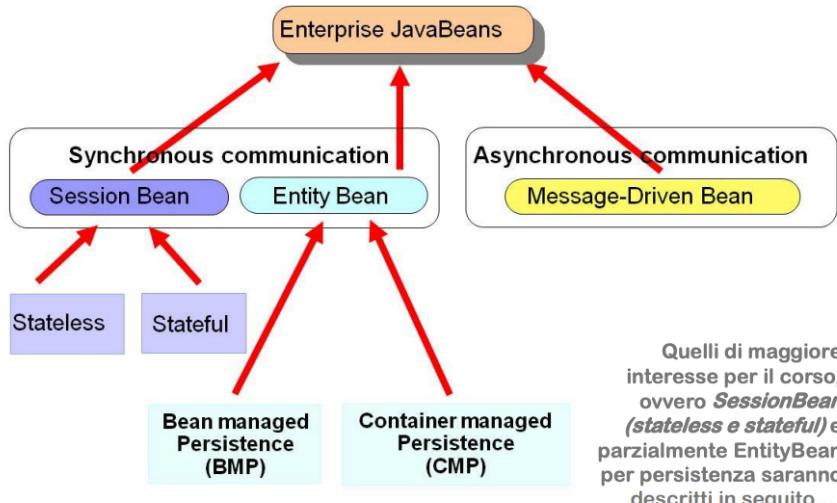
EJB è una tecnologia per componenti server-side. Separa logica di business e codice di sistema proponendo un modello a container pesante per la fornitura dei servizi.

I principi di design impongono che applicazioni EJB:

- Hanno componenti debolmente accoppiati
- NON si occupano della gestione di risorse
- Sono N-tier (suddivisa su più livelli, ognuno con una sua funzione)



Il cliente può interagire remotamente con un componente EJB tramite interfacce ben definite e passando sempre dal container pesante, attivo all'interno di un EJB Server.



I principali componenti EJB sono:

- Session Bean (Stateful / Stateless)
- Entity Bean (BMP / CMP)
- Message-Driven Bean

12.2.1 EJB componenti

> Session Bean <

Ci concentriamo sulla realizzazione di un Session Bean per la realizzazione di un controller. Questi lavorano tipicamente per un singolo cliente, NON sono persistenti, NON rappresentano dati di un DB (anche se possono accedere/modificare dati).

In EJB2.x la classe Bean deve implementare l'interfaccia javax.ejb.SessionBean.

Quando usare i Session Bean?

- Per modellare oggetti di processo o di controllo specifici per un cliente
- Per modellare workflow o attività di gestione e per coordinare interazioni
- Per muovere la logica applicativa di business dal lato cliente al servitore

Se...

Stateless: esegue una richiesta e restituisce risultati senza salvare alcuna informazione di stato relativa al cliente.

Stateful: può mantenere uno stato specifico per un cliente.

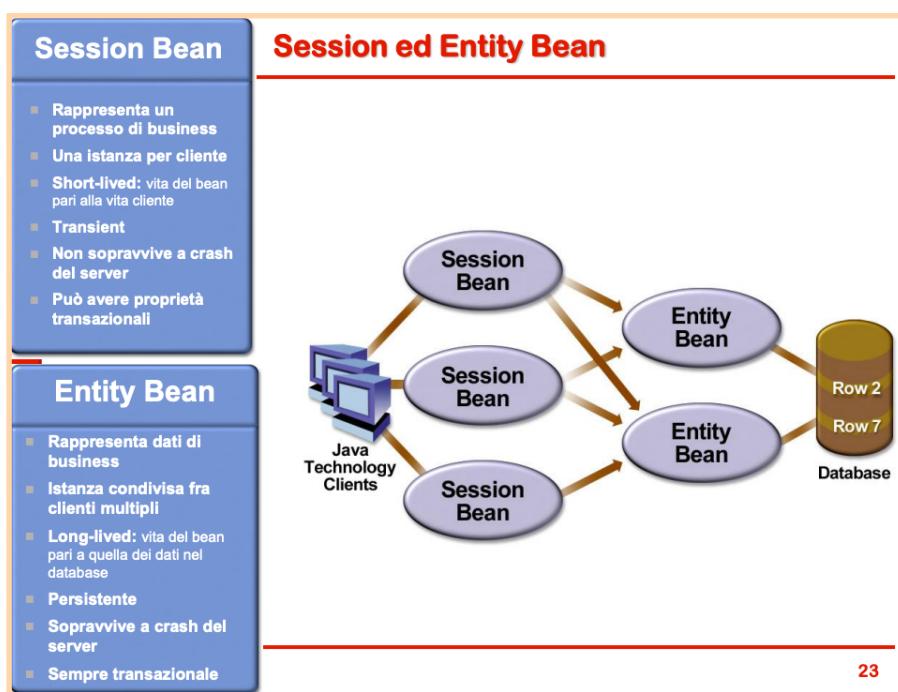
> Entity Bean <

Forniscono una vista ad oggetti dei dati mantenuti in un database.

I componenti rimangono nel sistema fino a quando i dati esistono nel DB.

L'accesso è condiviso per clienti differenti.

In EJB2.x la classe Bean deve implementare l'interfaccia javax.ejb.EntityBean.



23

> Message-Driven Bean <

Sono i consumatori dei messaggi asincroni.

Non possono essere invocati direttamente dai clienti, sono attivati in seguito all'arrivo di un messaggio. I messaggi sono inviati dal Cliente verso *code* o *i topic* per i quali questi componenti sono in ascolto.

M-DB deve implementare `javax.jms.MessageListener` e il metodo `onMessage()`.

Osservazione: EJB 3.0, i bean di tipo sessione e message-driven sono classi Java ordinarie (Plain Old Java Object - POJO)

[esempi di codice sulle slide]

12.2.2 Servizi container-based

È rilevante capire quali servizi e come vengono supportati in un modello a container pesante.

Gestione concorrenza

→ **Resource Pooling:** pooling dei componenti, evitare di mantenere una istanza separata di ogni EJB per ogni cliente (**stateless session bean**). Ogni EJB container mantiene un insieme di istanze del bean pronte per servire richieste cliente. Ogni metodo è indipendente.

→ **Activation:** usata per **stateful session bean**. Gestisce la coppia EJB + istanza bean stateful.

- **Passivation:** dissociazione fra stateful bean instance e suo oggetto EJB, con salvataggio dell'istanza su memoria (serializzazione).
- **Activation:** recupero dalla memoria dello stato dell'istanza e riassociazione con oggetto EJB

Per definizione, *session bean* non possono essere concorrenti, nel senso che una singola istanza è associata ad un singolo cliente. Vietato l'utilizzo di thread a livello applicativo e, ad esempio, della keyword `synchronized`.

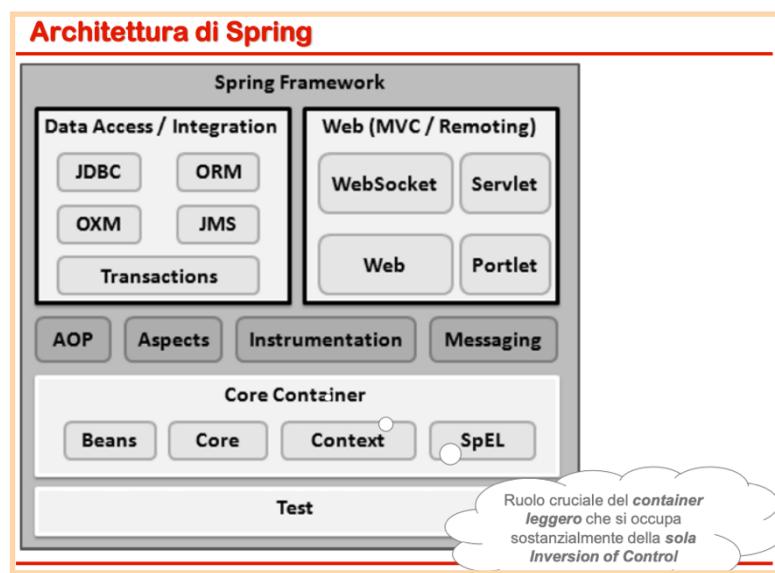
12.3 Spring

Spring rappresenta l'implementazione di modello a container leggero per la costruzione di applicazioni Java SE e Java EE.

Le funzionalità chiave sono:

- **Inversion of Control e Dependency injection**
Eliminazione della necessità di fare binding “manuale” fra componenti
- **Persistenza**
Livello di astrazione generico per la gestione delle transazioni con DB
- **Integrazione Web tier**
Framework MVC per applicazioni Web, costruito sulle funzionalità base Spring
- **Supporto ad Aspect Oriented Programming**
Framework di supporto a servizi di sistema, come gestione delle transazioni, tramite tecniche AOP. Miglioramento soprattutto in termini di modularità e parzialmente correlata anche la facilità di testing

Spring si presenta come framework modulare. Ha un'architettura a layer, con la possibilità di utilizzare anche solo alcune parti isolate.



Nell'architettura Spring è importante il ruolo del **Core Package**.

Consiste in un container leggero che si occupa di *Inversion of Control (Dependency Injection)*. L'elemento fondamentale è **BeanFactory**, che fornisce una implementazione estesa del pattern factory ed elimina la necessità di gestione di singleton (classi con una solo istanza) a livello di programmazione, permettendo di disaccoppiare configurazione e dipendenze dalla logica applicativa.

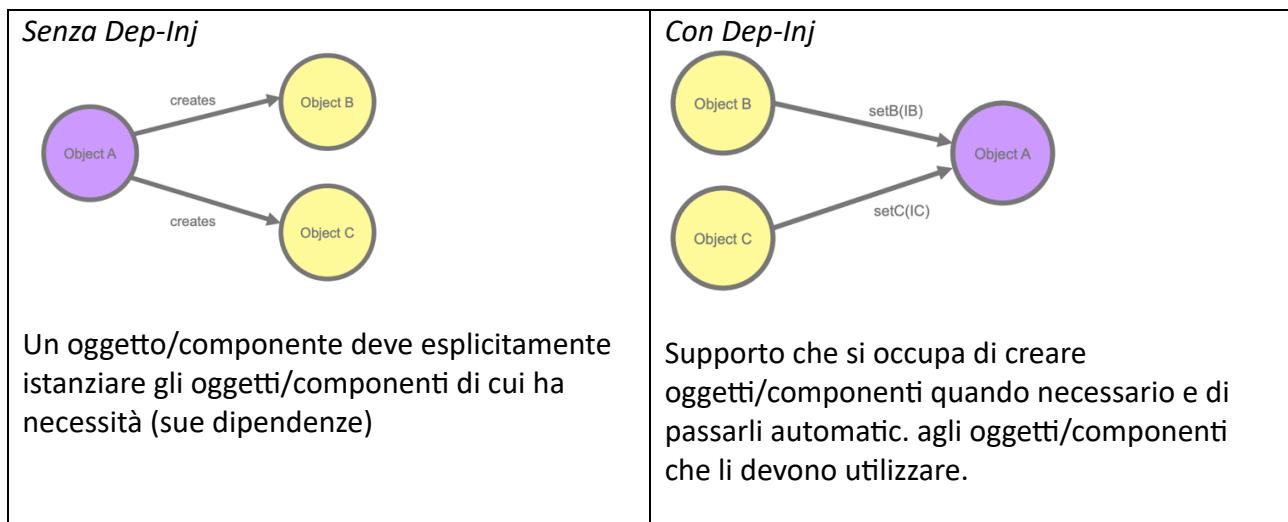
Altro elemento fondamentale è **MVC package** con ampio supporto a progettazione e sviluppo di architetture Model-View-Controller.

12.3.1 Dependency Injection

La Dependency Injection è l'applicazione più nota del principio di Inversion of Control. Il Container leggero si occupa di risolvere le dipendenze dei componenti attraverso l'opportuna configurazione dell'implementazione dell'oggetto.

Opposta ai pattern più classici di istanziazione di componenti o Service Locator, dove è il componente che deve determinare l'implementazione della risorsa desiderata.

I benefici sono vari tra cui: flessibilità, possibilità e facilità di testing e manutenibilità.



Idea base per implementazione:

→ **a livello di costruttore**

Dipendenze fornite attraverso i costruttori dei componenti.

Costruttori in oggetto A che accettano B e C come parametri di ingresso

```
public class ConstructorInjection {  
    private Dependency dep;  
    public ConstructorInjection(Dependency dep) {  
        this.dep = dep;    } }
```

→ **a livello di metodi setter**

Dipendenze fornite attraverso i metodi di configurazione.

A contiene metodi setter che accettano interfacce B e C come parametri di ingresso.

```
public class SetterInjection {  
    private Dependency dep;  
    public void setMyDependency(Dependency dep) {  
        this.dep = dep;    } }
```

12.3.2 BeanFactory

L'oggetto **BeanFactory** è responsabile della gestione dei bean che usano Spring e delle loro dipendenze.

L'Oggetto viene creato dall'applicazione nella forma: **XmLBeanFactory**.

Una volta creato, l'oggetto BeanFactory legge un file di configurazione e si occupa di fare l'injection.

```
public class XmlConfigWithBeanFactory {  
  
    public static void main(String[] args) {  
        XmlBeanFactory factory = new XmlBeanFactory(new  
            FileSystemResource("beans.xml"));  
        SomeBeanInterface b = (SomeBeanInterface) factory.  
            getBean("nameOftheBean"); } }
```

Rispetto ad un container pesante dove è possibile accedere ai componenti solamente attraverso l'interfaccia di un container. **In un container leggero** una volta ottenuto il riferimento al componente dal container, posso accedervi direttamente alle successive interazioni.



12.3.3 Injection e bean

Spring supporta diversi tipi di parametri con cui fare injection:

1. Valori semplici
2. Bean all'interno della stessa factory
3. Bean anche in diverse factory
4. Collezioni (collection)
5. Proprietà definite esternamente

Tutti questi tipi possono essere usati sia per injection sui costruttori che sui metodi setter.

```
Ad esempio, injection di valori semplici  
<beans>  
    <bean id="injectSimple" class="InjectSimple">  
        <property name="name"> <value>John Smith</value></property>  
        <property name="age"> <value>35</value> </property>  
        <property name="height"> <value>1.78</value> </property>  
    </bean>  
</beans>
```

L'injection di Bean della stessa Factory è usata quando è necessario fare injection di un bean all'interno di un altro (target bean). Si usa il tag `<ref>` in `<property>` o `<constructor-arg>` del target bean.

```
<beans>
    <bean id="injectRef" class="InjectRef">
        <property name="oracle">
            <ref local="oracle"/>
        </property>
    </bean>
</beans>
```

Controllo lasco sul tipo del bean “iniettato” rispetto a quanto definito nel target.

- Se il tipo definito nel target è un'interfaccia,
il bean injected deve essere un'implementazione di tale interfaccia.
- Se il tipo definito nel target è una classe,
il bean injected deve essere della stessa classe o di una sottoclasse.

Come fa BeanFactory a trovare il bean richiesto?

Ogni bean deve avere un nome unico all'interno della BeanFactory contenente!

Procedura di risoluzione dei nomi:

- Se un tag ha un attributo di nome **id**, il valore di questo attributo viene usato come nome
- Se non c'è attributo id, Spring cerca un attributo di nome **name**
- Se non è definito né id né name, Spring usa il **nome della classe** del bean come suo nome.

12.3.4 Spring MVC

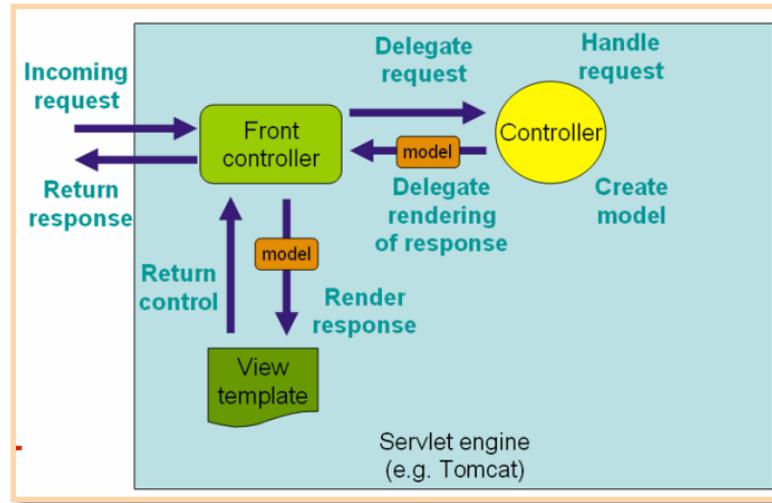
Logica:

Supporto a componenti “controller”, responsabili per interpretare richiesta utente e interagire con business object applicazione. Una volta che il controller ha ottenuto i risultati (parte “model”), decide a quale “view” fare forwarding del model; view utilizza i dati in model per creare una presentazione verso l'utente.

Caratteristiche:

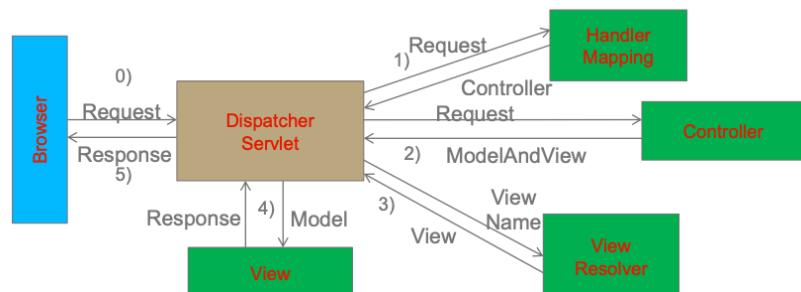
- Chiara separazione ruoli: controller, validator, oggetti command/form/ model, DispatcherServlet, handler mapping, view resolver, ...
- Adattabilità e riutilizzabilità: possibilità di utilizzare qualsiasi classe per controller purché implementi interfacciapredefinita
- Flessibilità nel trasferimento model: via un Map nome/valore, quindi possibilità di integrazione con svariate tecnologie per view
- Facilità di configurazione: grazie al meccanismo standard di dependency injection di Spring
- Handler mapping e view resolution configurabili
- Potente libreria di JSP tag (Spring tag library): supporto a varie possibilità di temi (theme)
- Componenti con ciclo di vita scoped e associati automaticam a HttpServletRequest o HttpSession (grazie a WebApplicationContext di Spring)
- Progettato attorno a una servlet centrale**
che fa da dispatcher delle richieste (**DispatcherServlet**)

- ❑ Completamente integrata con IoC container di Spring
- ❑ **DispatcherServlet** come “Front Controller”
- ❑ **DispatcherServlet** è una normale servlet e quindi serve URL mapping tramite web.xml



Il **DispatcherServlet**...

- Intercetta le HTTP Request in ingresso che giungono al Web container
- Cerca un controller che sappia gestire la richiesta
- Invoca il controller ricevendo un model (output business logic) e una view (come visualizzare output)
- Cerca un View Resolver opportuno tramite cui scegliere View e creare HTTP Response



Azioni svolte da **DispatcherServlet**:

- ❑ *WebApplicationContext* è associato alla richiesta (controller e altri componenti potranno utilizzarlo) `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE`
- ❑ “*locale*” resolver associato alla richiesta (può servire nel processamento richiesta e view rendering)
- ❑ *theme resolver* associato alla richiesta (view potranno determinare quale theme usare)
- ❑ Viene cercato un handler appropriato. Se trovato, viene configurata una catena di esecuzione associata all’handler (preprocessori, postprocessori e controller); risultato è preparazione di model
- ❑ Se restituito un model, viene girato alla view associata (perché un model potrebbe non essere ricevuto?)

>Spring Controller< -----

Spring supporta la nozione di controller in modo astratto permettendo creazione di ampia varietà di controller: form-specific controller, command-based controller, controller che eseguono come wizard, ...

È basato sull'interfaccia:

```
org.springframework.web.servlet.mvc.Controller  
public interface Controller {  
    ModelAndView handleRequest( HttpServletRequest request,  
        HttpServletResponse response) throws Exception; }
```

Ma molte implementazioni sono già disponibili, come:

- *AbstractController*

classe che offre già supporto per caching. Quando la si utilizza come baseclass, necessita di fare overriding del metodo

handleRequestInternal(HttpServletRequest, HttpServletResponse)

```
public class SampleController extends AbstractController {  
    public ModelAndView handleRequestInternal( HttpServletRequest  
        request, HttpServletResponse response) throws Exception {  
        ModelAndView mav = new ModelAndView("hello");  
        mav.addObject("message", "Hello World!");  
        return mav; } }
```

O altri come ParameterizableViewController, UrlFilenameViewController...

Notiamo i **Command controller** che permettono di associare dinamicamente parametri di HttpServletRequest verso oggetti dati specificati.

Alcuni controller disponibili:

- *AbstractCommandController* nessuna funzionalità form, solo consente di specificare che fare con oggetto command (JavaBean) popolato automaticamente coi parametri richiesta.
- *AbstractFormController* offre supporto per form; dopo che utente ha compilato form, mette i campi in oggetto command. Il programmatore deve specificare metodi per determinare quali view utilizzare per presentazione form.
- ...

>Spring Handler< -----

Funzionalità base: fornitura di HandlerExecutionChain, che contiene un handler per la richiesta e può contenere una lista di handler interceptor da applicare alla richiesta, prima o dopo esecuzione handler. All'arrivo di una richiesta, DispatcherServlet la gira a handler per ottenere un HandlerExecutionChain appropriato; poi DispatcherServlet esegue i vari passi specificati nella chain.

Concetto potente e molto generale: pensate a handler custom che determina una specifica catena non solo sulla base dell'URL della richiesta ma anche dello stato di sessione associata.

Diverse possibilità per handler mapping in Spring.

Maggior parte estendono **AbstractHandlerMapping** e condividono le proprietà seguenti:

interceptors: lista degli intercettori

defaultHandler: default da utilizzare quando no matching specifico possibile

...

>Spring View< -----

Spring mette a disposizione anche componenti detti “view resolver” per semplificare rendering di un model su browser, senza legarsi a una specifica tecnologia per view.

Due interfacce fondamentali sono:

- ❑ **ViewResolver** per effettuare mapping fra nomi view e reale implementazione di view
 - **AbstractCachingViewResolver** realizza trasparentemente caching di view per ridurre tempi preparazione
 - **XmlViewResolver** ...
- ❑ **View** per preparazione richieste e gestione richiesta verso una tecnologia di view

>Spring tag library< -----

Ampio set di tag specifici per Spring per gestire elementi di form quando si utilizzano JSP e MVC

- Accesso a oggetto command
- Accesso a dati su cui lavora controller

Libreria di tag contenuta in spring.jar, descrittore chiamato spring-form.tld, per utilizzarla:

```
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>
dove form è prefisso che si vuole utilizzare per indicare tag nella libreria
```

dove form è prefisso che si vuole utilizzare per indicare tag nella libreria.

>Spring e gestione di eccezioni< -----

HandlerExceptionResolver per semplificare gestione di eccezioni inattese durante esecuzione controller; le eccezioni contengono info su handler che stava eseguendo al momento dell’eccezione.

- Implementazione di interfaccia **HandlerExceptionResolver** (metodo resolveException(Exception, Handler) e restituzione di oggetto ModelAndView)
- Possibilità di uso di classe **SimpleMappingExceptionResolver**, con mapping automatico fra nome classe eccezione e nome view

Osservazione:

In molti casi è sufficiente usare convenzioni pre-stabilite e ragionevoli default per fare mapping senza bisogno di configurazione Convention-over-configuration Riduzione quantità di configurazioni necessarie per configurare handler mapping, view resolver, istanze ModelAndView,...

[Vedi slide]

13 Node.js

Node.js è una tecnologia server-side per il backend basato sul motore V8 di chrome; di notevole importanza in quanto...

- **Non utilizzo di thread/processi** dedicati
- **Maggiore scalabilità**
- Supporto efficiente a **I/O asincrono non bloccante**
- **Modello ad eventi**

Al posto dei thread usa “**event loop**” con stack; questo riduce fortemente overhead di context switching. In particolare, usa il framework CommonJS, risulta più simile ad un linguaggio di programmazione.

Thread vs Event-driven

Thread	Asynchronous Event-driven
Blocca applicazione/richieste con listener-worker thread	Un solo thread, che fa ripetutamente fetching di eventi da una coda
Usa modello incoming-request	Usa una coda di eventi e processa eventi presenti
Multithreaded server potrebbe bloccare una richiesta che coinvolge eventi multipli	Salva stato e passa poi a processare il prossimo evento in coda
Usa context switching	No contention e NO context switch
Usa ambienti multithreading in cui listener e worker thread spesso acquisiscono incoming-request lock	Usa framework con meccanismi per cosiddetto I/O asincrono (callback, NO poll/select, O_NONBLOCK)

```
request = readRequest(socket);
reply = processRequest(request);
sendReply(socket, reply);
```

Implementation:

Thread switching (i.e. blocking) and a scheduler

```
startRequest(socket);
listen("requestAvail", processRequest);
listen("processDone", sendReplyToSock);
```

Implementation:

Event queue processing

Usando callback:

```
request = readRequest(socket);
reply = processRequest(request);
sendReply(socket, reply);
```

Implementation:

Thread switching (i.e. blocking) and a scheduler

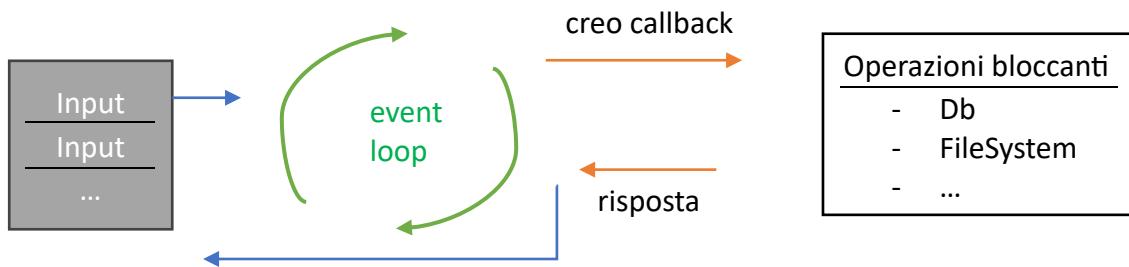
```
readRequest(socket, function(request) {
    processRequest(request,
        function (reply) {
            sendReply(socket, reply);
        });
});
```

Implementation:

Event queue processing

13.1 Event loop

Per ogni input ricevuto, un solo thread lavora su una coda di eventi.



Riceve il primo input ed esegue l'operazione associata, nel caso in cui coinvolga un DB l'operazione viene passata proprio ad esso e viene definita una callback per quando l'operazione sarà terminata. L'event loop, nel frattempo, continua ad eseguire operazioni! ad esempio quelle legate al secondo input (che a sua volta potrebbe definire un ulteriore callback).

Un solo thread gestisce tantissime richieste senza spreco di risorse, non viene infatti **mai bloccato e non è mai necessario context switching!**

In caso di I/O bloccante, un server “tradizionale” usa multi-threading (un thread per ogni richiesta) per limitare l'attesa; ma comunque ognuno passa maggior parte del tempo in attesa di I/O. Andare verso altissimi numeri di thread introduce overhead di context switching e significativo uso di memoria.

Allora quando usare Node.js?

Node.js è particolarmente adatto alla creazione di Web server e strumenti di networking vari.

→ Uso di una collezione di moduli che realizzano varie funzionalità core: per file system I/O, per networking, per funzioni crittografiche, per gestione stream di dati, ecc.

→ Insiemi di moduli (framework) per velocizzare lo sviluppo di applicazioni Web, come Express.js.

Non solo server-side (come già visto nell'esercitazione su React): offre diversi strumenti per sviluppo frontend.

Osservazione: Stile di programmazione Thread vs Node Callback

Threads	Callbacks
<pre>r1 = step1(); console.log('step1 done', r1); r2 = step2(r1); console.log('step2 done', r2); r3 = step3(r2); console.log('step3 done', r3); console.log('All Done!');</pre>	<pre>step1(function(r1) { console.log('step1 done', r1); step2(r1, function (r2) { console.log('step2 done', r2); step3(r2, function (r3) { console.log('step3 done', r3); console.log('All Done!'); }); }); });</pre>

Notiamo inoltre che, nonostante sia lato cliente sia lato server si usi javascript, il client lavora con il DOM mentre il server tipicamente con file e/o persistent storage.

13.2 Moduli Node.js

Il core di Node consiste di circa una ventina di moduli, alcuni di più basso livello come per la gestione di eventi e stream, altri di più alto livello come http.

```
// Carica il modulo http per creare un http server
var http=require('http');
// Configura HTTP server per rispondere con Hello World
var server=http.createServer(function(request,response) {
response.writeHead(200, {"Content-Type":"text/plain"});
response.end("Hello World\n");
});
// Ascolta su porta 8000
server.listen(8000);
// Scrive un messaggio sulla console terminale
console.log("Server running at http://127.0.0.1:8000/");
```

è stato progettato per essere piccolo e snello; i moduli che fanno parte del core si focalizzano su protocolli e formati di uso comune. Per ogni altra cosa, si usa **npm**.

NPM

- Package manager che semplifica sharing e riuso di codice JavaScript in forma di moduli
- Preinstallato con distribuzione Node
- Esegue tramite linea di comando e permette di ritrovare moduli dal registry pubblico in <http://npmjs.org>

Esempio di lettura di file in Node.js

```
var fs = require("fs"); // modulo fs richiesto
// oggetto fs fa da wrapper a chiamate bloccanti sui file
// read() a livello SO è sincrona bloccante mentre
// fs.readFile è non-bloccante
fs.readFile("smallFile", readDoneCallback); // Inizio lettura

function readDoneCallback(error, dataBuffer) {
// convenzione Node per callback: primo argomento è oggetto
// js di errore
if (!error) {
  console.log("smallFile contents", dataBuffer.toString());
}
}
```

13.3 Listener/Emitter

L'EventEmitter tiene traccia di un elenco di "listeners" per ciascun tipo di evento.

→ **listener** come funzione da chiamare quando evento associato viene lanciato

```
myEmitter.on('myEvent', function(param1, param2) {  
    console.log('myEvent occurred with ' + param1 + ' and '  
    + param2 + '!');  
});
```

→ **emitter** come segnale che un evento è accaduto

```
myEmitter.emit('myEvent', 'arg1', 'arg2');
```

L'emissione di un evento causa invocazione di TUTTE le funzioni listener.

In seguito a emit, i listener sono invocati in modo sincrono-bloccante e nell'ordine con cui sono stati registrati. No listener? No operazioni eseguite.

13.4 Stream

Node contiene moduli che producono/consumano flussi di dati (stream).

Si possono costruire stream anche dinamicamente e aggiungere moduli sul flusso.

Leggere File usando Stream

```
var readableStreamEvent =  
    fs.createReadStream("bigFile");  
readableStreamEvent.on('data', function (chunkBuffer) {  
    console.log('got chunk of', chunkBuffer.length,  
    'bytes');  
});  
  
readableStreamEvent.on('end', function() {  
// Lanciato dopo che sono stati letti tutti i datachunk  
console.log('got all the data');  
});  
  
readableStreamEvent.on('error', function (err) {  
console.error('got error', err);  
});
```

Scrivere File usando Stream

```
var writableStreamEvent =  
    fs.createWriteStream('outputFile');  
  
writableStreamEvent.on('finish', function () {  
console.log('file has been written!');  
});  
  
writableStreamEvent.write('Hello world!\n');  
writableStreamEvent.end();
```

13.5 TCP networking

Esiste un modulo di rete Node, chiamato net, che fa da wrapper per le chiamate di rete di SO
Include anche funzionalità di alto livello, come:

```
var net = require('net');
net.createServer(processTCPconnection).
    listen(4000);
```

Crea una socket, fa binding su porta 4000 e si mette in stato di listen per connessioni.

Per ogni connessione TCP, invoca la funzione processTCPconnection.

Esempio di Server per Chat

```
var clients = []; // Lista di client connessi
function processTCPconnection(socket) {
  clients.push(socket); // Aggiunge il cliente alla lista
  socket.on('data', function (data) {
    broadcast( "> " + data, socket);
    // invia a tutti i dati ricevuti });
  socket.on('end', function () {
    clients.splice(clients.indexOf(socket), 1); // elimina
    cliente da socket
  });
  // invia messaggio a tutti i clienti
  function broadcast(message, sender) {
    clients.forEach(function (client) {
      if (client === sender) return;
      client.write(message); });
  }
}
```

Osservazione finale:

Express.js è il framework più utilizzato oggi per lo sviluppo di applicazioni Web su Node

Node.js

Una volta installato Node.js, esiste la possibilità di imparare e verificare le proprie capacità “giocando” con learnyounode (da terminale). Per installarlo: `npm install -g learnyounode`.

Documentation: <https://nodejs.org/docs/latest/api/>

```
const x = require('x_mod');
```

`require` ci permette di includere nel nostro file moduli esterni.

La variabile nominata `x` da l'accesso all'intero modulo e alle sue operazioni associate.

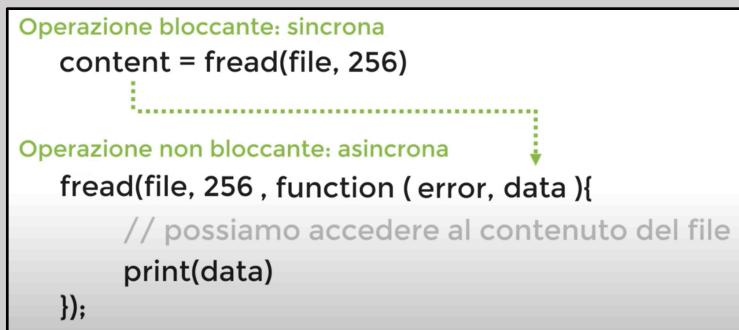
Operazione bloccante io-synch.js: legge le righe di un file

```
const fs = require('fs');
let file = fs.readFileSync(process.argv[2]);
let lines = file.toString().split('\n').length - 1;
console.log(lines);
```

Operazione NON-bloccante ASINCRONA io-Asynch.js: legge le righe di un file

Passiamo da una logica sincrona a una asincrona:

Non ci aspettiamo che il risultato di una funzione sia immagazzinato in una variabile ma usiamo una callback.



```
const fs = require('fs');
/*console.log("Inizio")*/
fs.readFile(process.argv[2], function(error, data){
  if(error) throw error;
  console.log(data.toString().split('\n').length - 1);
});
/*console.log("Fine")*/
```

NOTA: Le callback sono eseguite in maniera ASINCRONA NON BLOCCANTE verrebbe stampato:

Inizio

Fine

risultato_della_callback

Node.js

HelloWorld nel nostro server:

```
const http = require('http');

const hostname = 'localhost';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
  /*.end scrive ed indica che non verrà scritto più nulla dopo
   | un ulteriore write > res.write('Hello again'); lancerebbe errore
   */
});

server.listen(port, hostname, () => {
  console.log('server running');
});
```

Collegandosi a localhost:3000 verifichiamo il risultato

Esercizi svolti in lab:

- 1) Restituire una pagina Web con un input text readonly e una label per la presentazione del risultato. Conta parole delimitate da uno o più spazi.
- 2) Uso del modulo url, restituisce vari conteggi del file scelto
- 3) Uso del modulo querystring.
Si accetta come parametro di una GET una parola da cercare in un file specifico.
Se il numero di ripetizioni supera 5, riscrivere il file eliminando tutte le occorrenze di tale parola.

Nota:

Se si vuole restituire codice html, quando si crea il server...

```
res.setHeader('Content-type', 'text-html');
```

14 JSF e Web socket

Partiamo proponendo dei cenni su **JSF** (Java Server Faces). Una tecnologia basata su componenti da inserire nelle pagine Web ma anche collegati a componenti server-side (backing bean).

Presenta:

- Ricche API per rappresentazione componenti e gestione loro stato, gestione eventi, validazione e conversione dati server-side, definizione percorso navigazione pagine, supporto a internazionalizzazione.
- Ampia libreria di tag per aggiungere componenti a pagine Web e per collegarli a componenti server-side
- Offre un ottimo supporto alle Servlet; è una possibile alternativa a JSP

14.1 JSF

Vediamo un esempio esplicativo...

- Si può costruire un **Backing Bean** (o managed bean) in modo semplice:
Annotazione **@ManagedBean** regista automaticamente il componente come risorsa utilizzabile all'interno del container JSF, da parte di tutte le pagine che conoscano come riferirlo.

```
package hello;
import javax.faces.bean.ManagedBean;
@ManagedBean
public class Hello {
    final String world = "Hello World!";
    public String getworld() {
        return world;
    }
}
```

Il bean *Hello* è super-semplice, ma in generale contiene logica di business **controller** il cui risultato finale è, in modo diretto o tramite invocazione di altri componenti, di produrre dati **model**.

- Poi facile costruzione di pagina Web, scritta in XHTML, che usi il backing bean.
- Connessione tra pagina Web e componente tramite espressioni in **Expression Language (EL)**.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
    <title>Facelets Hello World</title>
</h:head>
<h:body>
    #{hello.world}
</h:body>
</html>
```

Esempio di pagina beanhello.xhtml

Facelets come linguaggio per costruzione di view JSF e di alberi di componenti.

- In tecnologia JSF, è inclusa servlet predefinita, chiamata FacesServlet, che si occupa di gestire richieste per pagine JSF (Serve mapping tramite solito descrittore di deployment (web.xml)).

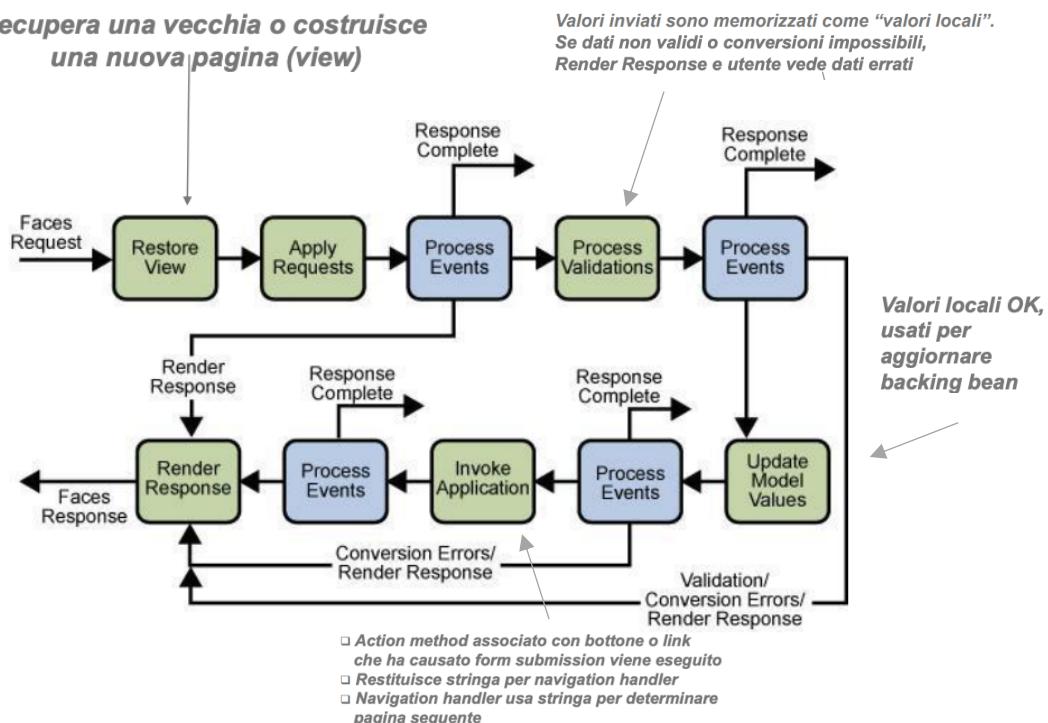
```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

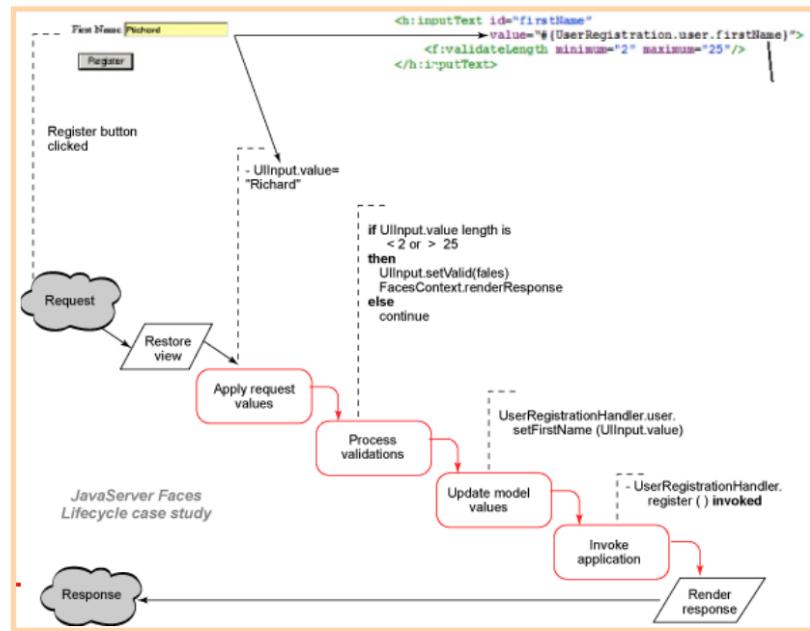
14.2 Ciclo di vita di un'applicazione Facelets

Il programmatore può anche non voler avere visibilità della gestione del ciclo di vita dell'applicazione Facelets, svolta automaticamente dal container per JSF.

Tipico ciclo di vita:

- Deployment dell'applicazione su server; prima che arrivi prima richiesta utente, applicazione in stato non inizializzato (anche non compilato...)
- Quando arriva una richiesta, viene creato un albero dei componenti contenuti nella pagina (messo in FacesContext), con validazione e conversione dati automatizzata
- Albero dei componenti viene popolato con valori da backing bean (uso di espressioni EL), con possibile gestione eventi e handler
- Viene costruita una view sulla base dell'albero dei componenti
- Rendering della vista al cliente, basato su albero componenti
- Albero componenti deallocated automaticamente
- In caso di richieste successive (anche postback), l'albero viene ri-allocato





I **Managed Bean** sono configurati nella seconda parte di faces-config.xml.

Sono semplici JavaBean che seguono regole standard:

- Costruttore senza argomenti (empty)
- No variabili di istanza public
- Metodi “accessor” per evitare accesso diretto a campi
- Metodi getXxx() e setXxx()

Ma presentano anche metodi detti “action”:

- Invocati automaticamente in risposta ad azione utente o evento
- Simili a classi Action in STRUTS

4 possibili scope:

- **Application** – singola istanza per applicazione
- **Session** – nuova istanza per ogni nuova sessione utente
- **Request** – nuova istanza per ogni richiesta
- **Scopeless** – acceduta anche da altri bean e soggetta a garbage collection come ogni oggetto Java

Configurazione JSF Managed Bean:

```

<managed-bean>
    <managed-bean-name>library</managed-bean-name>
    <managed-bean-class>com.oreilly.jent.jsf.library.model.Library
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
</managed-bean>

<managed-bean>
    <managed-bean-name>usersession</managed-bean-name>
    <managed-bean-class>com.oreilly.jent.jsf.library.session.UserSession
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
    <managed-bean-name>loginform</managed-bean-name>
    <managed-bean-class>com.oreilly.jent.jsf.library.backing.LoginForm
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>

```

14.3 JSF e templating

Facilità di estensione e riuso come caratteristica generale di JSF Templating: utilizzo di pagine come base (o template) per altre pagine, anche mantenendo look&feel uniforme.

Ad esempio, tag:

ui:insert – parte di un template in cui potrà essere inserito contenuto (tag di amplissimo utilizzo)

ui:component – definisce un componente creato e aggiunto all'albero dei componenti

ui:define – definisce contenuto con cui pagina “riempie” template (vedi insert)

ui: include – incapsula e riutilizza contenuto per pagine multiple

ui: param – per passare parametri a file incluso

14.4 Navigation rule

C'è necessità di un file configurazione specifico per JSF: faces-config.xml.

Soprattutto per configurazione *navigation rule* e *managed bean*.

Ogni regola di navigazione è come un flowchart con un ingresso e uscite multiple possibili.

Un singolo <from-view-id> per fare match con URI. Quando restituito controllo: stringa risultato viene valutata (ad es. success, failure, verify, login).

→ deve fare match con stringa risultato

→ determina URI verso cui fare forwarding

```
<navigation-rule>
    <from-view-id>/login.xhtml</from-view-id>
    <navigation-case>
        <from-action>#{LoginForm.login}</from-action>
        <from-outcome>success</from-outcome>
        <to-view-id>/storefront.xhtml</to-view-id>
    </navigation-case>

    <navigation-case>
        <from-action>#{LoginForm.logon}</from-action>
        <from-outcome>failure</from-outcome>
        <to-view-id>/logon.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Caso 1 di navigazione

Caso 2 di navigazione

14.5 WebSocket

A valle di quanto abbiamo visto...

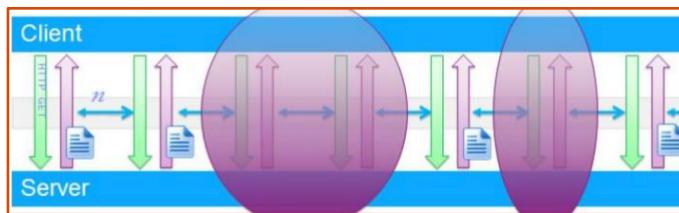
Le **web socket** nascono per risolvere il problema delle interazioni protocollo HTTP per comunicazione 2-way (client→server, server→client).

Possono servire a migliorare sviluppo (più facile e «naturale») ed esecuzione runtime di applicazioni Web bidirezionali e non strettamente request-response!

Le web socket permettono diversi tipi di connessione:

>>Polling

Il cliente fa richieste ad intervalli regolari a cui il server risponde immediatamente. Diventa inefficiente qual ora il server non ha nulla da inviare.

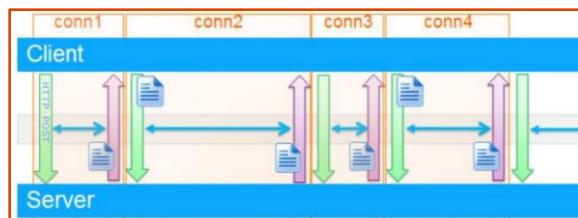


>>Long Polling

Il cliente manda una richiesta ed il server attende fino a quando non ha dati da inviare.

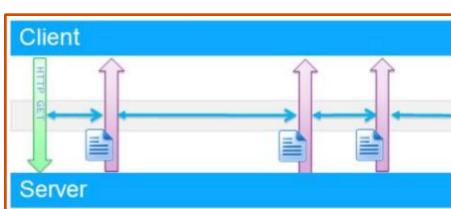
Appena ricevuta risposta il cliente reagisce mandando una nuova richiesta.

Ogni req/res si appoggia ad una nuova connessione!



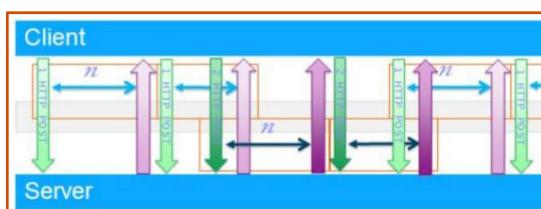
>>Streaming / Forever response

Il cliente manda una richiesta e il server risponde con uno streaming su una connessione mantenuta sempre aperta. È una half-duplex (solo server→client).

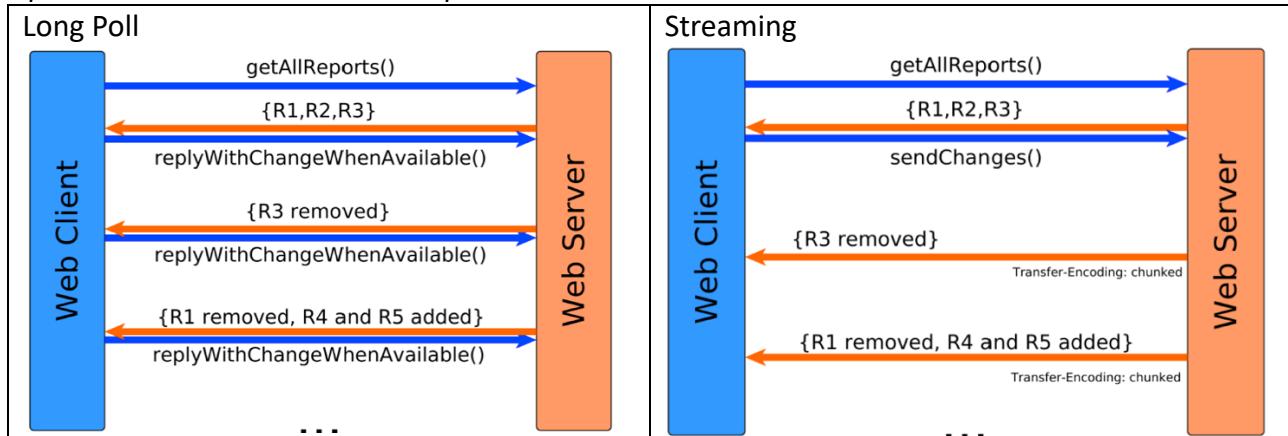


>>Connessioni Multiple

Si fa long polling su due connessioni separate, una client→server, l'altra server→client. Diventa complesso il coordinamento e la gestione di connessioni. (Overhead di due connection).



È possibile ottenere anche un comportamento simile con AJAX:



14.6 Caratteristiche WS

Bi-direzionali

➤ Client e server possono scambiarsi messaggi quando desiderano

Full-duplex

➤ Nessun requisito di interazione solo come coppia request/response e di ordinamento messaggi

Unica connessione long running

Visto come «upgrade» di HTTP

➤ Nessuno sfruttamento di protocollo completamente nuovo, nessun bisogno di nuova «infrastruttura»

Uso efficiente di banda e CPU

➤ Messaggi possono essere del tutto dedicati a dati applicativi

A livello di **protocollo** gli elementi base sono:

- Handshake: cliente e servitore cominciano una connessione (server accetta upgrade)
- Entrambi endpoint notificati che è aperta
- Entrambi possono inviare messaggi e chiudere socket in ogni istante

```

GET ws://server.org/wsendpoint
HTTP/1.1
Host: server.org
Connection: Upgrade
Upgrade: websocket
Origin: http://server.org
Sec-WebSocket-Version: 13
Sec-WebSocket-Key:
GhkZick+0/91FXIbUuR1VQ==
Sec-WebSocket-Extensions:
permessage-deflate;
client_max_window_bits

```

```

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: upgrade
Sec-WebSocket-Accept:
jpwu9a/SXDrssoRR26Oa3JUEFchY=
Sec-WebSocket-Extensions:
permessage-deflate;client_max_window_bits
...

```

Dati trasmessi con minimo overhead in termini di header.

14.7 WebSocket API server

Approccio integrato con Javascript lato cliente e programmazione JEE lato servitore.

Java API for WebSocket (JSR-356) lato servitore

- Gestione ciclo di vita
 - **onOpen, onClose, onError**
- Comunicazione tramite messaggi
 - **onMessage, send**
- Possibilità di uso di sessione
- Encoder e decoder per formattazione messaggi (messaggi <--> oggetti Java)

```
@ServerEndpoint("/actions")
public class WebSocketServer {

    @OnOpen
    public void open(Session session) { ... }

    @OnClose
    public void close(Session session) { ... }

    @OnError
    public void onError(Throwable error) { ... }

    @OnMessage
    public void handleMessage(String message, Session session) {
        // actual message processing
    }
}
```

Inviare e ricevere messaggi

Gli endpoint possono ricevere/inviare messaggi sotto forma di testo o binary.

per l'invio...

→ Ottenere oggetto Session dalla connessione

Disponibile come parametro in molti metodi. Ad esempio, nel metodo che ha ricevuto un messaggio (metodo annotato con @OnMessage); oppure, come variabile di istanza della classe endpoint nel metodo @OnOpen

→ Usare oggetto Session per ottenere un RemoteEndpoint

Session.getBasicRemote e Session.getAsyncRemote

restituiscono RemoteEndpoint.Basic e RemoteEndpoint.Async rispettivamente:

- void RemoteEndpoint.Basic.sendText(String text)
- void RemoteEndpoint.Basic.sendBinary(ByteBuffer data)
- void RemoteEndpoint.Basic.sendPing(ByteBuffer appData)

Per inviare messaggi a tutti i peer connessi a un Endpoint...

Ogni istanza di classe endpoint class è normalmente associata con una connessione e un peer; tuttavia, è possibile anche associare una istanza a una pluralità di peer connessi, per esempio per applicazioni di chat.

Si usa l'interfaccia Session del metodo getOpenSession:

```
@ServerEndpoint("/echoall")
public class EchoAllEndpoint {
    @OnMessage
    public void onMessage(Session session, String msg) {
        try {
            for (Session sess : session.getOpenSessions())
                if (sess.isOpen())
                    sess.getBasicRemote().sendText(msg);
        } catch (IOException e) { ... }
    }
}
```

per ricevere...

Si possono avere al massimo 3 metodi annotati con `@OnMessage` in un endpoint, uno per ogni tipo di messaggio, ovvero text, binary e pong.

```
@ServerEndpoint("/receive")
public class ReceiveEndpoint {
    @OnMessage
    public void textMessage(Session session, String msg) {
        System.out.println("Text message: " + msg);
    }
    @OnMessage
    public void binaryMessage(Session session, ByteBuffer msg) {
        System.out.println("Binary message: " +
                           msg.toString());
    }
    @OnMessage
    public void pongMessage(Session session, PongMessage msg) {
        System.out.println("Pong message: " +
                           msg.getApplicationData().toString());
    }
}
```

Mantenimento dello stato del cliente...

Il container lato server crea una istanza della classe endpoint per ogni connessione; quindi, si possono usare variabili di istanza per salvare dato cliente.

Inoltre, il metodo `Session.getUserProperties` restituisce una modifiable map per memorizzare le proprietà utente.

```
@ServerEndpoint("/delayedecho")
public class DelayedEchoEndpoint {
    @OnOpen public void open(Session session) {
        session.getUserProperties().put("previousMsg", " ");
    }
    @OnMessage public void message(Session session, String msg) {
        String prev = (String) session.getUserProperties()
            .get("previousMsg");
        session.getUserProperties().put("previousMsg", msg);
        try { session.getBasicRemote().sendText(prev); }
        catch (IOException e) { ... } }
}
```

Per info comuni a tutti i client, si possono anche utilizzare variabili di classe (static); in questo caso, responsabilità dello sviluppatore assicurare che accesso sia thread-safe.

Encoder e decoder...

Le Java API per WebSocket forniscono supporto per conversione di messaggi
WebSocket → oggetti Java (e viceversa) tramite encoder e decoder.

Uso di encoder

→ Implementare una di queste interfacce:

`Encoder.Text<T>` per messaggi testuali

`Encoder.Binary<T>` per messaggi binary

Queste interfacce specificano il metodo di encode; occorre implementare una encoder class per ogni tipo Java custom che si vuole inviare come messaggio WebSocket.

→ Aggiungere il nome delle classi encoder al parametro opzionale
della annotazione `ServerEndpoint`

→ Usare il metodo `sendObject(Object data)` di `RemoteEndpoint.Basic`
o di `RemoteEndpoint.Async`. Il container cerca un encoder che faccia match con il tipo e lo usa per la conversione verso un messaggio WebSocket.

```
public class MessageATextEncoder implements Encoder.Text<MessageA> {  
    @Override public void init(EndpointConfig ec) { }  
    @Override public void destroy() { }  
    @Override  
    public String encode(MessageA msgA) throws EncodeException {  
        // Access msgA's properties and convert to JSON text...  
        return msgAJsonString;  
    }  
}  
...
```

Similmente per MessageBTextEncoder

```
...  
@ServerEndpoint(  
    value = "/myendpoint",  
    encoders = { MessageATextEncoder.class, MessageBTextEncoder.class  
})  
public class EncEndpoint { ... }  
...  
MessageA msgA = new MessageA(...);  
MessageB msgB = new MessageB(...);  
session.getBasicRemote.sendObject(msgA);  
session.getBasicRemote.sendObject(msgB);
```

Come per gli endpoint, le istanze di encoder sono associate con una connessione e un peer WebSocket. Quindi c'è un solo thread ad eseguire il codice di una istanza di encoder ad ogni istante.

Uso di decoder

→ In modo analogo implementando una di queste interfacce:

[Decoder.Text](#) per messaggi testuali

[Decoder.Binary](#) per messaggi binary

```
public class MessageTextDecoder implements Decoder.Text<Message> {
    @Override public void init(EndpointConfig ec) { }
    @Override public void destroy() { }
    @Override public Message decode(String string) throws
        DecodeException {
        // Read message...
        if ( /* message is an A message */ )
            return new MessageA(...);
        else if ( /* message is a B message */ )
            return new MessageB(...);
    }
    @Override
    public boolean willDecode(String string) {
        return canDecode;
    }
}
```

```
@ServerEndpoint(
    value = "/myendpoint",
    encoders = {
        MessageATextEncoder.class, MessageBTextEncoder.class },
    decoders = { MessageTextDecoder.class }
)
public class EncDecEndpoint { ... }

@OnMessage
public void message(Session session, Message msg) {
    if (msg instanceof MessageA) {
        // We received a MessageA object...
    } else if (msg instanceof MessageB) {
        // We received a MessageB object...
    }
}
```

Come per gli endpoint, le istanze di decoder sono associate con una sola connessione e un solo peer WebSocket; quindi, un solo thread esegue il codice di una istanza di decoder in ogni istante.

14.8 WebSocket API cliente (Javascript)

```
var socket = new WebSocket("ws://server.org/
wsendpoint");
socket.onmessage = onMessage;

function onMessage(event) {
    var data = JSON.parse(event.data);
    if (data.action === "addMessage") {
        ...
        // actual message processing
    }
    if (data.action === "removeMessage") {
        ...
        // actual message processing
    }
}
```

Il **costruttore** ha sintassi: `var socket = new WebSocket(url [, protocols]);`

Alcune **proprietà**:

- `WebSocket.bufferedAmount` sola lettura, numero di byte di dati accodati
- `WebSocket.onclose` listener all'evento di chiusura della connessione
- `WebSocket.onerror` listener all'evento di errore sull'uso della WebSocket
- `WebSocket.onmessage` listener all'evento di ricezione di un messaggio dal server
- `WebSocket.onopen` listener all'evento di connessione aperta
- `WebSocket.protocol` sola lettura, sub-protocol selezionato dal servitore
- `WebSocket.readyState` sola lettura, stato corrente della connessione
(`WebSocket.CONNECTING` 0, `WebSocket.OPEN` 1, `WebSocket.CLOSING` 2, `WebSocket.CLOSED` 3)
- `WebSocket.url` sola lettura, URL assoluto associato

Metodi:

- `WebSocket.close([code[, reason]])` chiude la connessione
- `WebSocket.send(data)` accoda nuovi dati per l'invio

Eventi: Possibile agganciarsi a questi eventi usando `addEventListener()` o assegnando un event listener alla proprietà `onNomeEvento`.

- `close` Evento di chiusura connessione, anche disponibile tramite proprietà `onclose`
- `error` Evento di errore che ha prodotto la chiusura di WebSocket, ad esempio con mancato invio di un dato; anche disponibile tramite proprietà `onerror`
- `message` Evento associato alla ricezione di un messaggio dal server, anche disponibile tramite proprietà `onmessage`
- `open` Evento di apertura di una connessione WebSocket, anche disponibile tramite proprietà `onopen`

WebSocket 08_TecWeb

Una semplice calcolatrice. **Lato cliente...**

→ si deve verificare che quanto inserito nei campi di input sia corretto

→ trasformazione dei dati in JSON per il trasporto al servitore



Successivamente alla socket.send() i dati sono passati al server tramite WebSocket.

Il nostro server (è una servlet) ha un “Endpoint” specifico, al quale i clienti possono connettersi.

ProvaWS.java

```
1 package servlets;
2
3 import java.io.IOException;
4
5
6 @ServerEndpoint("/actions") ←
7 public class ProvaWS{
8
9
10     private Session sessione;
11     private Gson g;
12     private Map<String, Integer> reqCounter = new HashMap<String, Integer>();
13     private Map<String, OperationResp> lastOperations = new HashMap<String, OperationResp>()
```

Lato servitore...

→ si devono effettuare i calcoli

→ si mantiene il risultato dell’ultima operazione eseguita con successo

→ dati scambiati formato JSON

→ Nessuna info di stato deve essere visibile a tutti gli utenti

Ricordiamo il **ciclo di vita** di una WS lato servitore:

@OnOpen apertura nuova connessione

Il container lato server crea una istanza della classe endpoint per ogni connessione

```
@OnOpen  
public void open(Session session)  
{  
    this.sessione = session;  
    g = new Gson();  
    System.out.println("Connessione Aperta ");  
    reqCounter.put(session.getId(), 0);  
    lastOperations.put(session.getId(), new OperationResp());  
}
```

@OnClose chiusura connessione

```
@OnClose  
public void close(Session session)  
{  
    System.out.println("Connessione Chiusa ");  
}
```

@OnError in caso di errore

```
@OnError  
public void onError(Session session, Throwable throwable)  
{  
    System.out.println("Errore ");  
}
```

La WS riceve attraverso **@OnMessage**

```
@OnMessage  
public void handleMessage(String message, Session session) throws IOException, EncodeException {  
  
    int counter= reqCounter.get(session.getId());  
    OperationResp resp;  
    if(counter>100)  
    {  
        resp = new OperationResp();  
    }
```

Invia recuperando la sessione ed ottenendo RemoteEndpoint

```
private void sendback(String message) throws IOException, EncodeException {  
    // TODO Auto-generated method stub  
    try {  
        System.out.println("Sto inviando: "+message);  
        this.sessione.getBasicRemote().sendText(message);  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

(Nota che nel nostro esempio *message* è già stato convertito in JSON)

wssocket.js

Lato client usiamo la proprietà onmessage della socket

```
socket.onmessage = function (event){  
  
    var message = JSON.parse(event.data);  
    if(message.valid)  
    {  
        var log = document.getElementById("risultato");  
        log.value = "";
```

NOTA: Tramite JSON si passano classi Java Bean con determinati attributi. Importante conoscerli per poterli usare in JS e facilitarci il lavoro (*ES: var res = message.risultato*).

WebSocket 08_TecWeb VERSIONE BROADCAST

Ogni volta che un client inserisce un operando in input tutti i client vengono aggiornati.

```
<div id="calcolatrice">
    Operando 1: <input type="text" id="op1" size="3" onkeyup="myUpdate('op1');"><br>
    Operando 2: <input type="text" id="op2" size="3" onkeyup="myUpdate('op2');"><br>
    <br>Please select operator!</p>
```

Un'opportuna funzione myUpdate() costruisce un oggetto "update" da inviare al server tramite la socket.

```
function myUpdate(element)
{
    var elemento = document.getElementById(element);
    ...
    var json = JSON.stringify(updateReq);
    console.log("[myUpdate] sending", json);
    socket.send(json);
}
```

Il server ascolta su @OnMessage e controlla se arriva un update o una richiesta di calcolo.

Una volta scoperto costruisce la risposta opportuna e risponde IN BROADCAST!!!

```
private static void broadcast(String message) throws IOException, EncodeException {
    calcEndpoints.forEach(endpoint -> {
        try {
            endpoint.session.getBasicRemote().sendText(message);
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
}
```

Dall'altra parte il client controlla se gli arriva un update o un risultato.

```
socket.onmessage = function (event){
    console.log("[socket.onmessage]", event.data);
    var message = JSON.parse(event.data);
    if(event.data.includes("update"))
    {
        var toUpdate = message.update:
```

Problemi ricorrenti

RECUPERO DATI JSON CON GSON-----

Inviare dati in formato JSON ad una JSP NO AJAX

Se non posso usare AJAX posso inviare i **dati** di un **form** in formato JSP usando un **input nascosto**.

```
<form id="myForm" action="url_del_server_dove_gestire_i_dati" method="post">
    <input type="text" id="name" name="name"><br><br>
    <input type="email" id="email" name="email"><br><br>
    <input type="hidden" id="jsonData" name="jsonData">
    <button type="button" onclick="prepareAndSubmit()">Invia</button>
</form>

<script>
function prepareAndSubmit() {
    // Recupera i dati del modulo
    var formData = {
        name: document.getElementById("name").value,
        email: document.getElementById("email").value
    };
    // Converte i dati in JSON e li inserisce nel campo di input nascosto
    document.getElementById("jsonData").value = JSON.stringify(formData);
    // Invia il modulo
    document.getElementById("myForm").submit();
} </script>
```

Posso usare **JSON** per leggere i parametri ricevuti (JSP o Servlet).

```
<%@ page import="java.io.* , javax.servlet.* , javax.servlet.http.* , com.google.gson.*" %>
```

```
<%
    // Recupera il JSON inviato dal modulo
    String jsonData = request.getParameter("jsonData");

    // Effettua il parsing del JSON utilizzando GSON
    Gson gson = new Gson();
    JsonObject formData = gson.fromJson(jsonData, JsonObject.class);

    // Recupera i valori dei campi del modulo
    String name = formData.get("name").getAsString();
    String email = formData.get("email").getAsString();
%>
```

RECUPERO DATI JSON CON GSON-----

SERVLET che invia un Bean in JSON a una JSP o a JAVASCRIPT AJAX

Una **Servlet** può lavorare su un risultato da inviare sfruttando un **Bean**.

```
public class Persona {  
    private String nome;  
    private int eta;  
  
    // Costruttore vuoto  
    public Persona() {  
    }  
  
    // Metodi getter e setter per i campi  
    public String getNome() {  
        return nome;  
    } ...
```

Convertendo in **JSON**, **VENGONO CONSIDERATI SOLO GLI ATTRIBUTI**, non i metodi della classe:

```
g.toJson(persona)  
// formato dati  
{  
    "nome" : "Alessandro",  
    "eta" : 21  
}
```

- Se i dati sono inviati tramite **forward** ad una **jsp** posso recuperarli o come nel **caso precedente** o **importando il bean**

```
<%@ page import="your.package.Persona" %>  
<%  
    // Creazione di un oggetto Gson  
    Gson gson = new Gson();  
  
    // Converti il JSON in un oggetto Persona  
    Persona persona = gson.fromJson(jsonString, Persona.class);  
  
    // Utilizza i dati  
    out.println("Nome: " + persona.getNome() + "<br/>");  
    out.println("Cognome: " + persona.getCognome() + "<br/>");  
    out.println("Età: " + persona.getEta() + "<br/>");  
%>
```

posso passare anche liste e array

```
// Creazione di un oggetto Gson
Gson gson = new Gson();

// Converte il JSON in una lista di oggetti Persona
List<Persona> persone = gson.fromJson(jsonString, new ArrayList<Persona>().getClass());

// Itera attraverso la lista di persone e visualizza i dati
out.println("<h2>Dati delle persone:</h2>");
for (Persona persona : persone) {
    out.println("<p>Nome: " + persona.getNome() + "</p>");
    out.println("<p>Cognome: " + persona.getCognome() + "</p>");
    out.println("<p>Età: " + persona.getEta() + "</p>");
    //hobby è un array
    out.println("<p>Hobby:");
    out.println("<ul>");
    for (String hobby : persona.getHobby()) {
        out.println("<li>" + hobby + "</li>");
    }
    out.println("</ul></p>");
    out.println("<hr/>");
}
```

➤ Se inviati usando funzione **AJAX** (parliamo di JavaScript)

```
//Da servlet ad ajax
response.getWriter().print(g.toJson(persona));
```

```
//Callback ajax riceve responseText
//readyState = 4, invoco un'altra funzione che gestisce la risposta
result(theXhr.responseText);
```

```
function result(jsonText){
    var bean = JSON.parse(jsonText);
    var nome = bean.nome;
}
```

Guarda esame: 05_doppiaServAjax

JAVASCRIPT foreach di un array-----

Esempio d'uso della funzione di callback *forEach()* di un array:

```
//RICEVO L'OGGETTO RISULTATO in JSON che contiene SOLO gli attributi del bean Risultato
// autore : "SI"
// anagrammi : ["an1", "an2", ...]
//
var risultato = JSON.parse(jsoncontext);
var autore = risultato.autore;
var anagrammi = risultato.anagrammi;

element.innerHTML += "/n"+autore+"/n"
anagrammi.forEach((anagramma) => {
    element.innerHTML += anagramma + "/n";
});
```

JAVA foreach di un array, lista -----

Esempio d'uso della funzione di callback *forEach* di un **array**:

```
String[] nomi = {"Alice", "Bob", "Charlie", "David"};

// Utilizzo del foreach per attraversare l'array e stampare ogni elemento
System.out.println("Nomi nell'array:");
for (String nome : nomi) {
    System.out.println(nome);
}
```

Esempio d'uso della funzione di callback *forEach* di una **lista di qualsiasi <TIPO>**:

```
List<String> nomi = new ArrayList<>();
nomi.add("Alice");
nomi.add("Bob");
nomi.add("Charlie");
nomi.add("David");

// Utilizzo del foreach per attraversare la lista e stampare ogni elemento
System.out.println("Nomi nella lista:");
for (String nome : nomi) {
    System.out.println(nome);
}
```

USARE setAttribute -----

```
request.setAttribute();
```

Imposta un attributo sulla **RICHIESTA http**

L'attributo è accessibile (**request.getAttribute();**) in tutte le parti di codice con **STESSA RICHIESTA http**.

USARE session -----

(JAVA) La Servlet/JSP può **recuperare la sessione dell'utente** tramite la **richiesta**:

```
HttpSession session = request.getSession();
```

Posso settare **attributi visibili** per una **STESSA SESSIONE**:

```
session.setAttribute();
session.getAttribute();
```

Per una **JSP session** rappresenta la **SESSIONE CORRENTE**:

```
<%
    String nome = (String)session.getAttribute("nome");
%>
```

USARE getServletContext() -----

In Java **this.getServletContext().setAttribute()** serve ad impostare un attributo in un ambito di condivisione dati accessibile a tutte le servlet e a tutti i componenti dell'applicazione web.