



Tecnologie web

▼ 0.0 - Info

Exam

- Prova scritta
 - Struttura: 3 esercizi (11 punti ciascuno).
 - Durata: 3 ore
 - Info:
 - Possibilità di accedere ad appunti, libri e codice software.
 - Si consiglia vivamente di presentarsi alle esercitazione con una memoria esterna (512MB sono più che sufficienti). Tale memoria verrà utilizzata per contenere il workspace di Eclipse e le versioni portable di Tomcat. In tal modo sarà possibile salvare il lavoro svolto, ad esempio, i progetti realizzati, la configurazione del workspace ed il settaggio dei server.
- Prova orale
 - Struttura: domande veloci sull'intero programma.
 - Info:
 - Stesso appello d'esame della prova scritta.
 - È possibile accedervi solo se sono stati realizzati almeno 6 punti su ogni esercizio della prova scritta.
 - Fornisce un punteggio che varia da +4 punti a -4 punti rispetto al punteggio acquisito nella prova pratica.

Sito web

<http://lia.disi.unibo.it/Courses/twt2425-info/>

▼ 1.0 - Introduzione

Breve storia del web

Il www (world wide web) è stato proposto nel 1989 da Tim Berners Lee. L'idea alla base era la seguente:

- Documenti statici.
- Ipertestuale.
- Protocollo semplice.

All'inizio veniva utilizzato all'interno del dominio del CERN, poi è stato esteso a tutto il mondo.

Successivamente è avvenuta la creazione dell'HTTP, passando dalla versione 1.0 alla 1.1, poi alla 2 ed infine alla 3 creata nel 2022.

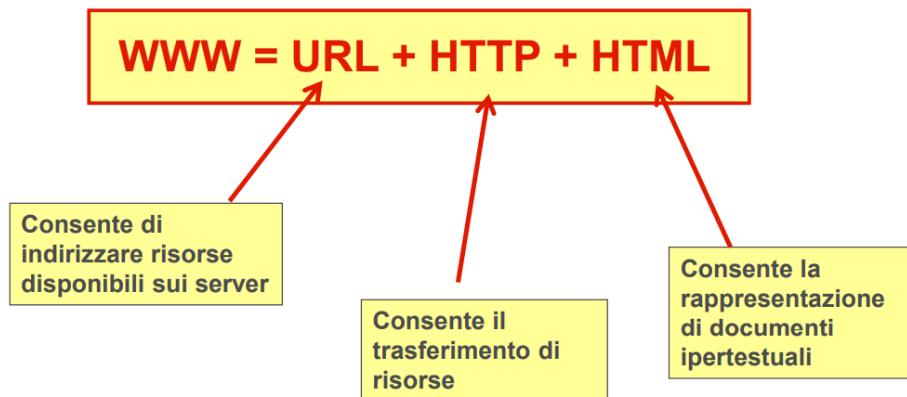
Per ipertesto si intende l'insieme di documenti messi in relazione tra loro tramite collegamenti monodirezionali. Può essere visto come una rete (grafo) avente i documenti al posto dei nodi.

Nel www i documenti, chiamati pagine, risiedono su server geograficamente distribuiti e da un qualunque documento è possibile passare ad un altro (navigazione). Per realizzare questo tipo di ipertesto è dunque necessario disporre di 3 elementi concettuali:

- Meccanismo per localizzare un documento.
- Protocollo per accedere alle risorse.
- Linguaggio per descrivere i documenti ipertestuali.

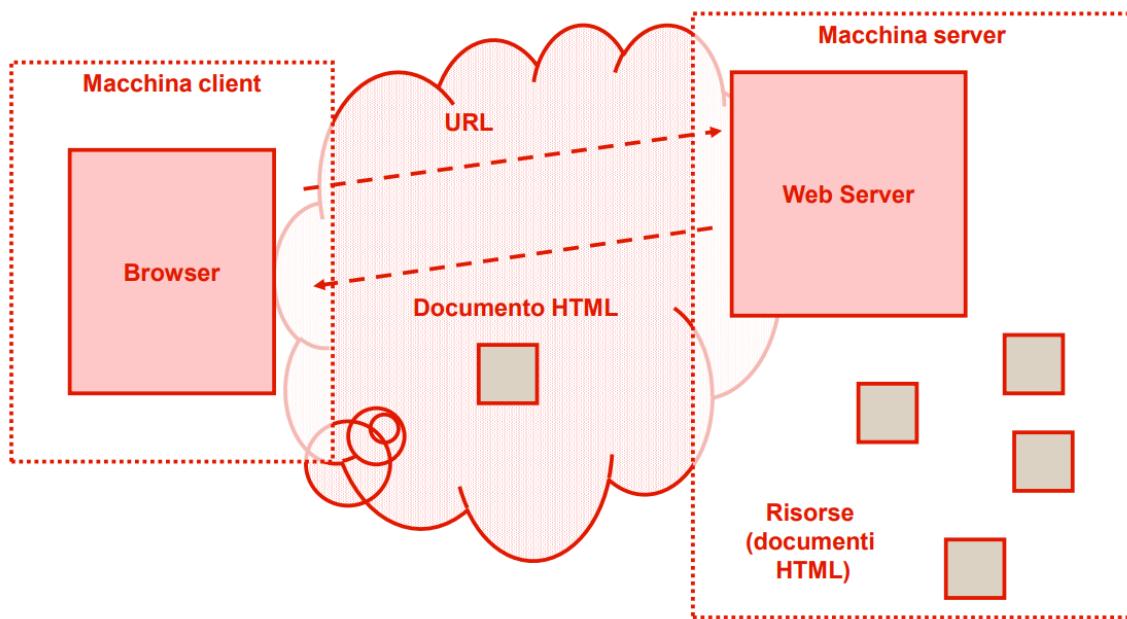
e di 2 elementi fisici:

- Server in grado di erogare le risorse.
- Client per visualizzare i documenti e permettere la navigazione.



Il web segue dunque un modello client-server, con:

- Client attivi, web browser.
- Server passivi, web server.



La home page è la pagina di accesso di un server web, la quale contiene i link per le altre pagine.

▼ 2.0 - URI e URL

Introduzione

$$\text{WWW} = \text{URL} + \text{HTTP} + \text{HTML}$$

3 problemi principali:

- Come identificare il server.
- Come identificare la risorsa.
- Quali meccanismi per accedere alla risorsa.

URI (Uniform Resource Identifier) è un meccanismo semplice per identificare una risorsa. Un URI non fa riferimento necessariamente a risorse accessibili tramite HTTP o a entità disponibili in rete, è un mapping concettuale a un'entità, che può anche variare il suo contenuto nel tempo.

Struttura degli URI

Gli URI sono stringhe con una sintassi definita, aventi questa forma generale:

<scheme>:<scheme-specific-part>

Per la componente <scheme-specific-part> non esiste una struttura o una semantica comune a tutti gli URI.

Esiste però un sottoinsieme di URI che condivide una sintassi comune:

<scheme>://<authority><path>?<query>

A parte <scheme>, le altre parti possono talora essere omesse.

Esistono due specializzazioni del concetto di URI:

- Uniform Resource Name (URN): identifica una risorsa per mezzo di un nome globalmente unico (es. codice ISBN che identifica in modo univoco un libro).



- Uniform Resource Locator (URL): identifica una risorsa per mezzo del suo meccanismo di accesso primario (es. locazione nella rete). Tiene conto anche della modalità per accedere alla risorsa, specificando il protocollo necessario (es. http), solitamente nello schema. La parte restante dipende poi dal protocollo.

Nella sua forma più comune (http, https, ftp ecc.), la struttura è la seguente:

```
<protocol>://[<username>:<password>@]
<host>[:<port>] /<path>[?<query>] [#fragment]
```

- `<protocol>`: Descrive il protocollo da utilizzare per l'accesso al server.
- `<username>:<password>@`: credenziali per l'autenticazione.
- `<host>`: indirizzo server su cui risiede la risorsa.
- `<port>`: definisce la porta da utilizzare. Se non viene indicata, si usa porta standard per il protocollo specificato (per HTTP è 80).
- `<path>`: percorso che identifica la risorsa nel file system del server. Se manca, tipicamente si accede alla risorsa predefinita (es. home page).
- `<query>`: una stringa di caratteri che consente di passare al server uno o più parametri. Di solito ha questo formato:
parametro1=valore¶metro2=valore2...

URI opache e URI gerarchiche

Le URI possono essere anche classificate come opache o gerarchiche:

- URI opaca: non è soggetta a ulteriori operazioni di parsing (es. mailto:paolo.rossi@disi.unibo.it).
- URI gerarchica: è soggetta a ulteriori operazioni di parsing, per esempio per separare l'indirizzo del server dal percorso all'interno file system (es.

<http://informatica.unibo.it/>).

Le operazioni che si possono effettuare sulle URI gerarchiche sono le seguenti:

- Normalizzazione: processo di rimozione dei segmenti "." e ".." e altri caratteri speciali
- Risoluzione: processo che a partire da una URI originaria porta all'ottenimento di una URI risultante. La URI originaria viene risolta basandosi su una terza URI, detta base URI.
- Relativizzazione: processo inverso della risoluzione.
 - **URI originaria:**
<http://disi.unibo.it/docs/guide/collections/designfaq.html#28>
 - **Base URI:**
<http://disi.unibo.it/>
 - **Risultato:**
<http://disi.unibo.it/docs/guide/collections/designfaq.html#28>

Esempio di risoluzione.

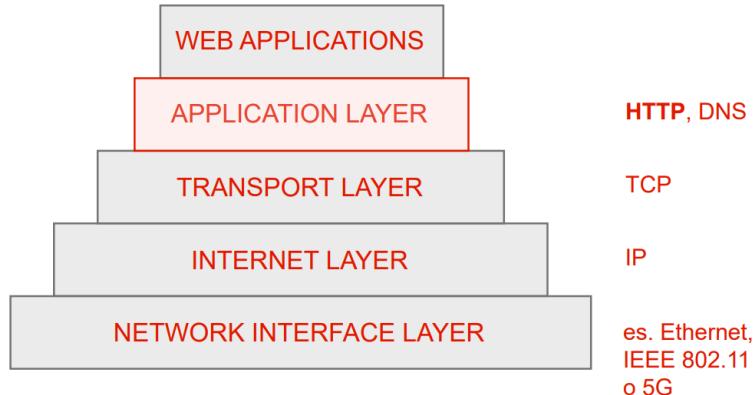
▼ 3.0 - Il protocollo HTTP

Introduzione

WWW = URL + HTTP + HTML

HTTP (HyperText Transfer Protocol) è un protocollo di livello applicativo utilizzato per trasferire risorse web da server a client. Tale protocollo gestisce richieste URL e risponde al client. È di tipo stateless, ovvero né il server né il client mantengono informazioni relative ai messaggi precedentemente scambiati.

HTTP si situa a livello application nello stack TCP/IP:



HTTP è un protocollo basato su TCP, dunque sia le richieste che le risposte sono trasmesse utilizzando stream TCP. Lo schema è di questo tipo:

1. Il server rimane in ascolto, tipicamente sulla porta 80.
2. Il client apre una connessione TCP sulla porta 80.
3. Il server accetta la connessione.
4. Il client invia una richiesta.
5. Il server invia una risposta e chiude la connessione.

Versioni HTTP

- HTTP 1.0: connessioni non persistenti. Se all'interno della pagina richiesta c'è un HTML con alcune immagini jpeg al suo interno, il client ripete i passi sopra citati per ciascuna immagine.
- HTTP 1.1: connessioni persistenti. Qui il server chiude la connessione quando viene specificato nell'header del messaggio, oppure quando scade il timeout.

Qui si può usare anche il pipelining, ovvero il client può inviare molteplici richieste prima di terminare la ricezione delle risposte. Le risposte devono essere inviate nello stesso ordine delle richieste.

- HTTP 2: basato su SPDY, è un protocollo open networking promosso da google.
- HTTP 3: miglioramento di HTTP tramite integrazione del trasporto QUIC (invece di TCP).

Messaggio HTTP

Il messaggio HTTP, costituito da un insieme di coppie nome-valore, è definito da 2 strutture:

- Message Header: contiene tutte le informazioni necessarie per l'identificazione del messaggio.
- Message Body: contiene i dati trasportati dal messaggio.

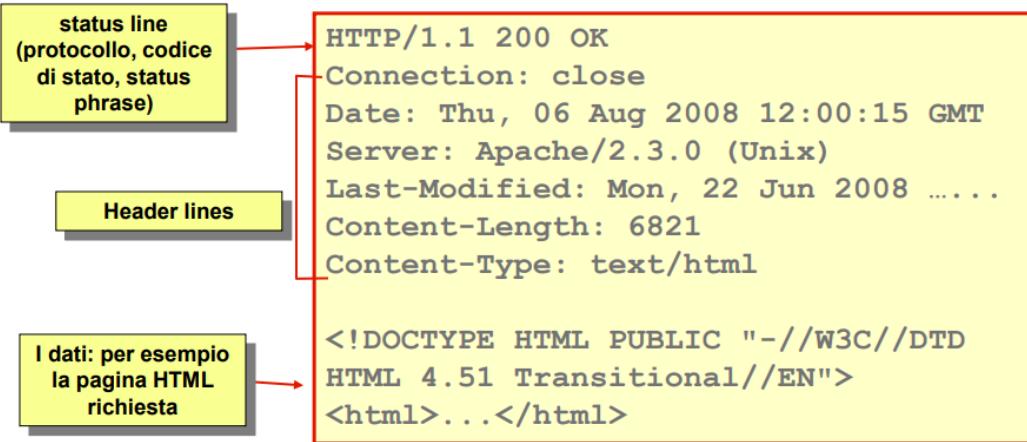
Il protocollo utilizza messaggi in formato ASCII.

I comandi delle richieste

- GET: richiede una risorsa ad un server. È previsto il passaggio di parametri nella parte <query> dell'url. La lunghezza massima di un URL è però limitata.
- POST: richiede una risorsa ad un server. A differenza di GET, i dettagli per l'identificazione ed elaborazione della risorsa non sono nell'URL, ma nel body del messaggio. Non ci sono limiti di lunghezza.
- PUT: richiede la memorizzazione sul server di una risorsa all'URL specificato.
- DELETE: richiede la cancellazione della risorsa riferita dall'URL specificato. Normalmente è disabilitato sui server pubblici.
- HEAD: simile a GET, ma il server deve rispondere con gli header relativi, senza body.
- OPTIONS: richiede informazioni sulle opzioni disponibili per la comunicazione.
- TRACE: invoca il loop-back remoto a livello applicativo del messaggio di richiesta. Consente al client di vedere che cosa è stato ricevuto dal server.

La risposta del server

La risposta viene poi inviata dal server nel seguente formato:



Nella risposta è incluso lo status code, ovvero un codice di 3 cifre di cui la prima indica la classe della risposta e le altre due la risposta specifica.

Ci sono 5 classi:

- 1xx: Informational. Una risposta temporanea alla richiesta, durante il suo svolgimento (sconsigliata a partire da HTTP 1.0)
- 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.

Alcuni esempi di codici di stato:

- 100 Continue (se il client non ha ancora mandato il body, deprecated da HTTP 1.0)
- 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)

I cookie

Parallelamente alle sequenze request/response, il protocollo prevede una struttura dati che si muove come un token, dal client al server e viceversa: i cookie.

I cookie possono essere generati sia dal client che dal server, e 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 stateless.

I cookie sono una collezione delle seguenti stringhe:

- 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.
- Max-age: (opzionale) numero di secondi di vita.
- Secure: (opzionale) questi cookie vengono trasferiti se e soltanto se il protocollo è sicuro (https). Non è molto usato.
- Version: identifica la versione del protocollo di gestione dei cookie.

Autenticazione

Esistono situazioni in cui si vuole restringere l'accesso alle risorse ai soli utenti abilitati. Ciò viene messo in atto tramite l'autenticazione, la quale può essere attuata tramite diverse tecniche:

- Filtro su set di indirizzi IP

È una soluzione che prevede vari svantaggi:

- Non funziona se l'indirizzo non è pubblico.
- Non funziona se l'indirizzo è assegnato dinamicamente.
- Esistono tecniche che consentono di presentarsi con un IP fasullo.
- Form per la richiesta di username e password (normalmente si utilizza il metodo POST)
- HTTP Basic

Il funzionamento è il seguente: il client fa una richiesta al server che risponde chiedendo l'autenticazione con uno stato di errore 401 Authorization Required. Il client deve dunque inviare le proprie credenziali (username e password) nella richiesta successiva, codificandole in Base64 nell'header.

- HTTP Digest

Tutti questi metodi devono essere accompagnati da sicurezza nel canale di trasporto, con tecniche come quella dell'SSL (Secure Sockets Layer) o TLS (Transport Layer Security), che è alla base di HTTPS. Queste tecniche pongono un livello che si occupa della gestione di confidenzialità, autenticità ed integrità della comunicazione fra HTTP e TCP, basato su crittografia a chiave pubblica (private key + public key e un certificato, usato in genere per autenticare il server).

Architetture più distribuite e articolate per il Web

- Proxy: programma applicativo in grado di agire sia come Client che come Server al fine di effettuare richieste per conto di altri clienti. Le Request vengono processate internamente oppure vengono ridirezionate al Server, e in tal caso il proxy deve interpretare e, se necessario, riscrivere le Request prima di inoltrarle.
- Gateway: Server che agisce da intermediario per altri Server. Al contrario dei proxy, il gateway riceve le request come se fosse il server originale e i Client non sono in grado di identificare che Response proviene da un gateway. Viene detto anche reverse proxy o server-side proxy.

- Tunnel: programma applicativo che agisce come "blind relay" tra due connessioni. Una volta attivato, il tunnel non partecipa attivamente alla comunicazione HTTP, ma semplicemente instrada i dati tra il client e il server senza modificarli, e ciò è utile per garantire privacy e sicurezza.

Caching distribuito sul web

L'idea di base del caching distribuito sul web è quella di memorizzare copie temporanee di documenti Web (es. pagine HTML, immagini) al fine di riutilizzarli per le successive richieste per ridurre l'uso della banda ed il carico sul server.

Tipi di Web cache sono i seguenti:

- User Agent Cache

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

- Proxy Cache

- Forward Proxy Cache: il proxy intercetta il traffico e mette in cache le pagine, permettendo a successive richieste di non dover provocare lo scaricamento di ulteriori copie delle pagine al server.
- Reverse (o server-side) Proxy Cache: il gateway cache intercetta il traffico e mette in cache le pagine. I client non sono in grado di capire se le pagine arrivano dal server o dal gateway.

▼ 4.0 - HTML

Introduzione

WWW = URL + HTTP + HTML

HTML è l'acronimo di HyperText Markup Language. È il linguaggio utilizzato per descrivere le pagine che costituiscono i nodi dell'ipertesto. È un linguaggio di codifica del testo del tipo a marcatori (markup).

Sistemi di codifica caratteri

In generale, ogni documento elettronico è costituito da una stringa di caratteri. Per codificare i caratteri si stabilisce una corrispondenza biunivoca tra elementi di una collezione ordinata di caratteri e un insieme di codici numerici. Si ottiene così un coded character set che di solito si rappresenta in forma di tabella.

Per ciascun coded character set si definisce poi una codifica dei caratteri (character encoding). La codifica mappa una o più sequenze di byte (8 bit) a un numero intero che rappresenta un carattere nel coded character set creato in precedenza.

I sistemi di codifica caratteri più noti sono l'ASCII e l'UTF-8 (più moderno, sta sostituendo l'ASCII).

Linguaggi a marcatori

Un linguaggio di mark-up è composto da:

- un insieme di istruzioni dette tag o mark-up (marcatori).
- una grammatica che regola l'uso del mark-up.
- una semantica che definisce il dominio di applicazione e la funzione del mark-up.

HTML e SGML

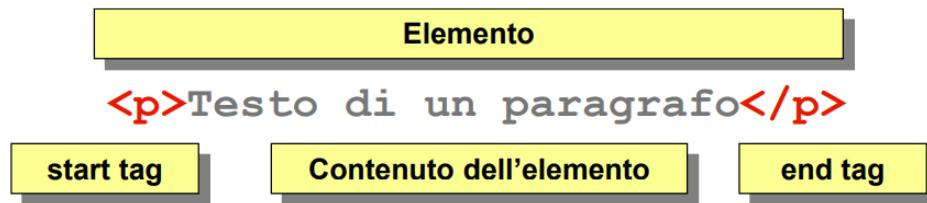
SGML (Standard Generalized Markup Language) è uno standard ISO per rappresentare documenti elettronici che comprende oggetti di varia classi chiamati elementi.

HTML è un'applicazione/semplicificazione di SGML, ovvero un linguaggio per la rappresentazione di un tipo di documento SGML. Oltre a descrivere il contenuto, HTML associa anche significati grafici agli elementi che definisce. L'ultima versione di HTML è HTM5.

I tag

I tag HTML sono usati per definire il mark-up di elementi HTML. Sono preceduti e seguiti rispettivamente da due caratteri < e > (parentesi angolari). Sono normalmente accoppiati; un esempio è dato da: <p> e </p>, detti rispettivamente start tag ed end tag. 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:



HTML rispetta in maniera poco rigorosa le specifiche SGML, ad esempio:

- Ammette elementi senza chiusura come `
`.
- I tag non sono case sensitive.
- L'apertura e chiusura di tag annidati può essere "incrociata"

`<i>...</i>`

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.

`<i>...</i>`

Attributi

Un elemento può essere dettagliato mediante attributi. 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 (gli apici possono essere omessi se il valore non contiene spazi).

Tipi MIME

Lo standard MIME rappresenta il tipo di contenuto di un messaggio. Classifica i tipi di contenuto sulla base di una logica a due livelli ed è largamente

utilizzata in ambito di HTML e delle tecnologie web in generale.

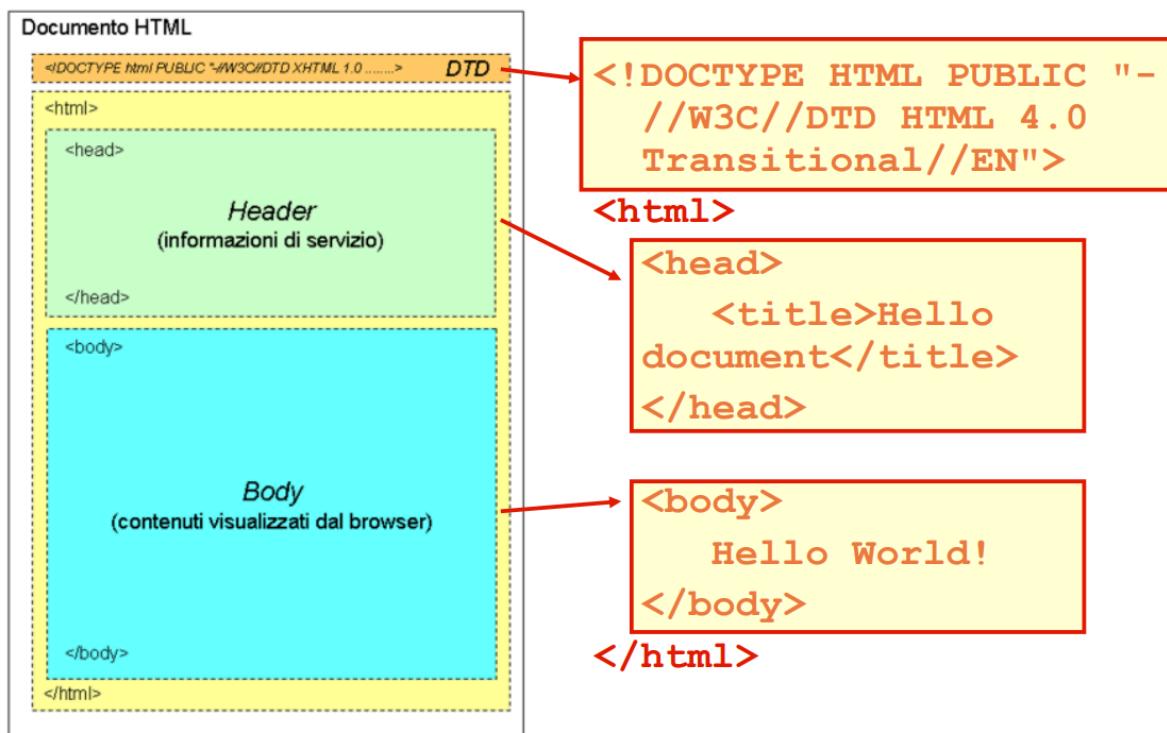
Un tipo MIME è espresso con questa sintassi: tipo/sottotipo (es. text/plain, text/html).

Commenti

È possibile inserire commenti in qualunque punto all'interno di una pagina HTML con la seguente sintassi:

```
<!-- Questo è un testo di commento -->
```

Struttura base di un documento HTML



DTD

Il primo elemento di un documento HTML è la definizione del tipo di documento (Document Type Definition o DTD), ad esempio:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
http://www.w3.org/TR/html4/loose.dtd>
```

Serve al browser per identificare le regole di interpretazione e visualizzazione da applicare al documento.

È costituita da diverse parti:

- HTML il tipo di linguaggio utilizzato è l'HTML
- PUBLIC il documento è pubblico
- W3C ente che ha rilasciato le specifiche
- DTD HTML 4.01 Transitional: versione di HTML
- EN la lingua con cui è scritta il DTD è l'inglese
- http://... URL delle specifiche

Header

È identificato dal tag `<head>` e contiene elementi non visualizzati dal browser.

I tag che si possono inserire al suo interno sono:

- `<title>` titolo della pagina.
- `<meta>` metadati, informazioni utili ad applicazioni esterne o al browser.
Esistono due tipi di elementi meta, distinguibili dal primo attributo: `http-equiv`
 - `name` .Gli elementi di tipo `http-equiv` danno informazioni al browser su come gestire la pagina e hanno una struttura di questo tipo: `<meta http-equiv=nome content=valore>`.
 - `refresh`: indica che la pagina deve essere ricaricata dopo un numero di secondi.

`<meta http-equiv=refresh content=45>`

- `expires`: stabilisce una data scadenza (fine validità) per il documento.

`<meta http-equiv=expires content="Tue, 20 Aug 1996 14:25:27 GMT">`

- `content type`: definisce il tipo di dati contenuto nella pagina (di solito il tipo MIME `text/html`).

```
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1">
```

Gli elementi di tipo name forniscono informazioni utili ma non critiche, e hanno una struttura di questo tipo: `<meta name=nome content=valore>`.

- author: autore della pagina.

```
<meta name=author content='John Smith'>
```

- description: descrizione della pagina.

```
<meta name=description content="Home page UNIBO">
```

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

```
<meta name=copyright content="Copyright 2009, John Smith">
```

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

```
<meta name=keywords lang="en" content="computer documentation, computers, computer help">
```

- date: data di creazione del documento.

```
<meta name="date" content="2008-05-07T09:10:56+00:00">
```

- `<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.
- `<script>` codice eseguibile utilizzato dal documento.
- `<style>` informazioni di stile (CSS locali).

Body

Il tag `<body>` delimita il corpo del documento. Contiene la parte che viene mostrata dal browser. Ammette alcuni attributi come:

- `background = uri`

Definisce l'URI di una immagine da usare come sfondo per la pagina

- `text = color`

Definisce il colore del testo

- `bgcolor = color`

In alternativa a background definisce il colore di sfondo della pagina

- `lang = linguaggio`

Definisce il linguaggio utilizzato nella pagina (es. language="it").

Tipi di elementi nel body

Nel body ci sono diversi i seguenti tipi di elementi:

- 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: campi di inserimento, checkbox e radio button, menu a tendina, bottoni, ecc.
- Collegamenti ipertestuali e ancora.
- 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": costituiscono un blocco attorno a sé e occupano l'intera larghezza disponibile (paragrafi, tavole, form...).
Un elemento block-level può contenere altri elementi dello stesso tipo o di tipo inline.
- Elementi "inline": non iniziano su una nuova riga e possono essere integrati nel testo (link, immagini,...).

Un elemento inline può contenere solo altri elementi inline.

- Liste: numerate, puntate.

Un'altra distinzione da ricordare è quella tra elementi rimpiazzati (replaced elements) ed elementi non rimpiazzati:

- Gli elementi rimpiazzati sono quelli di cui il browser conosce le dimensioni intrinseche (es. img, input, textarea, select).
- Tutti gli altri elementi sono in genere considerati non rimpiazzati.

Una lista di elementi utilizzabili in HTML è la seguente:

▼ Heading

I tag `<h1>, <h2>...<h6>` servono per definire dei titoli di importanza descrescente.

Ammettono attributi di allineamento: `<h1 align=left|center|right|justify>`.

▼ Paragrafi

Il tag `<p>` serve per definire un paragrafo, ovvero l'unità di base entro cui suddividere un testo: è un elemento di tipo blocco. Lascia una riga vuota prima della sua apertura e dopo la sua chiusura.

▼ Div

Se al posto di `<p>` si usa il tag `<div>` il blocco di testo lascia spazi prima e dopo la sua apertura.

▼ Span

Lo `` è un contenitore generico di tipo inline, quindi non va a capo ma continua sulla stessa linea del tag che lo include.

▼ Horizontal rule

Il tag `<hr>` serve ad inserire una riga di separazione.

Attributi:

- `align` = {left|center|right}
- `size` = pixels
- `width` = length

- `noshade` (Riga senza effetto di ombreggiatura)

▼ Liste non ordinate

Il tag `` (unordered list) permette di definire liste non ordinate (puntate).

Gli elementi della lista vengono definiti mediante il tag `` (list item).

L'attributo `type` definisce la forma dei punti e ammette 3 valori: disc, circle e square.

```
<ul type="disc">
  <li>Unordered information.</li>
  <li>Ordered information.</li>
  <li>Definitions.</li>
</ul>
```

▼ Liste ordinate

Il tag `` (ordered list) permette di definire liste ordinate (numerati). Gli elementi vengono definiti mediante il tag ``.

L'attributo `type` definisce il tipo di numerazione e ammette 5 valori: 1 (1,2,...), a (a,b,...), A (A,B,...), i (i,ii,...) e l (l,II,...).

```
<ol type="I">
  <li>Unordered information.</li>
  <li>Ordered information.</li>
  <li>Definitions.</li>
</ol>
```

▼ Liste di definizione

Il tag `<dl>` (definition list) permette di definire liste di definizione. Sono liste costituite alternativamente da termini (tag `<dt>`) e definizioni (tag `<dd>`).

```
<dl>
  <dt><strong>UL</strong></dt>
  <dd>Unordered List.</dd>
  <dt><strong>OL</strong></dt>
```

```
<dd>Ordered List.</dd>
</dl>
```

▼ Tabelle

`<table border="1" >
 <caption align="top">
 A test table with merged cells</caption>
 <tr>
 <th rowspan="2"></th>
 <th colspan="2">Average</th>
 <th rowspan="2">>Red
eyes</th>
 </tr>
 <tr><th>height</th><th>weight</th></tr>
 <tr><th>Males</th><td>1.9</td><td>0.003</td><td>40%</td></tr>
 <tr><th>Females</th><td>1.7</td><td>0.002</td><td>43%</td></tr>
</table>`

	Average		Red eyes
	height	weight	
Males	1.9	0.003	40%
Females	1.7	0.002	43%

Il tag `<table>` racchiude la tabella. Attributi:

- `align` = "{left|center|right}"
- `width` = "n|n%"
- `bgcolor` = "#xxxxxxxx"
- `border` = "n"
- `cellspacing`, `cellpadding`

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

Attributi:

- `align` = "{left|center|right|justify}"

- `valign` = "{top|middle|bottom|baseline}": allineamento verticale del contenuto delle celle della riga
- `bgcolor` = "#xxxxxx"

`<th>` e `<td>` sono i tag che racchiudono le celle. `<th>` serve per le celle della testata (in grassetto e centrate), mentre `<td>` serve per le celle del contenuto.

Attributi:

- Gli stessi di `<tr>`
- `width`, `height` = {length|length%}
- `rowspan`, `colspan` = n: indica su quante righe/colonne della tabella si estende la cella

▼ Link ipertestuali

Un link è costituito da due estremi, detti ancora. Link = source anchor → destination anchor.

In HTML le ancora, sia di origine che di destinazione, si esprimono utilizzando il tag `<a>`. Le ancora di origine sono caratterizzate da un attributo, denominato `href`, che contiene l'indirizzo di destinazione (è un URL, che può essere assoluto o relativo). Le ancora di destinazione sono invece caratterizzate dall'attributo `name`.

```

<p>
  <a href="#section1">
    Introduzione</a><br>
  <a href ="#section2">
    Concetti di base</a><br>
  <a href ="#section2.1">
    Definizione del problema</a><br>
    ...
</p>

<h2><a name="section1">
  Introduzione</a></h2>
  ...sezione 1...
<h2><a name ="section2">
  Concetti di base</a></h2>
  ...sezione 2...
<h3><a name ="section2.1">
  Definizione del problema</a></h3>
  ...sezione 2.1...

```

Si può esprimere un'ancora di destinazione in forma implicita, cioè senza utilizzare il tag `<a>`. Per fare ciò è sufficiente assegnare l'attributo ID a un qualunque elemento della pagina.

```

<p>
  <a href="#section1">
    Introduzione</a><br>
  <a href="#section2">
    Concetti di base</a><br>
  <a href="#section2.1">
    Definizione del problema</a><br>
    ...
</p>

<h2 id="section1">
  Introduzione</h2>
  ...sezione 1...
<h2 id="section2">
  Concetti di base</h2>
  ...sezione 2...
<h3 id="section2.1">
  Definizione del problema</h3>
  ...sezione 2.1...

```

Per il link ad una risorsa esterna si inserisce l'URL di tale risorsa nell'ancora di origine. Il caso più completo è quello di un link ad un punto preciso di un'altra risorsa, come nel seguente esempio:



▼ Immagini

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

```

```

Attributi:

- `src` = uri
- `alt` = text: testo alternativo nel caso fosse impossibile visualizzare l'immagine
- `align` = {bottom|middle|top|left|right} (deprecato)
- `width`, `height` = pixels
- `border` = pixels (deprecato)

▼ Form

Un form (modulo) è una sezione di documento HTML che contiene elementi di controllo che l'utente può utilizzare per inserire dati o in generale per interagire.

Il tag `<form>` racchiude tutti gli elementi del modulo (è un elemento di tipo blocco).

Attributi:

- `action` = uri: URI dell'agente che riceverà i dati del form
- `name` = text: nome del form
- `method` = {get|post}
- `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

Attributi:

- `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 (****)

- ▼ `file`: controllo che consente di caricare un file

Richiede una codifica particolare per il form (`enctype="multipart/form-data"`), perché le informazioni trasmesse con il POST contengono tipologie di dati diverse.

Attributi:

- `name` = text: nome del controllo
- `value` = content-type: lista di MIME types per l'upload

- ▼ `checkbox`: casella di spunta

Attributi:

- `name` = text: nome del controllo
- `value` = text: valore restituito se la casella viene spuntata
- `checked` = "checked": spuntato per default

▼ `radio`: radio button

Attributi:

- `name` = text: nome del controllo (gruppo di radio buttons)
- `checked` = "checked": spuntato per default

▼ `submit`: bottone per trasmettere il contenuto del form

Attributi:

- `name` = text: nome del controllo
- `value` = text: testo del bottone

▼ `image`: bottone di submit sotto forma di immagine

▼ `reset`: bottone che riporta tutti i campi al valore iniziale

Attributi:

- `name` = text: nome del controllo
- `value` = text: testo del bottone

▼ `button`: generico bottone di azione

Attributi:

- `name` = text: nome del controllo
- `value` = text: testo del bottone

▼ `hidden`: campo nascosto

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

Oltre ai tag input esistono altri tag da inserire in un form:

▼ `<button>`

In HTML 4 è stato introdotto il tag `<button>` che offre la possibilità di creare dei bottoni con un aspetto anche complesso. Infatti dà la possibilità di inserire il testo del bottone come contenuto del tag. Questo consente di specificare anche codice HTML all'interno del tag: testo formattato ma anche immagini.

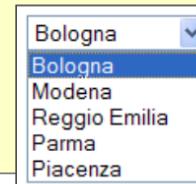
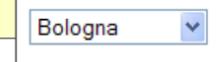
```
<form action="http://site.com/bin/adduser" method="post">
<button type="button">Generico</button>&nbsp;
<button type="reset"><i>Azzera</i></button>&nbsp;
<button type="submit"><b>Invia</b></button>
</form>
```

▼ <select>

Il tag `<select>` permette di costruire liste di opzioni. Per definire le singole opzioni si usa il tag `<option>`, e ricorrendo all'attributo `value` si può attribuire il valore. Con l'attributo `selected` si può indicare una scelta predefinita: `selected="selected"`.

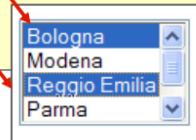
L'aspetto di default è quello di un combo box (tendina a discesa).

```
<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>
```



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. L'attributo `size` determina il numero di righe mostrate.

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



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>
```

Capoluogo
Emilia
Bologna
Modena
Reggio Emilia
Parma
Piacenza

▼ `<textarea>`

Il tag `<textarea>` consente di definire un campo di inserimento multiriga adatto a un testo lungo. Il testo iniziale che viene inserito nell'elemento è il contenuto html dello stesso. L'attributo `rows` indica il numero di righe della textarea, `cols` il numero di caratteri (cioè di colonne) che ogni riga può contenere.

▼ `<fieldset>`

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

```

<form action="http://site.com/bin/adduser" method="post">
<fieldset>
<legend>Nome e cognome</legend>
Nome: <input type="text" name="nome"><br>
Cognome: <input type="text" name="cognome">
</fieldset>
<fieldset>
<legend>Provincia</legend>
<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>
</fieldset>
</form>

```

▼ <label>

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 all'interno dell'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 frames

L'elemento `<iframe>` crea un frame inline che contiene un altro documento. È deprecato in HTML 4.01 ma è ancora molto utilizzato (in certi casi indispensabile, es. applicazioni «cross domain»)

Gli stili del testo

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.

- `<tt>` Carattere monospazio
- `<i>` Corsivo
- `` Grassetto
- `<u>` Sottolineato (deprecato)
- `<s>` Testo sbarrato (deprecato)

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

- `` Usualmente visualizzato in grassetto
- `` (emphasis) Usualmente visualizzato in corsivo
- `<code>` / `<pre>` Codice: usualmente monospaziato
- `<kbd>` Keyboard: usualmente monospaziato
- `<abbr>` Abbreviazione (nessun effetto)
- `<acronym>` Acronimo (nessun effetto)
- `<address>` Indirizzo fisico o e-mail: in corsivo
- `<blockquote>` Blocco di citazione: rientrato a destra
- `<cite>` Citazione: in corsivo

Il DOM

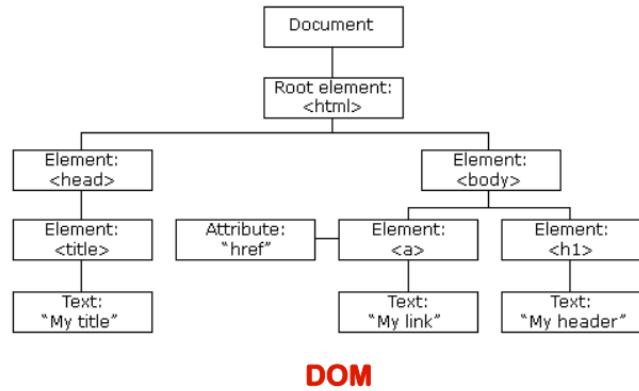
Una pagina HTML può essere rappresentata come una struttura ad albero. Questa struttura logica prende il nome di DOM: Document Object Model.

```

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

```

Testo HTML



DOM

Quando un browser riceve una pagina HTML, ne fa il parsing e costruisce la struttura ad albero del DOM.

HTML5

Molto di ciò che il linguaggio HTML 5 offre rispetto a HTML 4.01 è opzionale.

Tra le tante cose, HTML 5 estende notevolmente la possibilità di integrazione di contenuti multimediali nella pagina. Elementi come `<audio>`, `<video>`, `<canvas>`, `<math>` permettono di includere contenuti con i quali è possibile interagire in modo avanzato.

Il modello ad eventi di DOM è esteso con eventi specifici che permettono la costruzione di applicazioni sofisticate client-side.

Canvas

Una delle più importanti innovazioni di HTML 5 è la possibilità di disegnare direttamente sulla pagina e interagire con gli oggetti multimediali.

L'elemento `<canvas>` definisce un'area rettangolare in cui disegnare direttamente immagini bidimensionali e modificarle in relazione a eventi, tramite funzioni Javascript.

Audio e video

HTML 5 permette di includere video in una pagina senza richiedere plug-in esterni (Flash, Real Player, Quicktime).

L'elemento `<video>` specifica un meccanismo generico per il caricamento di file e stream video, più alcune proprietà DOM per controllarne l'esecuzione.

Ogni elemento <video> in realtà può contenere diversi elementi <source> che specificano diversi file, tra i quali il browser sceglie quello da eseguire.

```
<video id="sampleMovie" width="640" height="360" preload controls="met
  <source src="movie.mp4" type="video/mp4" codecs="avc1.42E01E, mp4;
  <source src="..." type="..." codecs="...">
  <source src="..." type="..." codecs="...">
</video>
```

L'elemento <audio> è usato allo stesso modo per i contenuti sonori.

API specifiche

HTML 5 introduce anche utili API specifiche per servizi avanzati di geolocalizzazione. Tra i metodi esposti:

- `getCurrentPosition()`, restituisce latitudine e longitudine della posizione dell'utente
- `watchPosition()`, restituisce la posizione corrente dell'utente e continua ad aggiornare la posizione dello stesso durante i suoi spostamenti
- `clearWatch()`, termina il metodo watchPosition()

▼ 5.0 - CSS

Introduzione

I fogli di stile a cascata (Cascading Style Sheets = CSS) hanno lo scopo fondamentale di separare contenuto e presentazione nelle pagine Web.

- Il linguaggio HTML serve a definire il contenuto senza tentare di dare indicazioni su come rappresentarlo.
- I CSS servono a definire come il contenuto deve essere presentato all'utente.

La parola chiave del linguaggio CSS è cascading: 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 di un documento HTML. I vantaggi della separazione di competenze sono evidenti:

- Lo stesso contenuto diventa riusabile in più contesti
- Basta cambiare i CSS e può essere presentato correttamente in modo ottimale su dispositivi diversi (es. PC e palmari) o addirittura su media diversi (es. video e carta).
- Si può dividere il lavoro fra chi gestisce il contenuto e chi si occupa della parte grafica.

Attualmente l'ultima versione di CSS è la level 3.

Il supporto dei vari browser a CSS è complesso e articolato. Infatti, i browser esistenti tendono a supportare aspetti diversi, non del tutto completi e incompatibili delle caratteristiche di CSS.

Applicare gli stili ad una pagina

Abbiamo due possibilità:

- Mettere gli stili in uno o più file separati

Sono possibili due sintassi diverse:

```
<HTML>
  <HEAD>
    <TITLE>Hello World</TITLE>
    <link rel="stylesheet"
          href="hello.css"
          type="text/css">
  </HEAD>
  <BODY>
    <H1>Hello World!</H1>
    <p>Usiamo i CSS</p>
  </BODY>
</HTML>
```

```
<HTML>
  <HEAD>
    <TITLE>Hello World</TITLE>
    <style type="text/css">
      @import url(hello.css);
    </style>
  </HEAD>
  <BODY>
    <H1>Hello World!</H1>
    <p>Usiamo i CSS</p>
  </BODY>
</HTML>
```

- Inserire gli stili nella pagina stessa

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>
    <H1>Hello World!</H1>
    <p>Usiamo i CSS</p>
  </BODY>
</HTML>

```

```

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

```

È sicuramente preferibile usare gli stili esterni, poichè è facile applicare diversi stili alla stessa pagina e si ottimizza il trasferimento delle pagine perché il foglio stile rimane nella cache del browser.

Regole

Un'espressione come `h1 {color: blue}` prende il nome di regola CSS.

Una regola CSS è composta da due parti:

- Selettore (`h1`)
- Dichiarazione (`color: blue`)

La dichiarazione a sua volta si divide in due parti:

- Proprietà (`color`)
- Valore (`blue`)

La sintassi generale è dunque la seguente:

```

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

```

Selettori

I selettori possono essere di diverse tipologie:

- Selettore universale: identifica tutti gli elementi

```
* {...}
```

- Selettore di tipo: si applica a tutti gli elementi di un determinato tipo (ad es. tutti i <p>)

```
tipo_elemento {...}
```

- Selettore di classe: si applica a tutti gli elementi che presentano l'attributo class="nome_classe"

```
.nome_classe {...}
```

- Selettore di identificatore: si applica agli 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 {...}
```

È anche possibile utilizzare le pseudoclassi, le quali si applicano ad un sottoinsieme degli elementi di un tipo identificato da una proprietà:

```
tipo_elemento:proprietà {...}
```

```
// es.
```

```
a:link {...}, a:visited {...}, h1:hover {...}
```

Gli pseudoelementi si applicano invece ad una parte di un elemento:

```
tipo_elemento:parte {...}
```

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

Infine, è possibile anche far uso di selettori gerarchici, i quali si applicano a tutti gli elementi di un dato tipo che hanno un determinato legame gerarchico (descendente, 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

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

```
h1, h2, h3 {font-family: sans-serif}
```

Valori

- I numeri possono essere interi o reali, i quali hanno “.” come separatore decimale.
- Le grandezze vengono usate per lunghezze orizzontali e verticali, e possono essere di due tipi:
 - 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

- Alcuni valori possono assumere anche la forma di percentuali, le quali rappresentano la percentuale del valore che assume la proprietà stessa nell'elemento padre; un numero seguito da %.
 - Per gli URL si usa la notazione `url(percuso)`
 - I colori possono essere identificati con tre metodi differenti:
 - Forma esadecimale `#RRGGBB`
 - Tramite la funzione `rgb(red, green, blue)`
 - Usando una serie di parole chiave che possono indicare colori assoluti o dipendenti dall'impostazione del PC (proprietà di sistema)
- Colori assoluti:

■	black - nero	■	green - verde
■	silver - argento	■	lime - verde chiaro
■	gray - grigio	■	olive - oliva
■	white - bianco	■	yellow - giallo
■	maroon - marrone	■	navy - blu scuro
■	red - rosso	■	blue - blu
■	purple - viola	■	teal - verde acqua scuro
■	fuchsia - fucsia	■	aqua - verde acqua

- Colori dipendenti dalle proprietà di sistema:

■	background - il colore di sfondo del desktop
■	buttonFace - il colore di sfondo dei pulsanti
■	buttonText - testo dei pulsanti
■	captionText - testo delle etichette
■	grayText - testo disabilitato

Attribuzione di uno stile ad un elemento

Per poter rappresentare una pagina HTML il browser deve riuscire ad applicare ad ogni elemento uno stile, in quanto un elemento privo di stile non può essere rappresentato. Per questo, anche se nella pagina non c'è nessuna

regola CSS (interna o esterna) ogni browser ha un foglio stile di default che contiene stili per ogni tag HTML.

L'attribuzione di uno stile può essere:

- Diretta: l'elemento contiene uno stile inline.
- Indiretta: l'elemento eredita lo stile dall'elemento che lo contiene.

In CSS, una proprietà applicata ad un blocco esterno si applica anche ai blocchi in esso contenuti, se questi non ridefiniscono tali proprietà. Ad esempio:



Non tutte le proprietà sono soggette al meccanismo dell'ereditarietà. In generale non vengono ereditate le proprietà che riguardano la formattazione del box model (il box è il riquadro che circonda ogni elemento).

Conflitti di stile

Nell'applicare gli stili possono nascere conflitti di competenza per diversi motivi. Lo standard CSS definisce un insieme di regole di risoluzione dei conflitti che prende il nome di cascade e che si basa su 2 elementi:

- Origine del foglio stile, che può essere di 3 tipologie:
 - 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 (es., un punteggio maggiore ad un selettore di ID rispetto ad uno di Classe)

Il CSS assegna un peso a ciascun blocco di regole, e 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 selettore: ha la precedenza il selettore con specificità maggiore
- Ordine di dichiarazione: se esistono due dichiarazioni con ugual specificità e origine vince quella fornita per ultima.

Inoltre, è possibile aggiungere al valore di una dichiarazione la clausola !important per identificare tale proprietà come precedente a tutte le altre, ad esempio:

```
p.myClass {text-color: red !important}
```

Proprietà

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

▼ Colori e sfondi

Color

Per ogni elemento si possono definire tre colori:

- `foreground-color` : il colore di primo piano
- `background-color` : il colore di sfondo
- `border-color` : il colore del bordo

La proprietà `color` definisce esclusivamente il colore di primo piano, ovvero quello del testo, e il colore del bordo di un elemento, quando non viene impostato esplicitamente con border-color.

Background

La definizione dello sfondo può essere applicata a due soli elementi: body e tavole.

- `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)

▼ Caratteri e testo

- `font-family` : font_name

I font pongono un problema di compatibilità piuttosto complesso: su piattaforme diverse (Windows, Mac, Linux...) sono disponibili caratteri diversi e ogni utente può avere un proprio set personalizzato. Per gestire questa situazione CSS mette a disposizione due meccanismi, ossia la definizione di famiglie generiche e la possibilità di dichiarare più font in una proprietà.

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)

Una dichiarazione multipla è invece fatta in questo modo:

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

Il browser procede per ordine: cerca prima il font Verdana, altrimenti cerca Helvetica e se manca anche questo ricorre all'ultimo tipo sans-

serif.

- `font-size` : size

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)
- In forma relativa
 - con le parole chiave smaller e larger
 - con l'unità em: proporzione rispetto al valore del font corrente (basato sulla M maiuscola)
 - es. 1.5em = una volta e mezzo
 - con l'unità ex: proporzione rispetto all'altezza del corpo della x minuscola del font corrente
 - in percentuale (%) rispetto al valore corrente: 75%

La scelta migliore (consigliata da W3C) è quella di esprimere in % la dimensione del testo nel body (tipicamente 100%) e poi usare gli em per gli elementi interni.

```
body {font-size:100%}  
h1 {font-size:2.5em}  
p {font-size:0.875em}
```

- `font-weight` : weight

Il peso si può esprimere in diversi modi:

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

- `font-style` : style

- 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:

selettori {font: font-style font-variant font-weight font-size font-family}

- **line-height** : height

Permette di definire l'altezza di una riga di testo all'interno di un elemento blocco. In pratica consente di definire l'interlinea (lo spazio fra le righe) usando i seguenti valori:

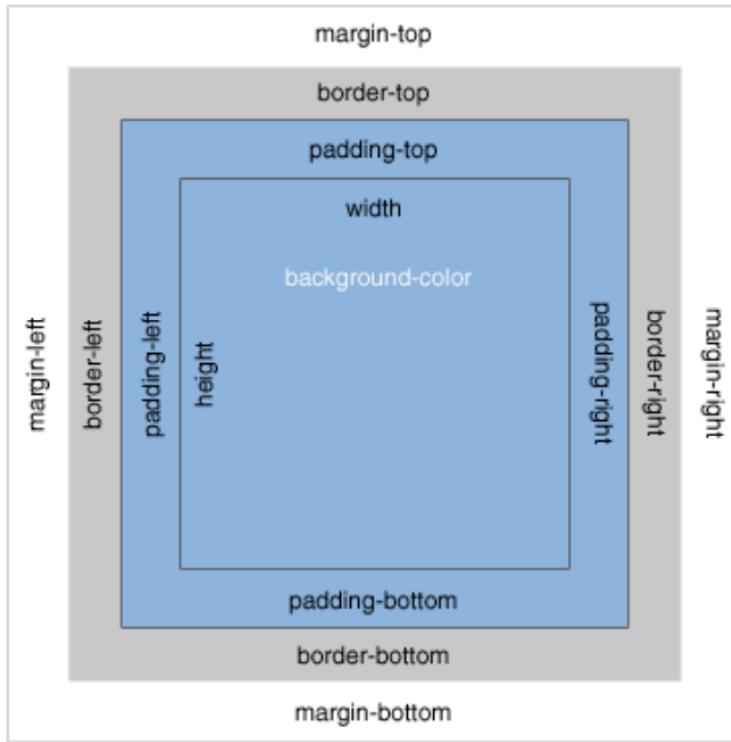
- normal: spaziatura di default stabilita dal browser
- valore numerico: moltiplicatore applicato al corpo del carattere (es. 1.5= una volta e mezzo il corpo, di solito è la scelta migliore)
- valore con unità di misura: altezza esatta della riga
- percentuale: altezza riga pari a una % del corpo
- **text-align** : {left|right|center|justify}
- **text-decoration** : {none|underline|overline|line-through}
- **text-indent** : indent

Definisce l'indentazione della prima riga in ogni elemento contenente del testo. Può essere espressa in due modi:

- valore numerico con unità di misura
- valore in percentuale rispetto alla larghezza del blocco di testo
- **text-transform** : {none|capitalize|uppercase|lowercase}

▼ Box model

Il box model comprende cinque elementi base rappresentati nella seguente figura



Se due elementi sono allineati verticalmente si ha il margin collapsing, quindi la distanza è pari al massimo fra il margine inferiore del primo e il margine superiore del secondo.

Per quanto riguarda le dimensioni del contenuto si hanno le seguenti proprietà:

- `height`
- `min-height/max-height`
- `width`
- `min-width/max-width`

Valori ammessi:

- `auto`: dimensione determinata dal contenuto (solo per width e height)
- 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. I valori che può assumere sono i seguenti:

- `visible` (default): il contenuto eccedente viene mostrato
- `hidden`: il contenuto eccedente non viene mostrato
- `scroll`: vengono mostrate barre di scorrimento che consentono di accedere al contenuto eccedente
- `auto`: il browser tratta il contenuto eccedente secondo le sue impostazioni predefinite (di solito barre di scorrimento)

Per i margini è possibile utilizzare le proprietà singole `margin-top`, `margin-right`, `margin-bottom`, `margin-left`, oppure la proprietà sintetica `margin` con i valori nell'ordine delle proprietà singole.

Per il padding si può utilizzare le proprietà singole `padding-top`, `padding-right`, `padding-bottom`, `margin-left` oppure la proprietà sintetica `padding`.

Per ciascuno dei 4 bordi è possibile definire stile, colore e spessore, ad esempio per il bordo superiore si ha `border-top-color`, `border-top-style` e `border-top-width`. Altrimenti è possibile farlo utilizzando la proprietà sintetica `border-top`. Per lo stile dei bordi è possibile utilizzare i seguenti valori: `none`, `hidden`, `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset`, `outset`. Per lo spessore, oltre ai valori numerici con unità di misura è possibile utilizzare `thin`, `medium` e `thick`.

▼ Liste

In virtù dell'ereditarietà, se applichiamo una proprietà alle liste la applichiamo a tutti gli elementi.

- `list-style-image` : {url/none}

Definisce l'immagine da utilizzare come "punto elenco".

- `list-style-position` : {inside/outside}

Indice la posizione del "punto elenco". `Inside` se il punto fa parte del testo, `outside` altrimenti.

- `list-style-type` : {none/disc/circle/square/decimal/decimal-leading-zero/lower-roman/upper-roman/lower-alpha/lower-latin/upper-alpha/upper-latin/lower-greek}

Indica l'aspetto del punto elenco.

▼ Display e gestione degli elementi floating

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`, `block`, `list-item` e `none` (l'elemento viene trattato come se non ci fosse, dunque non viene mostrato e non genera alcun box).

La proprietà `float` consente di estrarre un elemento dal normale flusso del documento e spostarlo su uno dei lati (destro o sinistro) del suo elemento contenitore. Il contenuto che circonda l'elemento scorre intorno ad esso sul lato opposto rispetto a quello indicato come valore di float. I valori che può assumere questa proprietà sono `left`, `right` e `none`.

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 compaiano altri elementi. Si applica solo agli elementi blocco e non è ereditata. I valori che può assumere sono `none`, `left`, `right` e `both`.

▼ 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 (al limite <html>).
- `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`

Coordinate del posizionamento. I valori possono essere assoluti e relativi.

- `visibility : {visible/hidden}`
- `z-index`

CSS gestisce gli elementi come se fossero fogli di carta e questa proprietà permette di stabilire quale sta sopra e quale sta sotto. I valori ammessi sono auto o un valore numerico (più è alto e più l'elemento è in cima al "mucchio di fogli").

▼ Tabelle

- `table-layout`

Ammette 2 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`, che ammette come valore un numero con unità di misura.

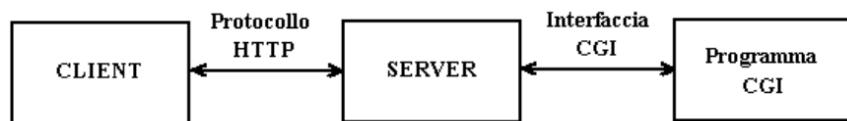
▼ 6.0 - Web dinamico

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. Le pagine vengono infatti preparate staticamente a priori e non esistono contenuti composti dinamicamente in

base all'interazione con l'utente. È un modello semplice e efficiente, ma presenta evidenti limiti.

CGI

La prima soluzione proposta per risolvere questo problema prende il nome di Common Gateway Interface (CGI). CGI è uno standard per interfacciare applicazioni esterne con Web server. Un programma CGI, il quale può essere scritto in qualunque linguaggio, viene eseguito dinamicamente in risposta alla chiamata e produce output che costituisce la risposta alla richiesta http.



Le operazioni si svolgono nel seguente ordine:

- Il client, tramite HTTP, invia al server la richiesta di eseguire un programma CGI con alcuni parametri e dati in ingresso.
- Il server, attraverso l'interfaccia standard CGI, chiama il programma passandogli i parametri e i dati inviati dal client.
- Eseguite le operazioni necessarie, il programma CGI rimanda al server i dati elaborati (pagina HTML), sempre facendo uso dell'interfaccia CGI.
- Il server invia al client i dati elaborati dal programma CGI tramite protocollo HTTP.

I programmi CGI e il server comunicano in quattro modi:

- Variabili di ambiente del sistema operativo.
- Parametri sulla linea di comandi: il programma CGI viene lanciato in un processo pesante.
- Standard Input (usato con il metodo POST).
- Standard Output: per restituire al server la pagina HTML da inviare al client.

Se arriva un certo URL che chiede l'esecuzione di un programma CGI, il server deve rendersi conto che tale URL non rappresenta un documento HTML ma un

programma CGI.

Perché ciò accada è necessario che i programmi CGI siano tutti in un'apposita directory, che solitamente viene chiamata cgi-bin.

Problemi dei programmi CGI

L'architettura che abbiamo appena visto presenta numerosi vantaggi ma soffre anche di diversi problemi:

- Ci sono problemi di prestazioni, infatti 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 (mancanza di moduli di base accessibili a tutti i programmi lato server).

La soluzione migliore per questo problema è quella di realizzare un contenitore in cui inserire tutte le funzioni server-side. Tale contenitore si preoccupa di fornire i servizi di cui le applicazioni hanno bisogno (interfacciamento con il Web Server, gestione del tempo di vita, interfacciamento con il database e gestione della sicurezza). Un ambiente di questo tipo prende il nome di application server.

Alcune tecnologie molto diffuse nell'ambito degli application server sono per esempio .NET, Java J2EE e node.js.

- Ci sono scarse garanzie sulla sicurezza.

Stato

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

- Stateful: esiste stato dell'interazione e quindi l'nesimo messaggio può essere messo in relazione con gli n-1 precedenti.
- Stateless: non si tiene traccia dello stato, ogni messaggio è indipendente dagli altri.

In termini generali, un'interazione stateless non genera grossi problemi solo se protocollo applicativo è progettato con operazioni idempotenti, ovvero

operazioni che producono sempre lo stesso risultato, indipendentemente dal numero di messaggi ricevuti dal server stesso.

Non tutte le applicazioni, però, possono fare a meno dello stato. In generale, tutte le volte in cui abbiamo bisogno di personalizzazione delle richieste Web, possiamo beneficiare di interazione stateful.

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

- Stato di esecuzione: insieme dei dati parziali per una elaborazione.
- Stato di sessione: insieme dei dati che caratterizzano una interazione con uno specifico utente.
- Stato informativo persistente: viene normalmente mantenuto in una struttura persistente come un database (es. gli ordini inseriti da un sistema di eCommerce).

La sessione rappresenta lo stato associato ad una sequenza di pagine visualizzate da un utente. Lo scope di sessione è dato dal tempo di vita della interazione utente (lifespan) e dall'accessibilità (solitamente concessa alla richiesta corrente e a tutte quelle successive proveniente dallo stesso processo browser).

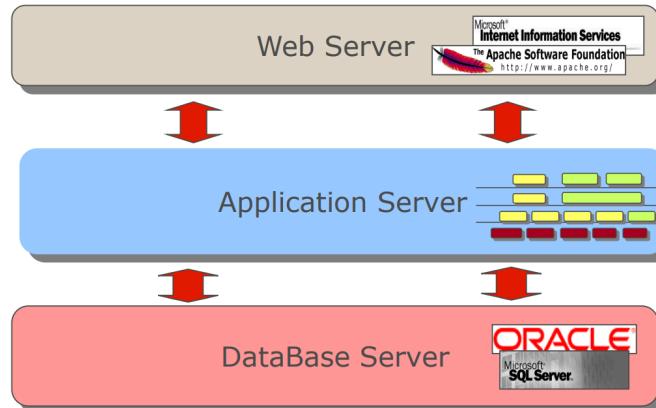
La conversazione rappresenta una sequenza di pagine di senso compiuto. A differenza della sessione, nell'esempio dell'acquisto di un prodotto, la conversazione comprende solamente un acquisto singolo, mentre la sessione rimane attiva al termine di questo, consentendo all'utente della sessione di effettuarne un altro.

Ci sono due tecniche di base per gestire lo stato, non necessariamente alternative ma integrabili:

- Utilizzo del meccanismo dei cookie (storage lato cliente)
- Gestione di uno stato sul server per ogni utente collegato (sessione server-side)

Architettura dei sistemi web

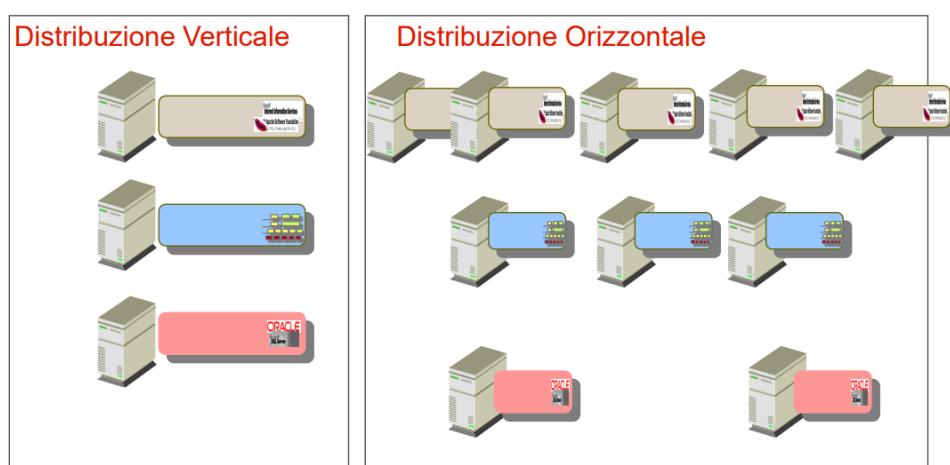
Un architettura frequente nei sistemi web è l'architettura a 3 tier.



Distribuzione verticale e orizzontale

Questi 3 servizi possono risiedere sullo stesso HW oppure essere divisi su macchine separate, e in tal caso si parla di distribuzione verticale dell'architettura.

Inoltre è possibile replicare ad ogni livello il servizio su diverse macchine. Si parla in questo caso di distribuzione orizzontale.



Web server è stateless per la natura del protocollo HTTP, per questo in genere è molto facile da replicare. Può accadere però che application server utilizzi oggetti o componenti con stato. In tal caso vengono in soccorso alcuni framework in grado di effettuare replicazione tramite tecniche di clustering, oppure è possibile gestire la sessione tramite cookie, rendendo il lato server stateless.

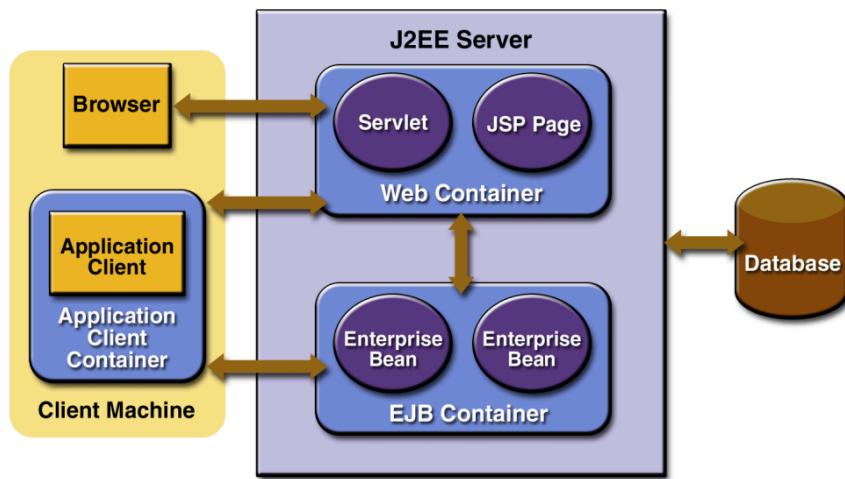
Il database server è normalmente di tipo stateful. La replicazione è quindi molto delicata perché deve mantenere il principio di atomicità delle transazioni. I database commerciali, come Oracle e Microsoft SQL Server prevedono delle configurazioni di clustering.

▼ 7.0 - Servlet

L'architettura Java Enterprise Edition (JEE o J2EE)

L'architettura Java Enterprise Edition è un modello a Componente-Container, ovvero che separa le applicazioni in componenti distinti, ognuno dei quali è gestito da un contenitore (container). Tale approccio facilita lo sviluppo di applicazioni distribuite e scalabili, migliorando la manutenibilità e l'efficienza complessiva del sistema.

Inoltre è un architettura multi-tier, ovvero è suddivisa in più livelli, ognuno dei quali ha responsabilità specifiche.



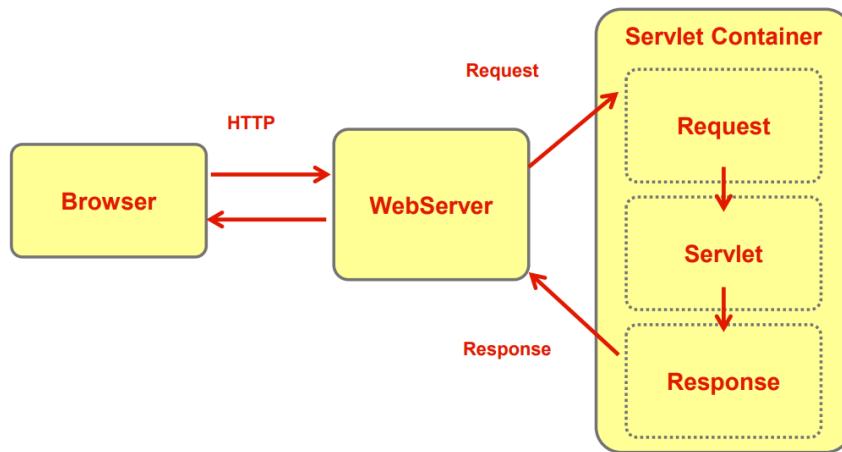
I Web Client sono spesso costituiti dal semplice browser Web senza bisogno di alcuna installazione ad hoc. Comunicano via HTTP e HTTPS con il server ed effettuano il rendering della pagina in HTML.

Una Web Application è un gruppo di risorse server-side che nel loro insieme creano una applicazione interattiva fruibile via Web. I Web Container forniscono un ambiente di esecuzione per Web Application, garantendo servizi di base alle applicazioni sviluppate secondo un paradigma a componenti.

Servlet

Le servlet sono classi Java che elaborano richieste seguendo un protocollo condiviso di tipo request/response. Le servlet HTTP sono il tipo più comune di servlet e possono processare richieste HTTP, producendo response HTTP.

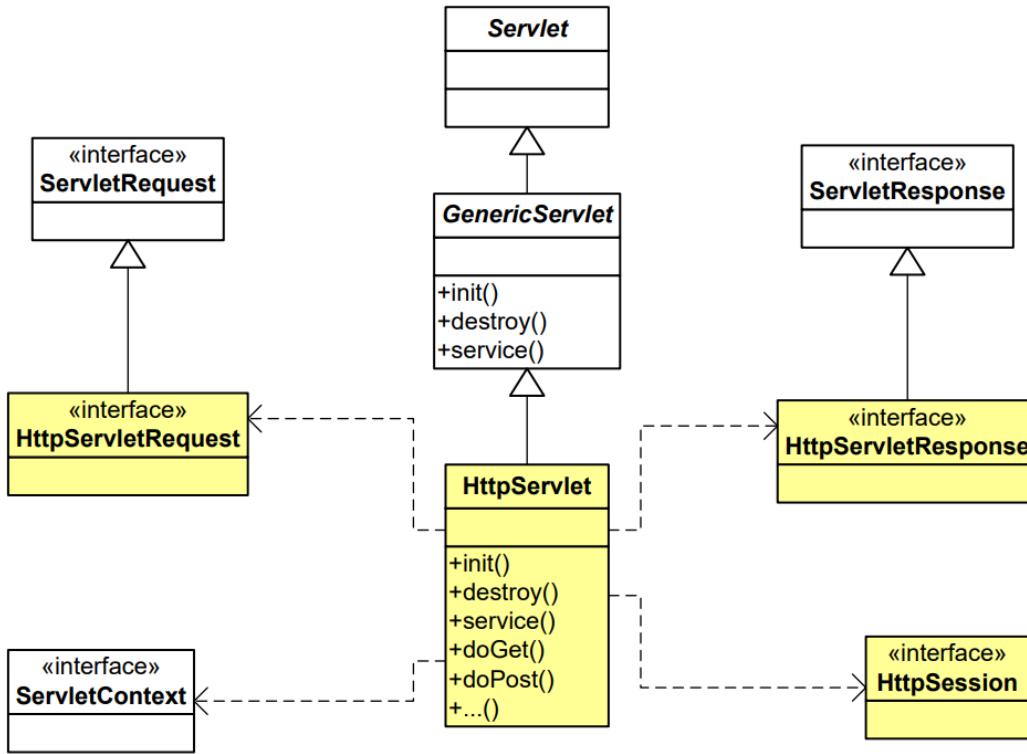
All'arrivo di una richiesta HTTP il Servlet Container crea un oggetto request e un oggetto response e li passa alla servlet.



Gli oggetti di tipo Request rappresentano la chiamata al server effettuata dal client e contengono varie informazioni, come chi ha effettuato la request, quali parametri e quali header sono stati passati.

Gli oggetti di tipo Response rappresentano le informazioni restituite al client in risposta ad una Request. Presentano un header e dei dati in forma testuale (es. html, text) o binaria (es. immagini).

Di seguito vi è un insieme di classi e interfacce per Servlet:



Il ciclo di vita delle Servlet

Il 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 e inizializza la servlet invocando 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`.

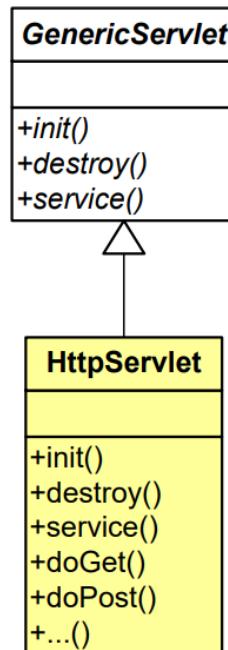
Nel modello “normale” vi è una sola istanza di servlet e un thread assegnato ad ogni richiesta http. Più thread condividono quindi la stessa istanza di una servlet e quindi si crea una situazione di concorrenza: il metodo `init()` della servlet viene chiamato una sola volta quando la servlet è caricata dal Web container, ma il metodo `service` (e dunque `doGet` e `doPost`) può essere invocato da numerosi client in modo concorrente ed è quindi necessario gestire le sezioni critiche (tramite uso di blocchi `synchronized`, semafori o mutex).

Alternativamente si può indicare al container di creare un'istanza della servlet per ogni richiesta concorrente. Questa modalità prende il nome di Single-Threaded Model ed è onerosa in termine di risorse, infatti è deprecata nelle specifiche 2.4 delle servlet. Se una servlet vuole operare in modo single-threaded deve implementare l'interfaccia marker SingleThreadModel.

Metodi per il controllo del ciclo di vita

- `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 (es. quando è rimossa). Tipicamente serve per rilasciare le risorse acquisite (es. connessione a db, eliminazione di variabili di stato per l'intera applicazione, ...).

I metodi `init()`, `destroy()` e `service()` sono definiti nella classe astratta `GenericServlet`.



L'oggetto request

Request contiene i dati inviati dal client HTTP al server. Fornisce metodi per accedere a varie informazioni:

- Request URL

Una URL HTTP ha la sintassi:

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

La query string è composta da un insieme di parametri che sono forniti dall'utente.

`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 query string.

`String getPathInfo()` per ottenere il path.

`String getPathTranslated()` per ottenere informazioni sul path nella forma risolta.

- Request 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.

`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.

- Request body

`InputStream getInputStream()` consente di leggere il body della richiesta, ad esempio:

```
public void doPost(HttpServletRequest request, HttpServletResponse res  
throws ServletException, IOException {
```

```

InputStream is = request.getInputStream();
BufferedReader in = new BufferedReader(new InputStreamReader(is));

PrintWriter out = response.getWriter();
out.println("<html>\n<body>");
out.println("Contenuto del body del pacchetto: ");
while ((String line = in.readLine()) != null)
    out.println(line)
out.println("</body>\n</html>");
}

```

- Tipo di autenticazione e informazioni su utente

`String getRemoteUser()` nome di user se la servlet ha accesso autenticato, null altrimenti.

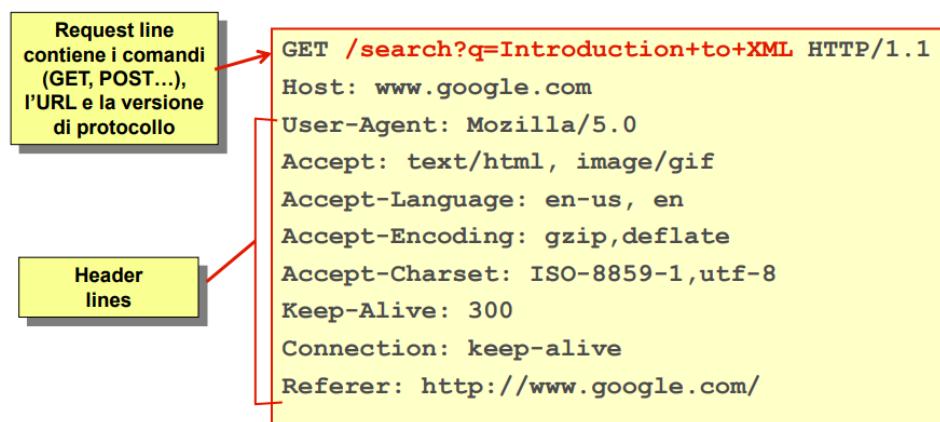
`String getAuthType()` nome dello schema di autenticazione usato per proteggere la servlet.

`boolean isUserInRole(java.lang.String role)` restituisce true se l'utente è associato al ruolo specificato.

- Cookie

`Cookie[] getCookies()` restituisce un array di oggetti cookie che il client ha inviato alla request.

- Session



Per quanto riguarda le richieste di tipo GET i parametri vengono inseriti all'interno della query string dell'URL, mentre per il tipo POST, ad esempio tramite un form, vengono dichiarati i campi utilizzando l'attributo name, il quale poi verrà utilizzato per invocare il metodo getParameter().

L'oggetto response

Contiene i dati restituiti dalla Servlet al Client:

- Status line

Per definire lo status code HttpServletResponse fornisce il metodo:

```
public void setStatus(int statusCode)
```

Per inviare errori possiamo anche usare:

```
public void sendError(int statusCode)
```

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

- Header

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

`public void setDateHeader(String headerName, long millisecs)` imposta la data.

`public void setIntHeader(String headerName, 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.

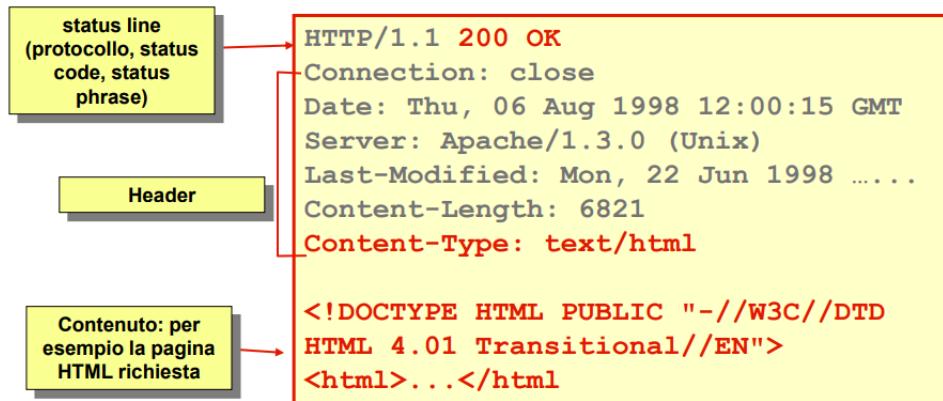
`sendRedirect()` imposta location header e cambia lo status code in modo da forzare una ridirezione.

- Body

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

- `public PrintWriter getWriter()` : mette a disposizione uno stream di caratteri (utile per restituire un testo nella risposta, tipicamente HTML).

- o `public ServletOutputStream getOuputStream()` : mette a disposizione uno stream di byte (più utile per una risposta con contenuto binario, ad esempio un'immagine).



Deployment

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, ad es. welcome page e pagine di errore.
- La configurazione delle caratteristiche di sicurezza dell'applicazione.

Web Archives

Gli Archivi Web (Web Archives) sono file con estensione ".war". Sono file jar con una struttura particolare. Per crearli si usa il comando jar:

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

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



Web.xml

web.xml è un file di configurazione che descrive la struttura dell'applicazione Web, permettendo di definire la mappatura tra URL e servlet. Contiene l'elenco delle servlet e per ogni servlet permette di definire:

- nome
- classe Java corrispondente
- una serie di parametri di configurazione (coppie nome-valore, valori di inizializzazione)

Un esempio di mappatura è il seguente:

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

L'URL che viene mappato a myServlet è dunque il seguente:

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

È possibile accedere ai parametri di configurazione (specificati in web.xml) di una servlet 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.

Una volta ottenuta l'istanza `ServletConfig`, è possibile richiamare su di essa il seguente metodo per ottenere i valori dei parametri di configurazione:

```
String getInitParameter(String parName)
```

ServletContext

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()`.

IMPORTANTE: servlet context viene condiviso tra tutti gli utenti, tutte le richieste e tutti i componenti server-side (servlet, ...) della stessa Web application.

Parametri di inizializzazione del contesto

I parametri di inizializzazione del contesto sono definiti all'interno di elementi di tipo context-param in web.xml.

```
<web-app>
  <context-param>
    <param-name>name</param-name>
    <param-value>value</param-value>
  </context-param> ...
</web-app>
```

Sono accessibili in lettura a tutte le servlet della Web application tramite il metodo `getInitParameter()` della classe ServletContext.

Attributi di contesto

Gli attributi di contesto funzionano come variabili globali, e possono anche contenere oggetti complessi, tramite la serializzazione e la deserializzazione.

È possibile scrivere, leggere e rimuovere attributi di contesto tramite i metodi `setAttribute()`, `getAttribute()` e `removeAttribute()` della classe ServletContext, esempio:

```
// scrittura
ServletContext ctx = getServletContext();
ctx.setAttribute("utente1", new User("Giorgio Bianchi"));
ctx.setAttribute("utente2", new User("Paolo Rossi"));

// lettura
ServletContext ctx = getServletContext();
Enumeration aNames = ctx.getAttributeNames();
while (aNames.hasMoreElements()) {
    String aName = (String) aNames.nextElement();
    User user = (User) ctx.getAttribute(aName);
    ctx.removeAttribute(aName);
}
```

Gestione dello stato (di sessione)

Come abbiamo già detto più volte, HTTP è un protocollo stateless.

Le applicazioni Web hanno però bisogno di stato, per questo sono state definite due tecniche per mantenere traccia delle informazioni riguardanti questo:

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

La sessione può far ricorso a due meccanismi base di implementazione:

- Cookie
- URL rewriting

Cookie

Il cookie è un'unità di informazione che il Web server deposita sul Web browser lato cliente, il quale li memorizza. Sono parte dell'header HTTP, trasferiti in formato testuale, e vengono mandati avanti e indietro nelle richieste e nelle risposte.

Nell'utilizzo dei cookie bisogna però fare attenzione al fatto che possono essere rifiutati dal browser, tipicamente perché disabilitati, e sono spesso considerati un fattore di rischio.

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.

I cookie si recuperano dalla request utilizzando il metodo `getCookies()`, il quale restituisce un'istanza della classe `Cookie`. Si aggiungono invece alla response utilizzando il metodo `addCookie()`. Un esempio di utilizzo è il seguente:

```
// creazione
Cookie c = new Cookie("MyCookie", "test");
c.setSecure(true); // il client viene forzato a inviare il cookie solo su protocollo
```

```

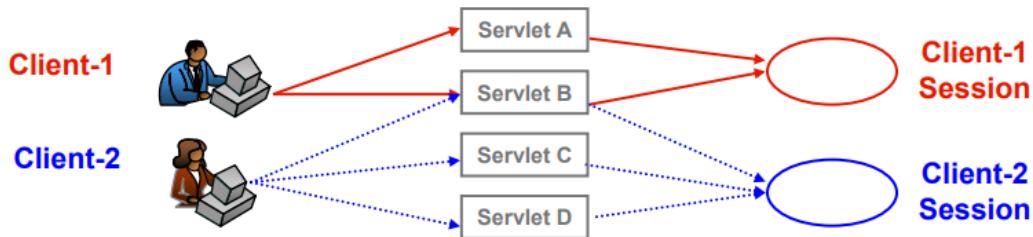
c.setMaxAge(-1);
c.setPath("/");
response.addCookie(c);

// 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());
    }
}

```

Sessione

La sessione Web è un'entità gestita dal Web container, la quale è condivisa fra tutte le richieste provenienti dallo stesso client, quindi consente di mantenere informazioni di stato (di sessione). Può contenere dati di varia natura ed è identificata in modo univoco da un session ID.



L'accesso alla sessione avviene mediante l'interfaccia HttpSession. Per ottenere un riferimento ad un oggetto di tipo HttpSession si usa il seguente metodo dell'interfaccia HttpServletRequest:

```
public HttpSession getSession(boolean createNew)
```

Il parametro `createNew` ha il seguente significato:

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

Si possono memorizzare dati specifici dell'utente negli attributi della sessione (coppie nome/valore), sempre tramite i metodi `getAttribute()`, `getAttributeNames()`, `setAttribute()` e `removeAttribute()`.

Altre operazioni che si possono effettuare su un oggetto HttpSession sono le seguenti:

- `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.

Il session ID è usato per identificare le richieste provenienti dallo stesso utente e mapparle sulla corrispondente sessione. Per trasmettere il session ID è possibile utilizzare due tecniche:

- Includerlo in un cookie.
- Includerlo nella URL (URL rewriting). In questo caso è buona prassi codificare sempre le URL generate dalle servlet usando il metodo `encodeURL()` di `HttpServletResponse`.

Scope degli oggetti della servlet

Gli oggetti di tipo `ServletContext`, `HttpSession`, `HttpServletRequest` forniscono metodi per immagazzinare e ritrovare oggetti nei loro rispettivi scope, definiti dal tempo di vita e dall'accessibilità da parte della servlet.

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

Riassumendo, tutti questi oggetti scoped forniscono i seguenti metodi per immagazzinare e ritrovare oggetti nei loro rispettivi scope:

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

Inclusione/Inoltro di risorse Web

Includere risorse Web (altre pagine, statiche o dinamiche) può essere utile quando si vogliono aggiungere contenuti creati da un'altra risorsa (ad es. un'altra servlet). Possono essere effettuate due tipologie di inclusioni:

- Inclusione di risorsa statica: includiamo un'altra pagina nella nostra (ad es. banner)
- Inclusione di risorsa dinamica: la servlet inoltra una request ad un componente Web che la elabora e restituisce il risultato. Il risultato viene poi incluso nella pagina prodotta dalla servlet.

Come si fa l'inclusione

Per includere una risorsa si ricorre a un oggetto di tipo `RequestDispatcher` che può essere richiesto al contesto indicando la risorsa da includere. 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);
```

Inoltro (forward)

L'inoltro, o forward, si usa in situazioni in cui una servlet si occupa di parte dell'elaborazione della richiesta e delega a qualcun altro la gestione della risposta. In questo caso la risposta è di competenza esclusiva della risorsa che riceve l'inoltro, infatti se nella prima servlet è stato fatto un accesso a ServletOutputStream o PrintWriter si ottiene una IllegalStateException.

Anche in questo caso si deve ottenere un oggetto di tipo RequestDispatcher da request passando come parametro il nome della risorsa. Si invoca quindi il metodo forward passando anche in questo caso request e response.

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

Ridirezione del browser

È anche possibile inviare al browser una risposta che lo forza ad accedere ad un'altra pagina (ridirezione). Si usa uno dei codici di stato di HTTP, sono i codici che vanno da 300 a 399 (in particolare 301 Moved permanently: URL non valida, il server indica la nuova posizione).

Possiamo ottenere questo risultato in due modi, agendo sempre sull'oggetto response:

- Invocando il metodo `public void sendRedirect(String url)`
- Lavorando più a basso livello con gli header:

```
response.setStatus(response.SC_MOVED_PERMANENTLY);  
response.setHeader("Location", "http://...");
```

▼ 8.0 - JSP

▼ 8.1 - Introduzione e tag JSP

Introduzione alle JSP

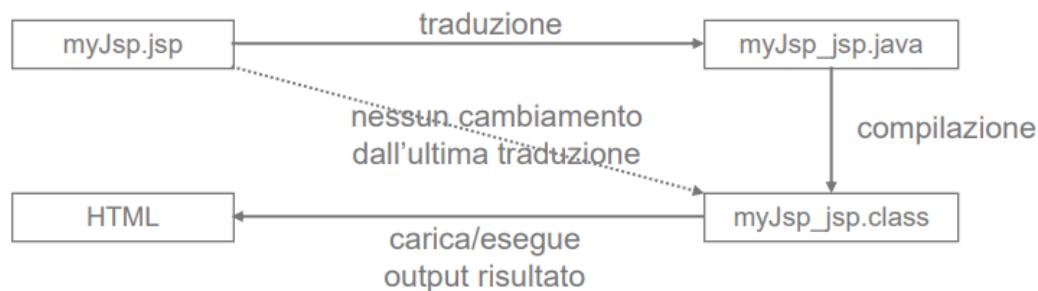
Le JSP (Java Server Pages) sono, insieme alle Servlet, uno dei due componenti di base della tecnologia J2EE, relativamente alla parte Web. In sintesi estendono HTML con codice Java custom, e quando viene effettuata una richiesta a una JSP la parte HTML viene direttamente trascritta sullo stream di output, mentre il codice Java viene eseguito sul server per la generazione del contenuto HTML dinamico.

JspServlet

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

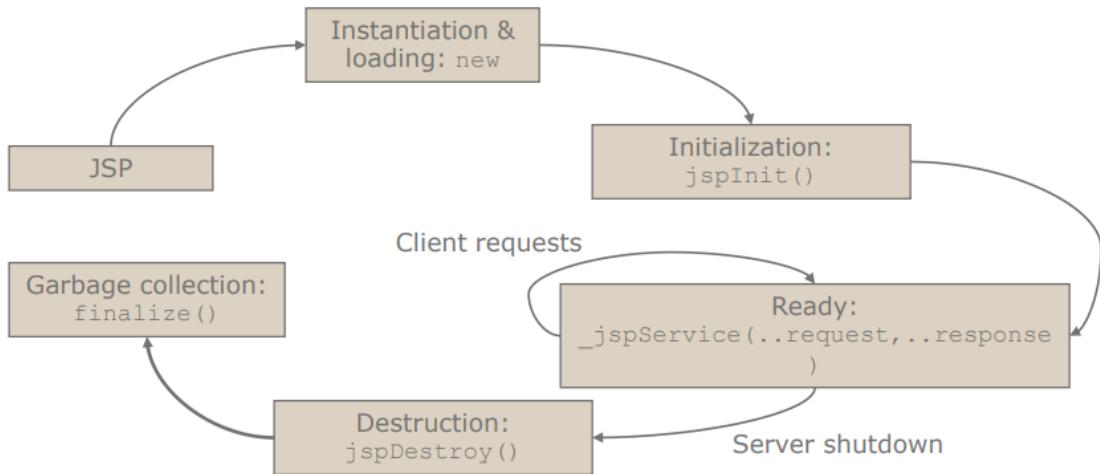
- Traduzione della JSP in una servlet.
- Compilazione della servlet risultante in una classe.
- Esecuzione della JSP.

I primi due passi vengono eseguiti solo quando cambia il codice della JSP.



Ciclo di vita delle JSP

Dal momento che JSP sono compilate in servlet, il ciclo di vita delle JSP, dopo compilazione, è controllato sempre dal medesimo Web container.



Perchè usare JSP

JSP nascono per facilitare la progettazione grafica e l'aggiornamento delle pagine.

Le JSP non rendono però inutili le servlet. Queste infatti forniscono agli sviluppatori delle applicazioni Web un completo controllo dell'applicazione. Ad esempio 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.

JSP rendono viceversa molto semplice presentare documenti HTML o XML (o loro parti) all'utente. Aiutano dunque nella realizzazione di pagine dinamiche semplici e di uso frequente. Come in tutti i linguaggi di script che poi generano codice, portano però maggiori problemi di controllo della correttezza e testing.

Tag

Le parti variabili della pagina sono contenute all'interno di tag speciali.

Sono possibili due tipi di sintassi per questi tag:

- Scripting-oriented tag

Sono definite da delimitatori entro cui è presente lo scripting, e sono di quattro tipi:

- `<%! %>` Dichiarazione.

- `<%= %>` Espressione.
- `<% %>` Scriptlet.
- `<%@ %>` Direttiva.
- XML-Oriented tag

Seguono la sintassi XML, e sono i seguenti:

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

Dichiarazioni

Si usano i delimitatori `<%!` e `%>` per dichiarare variabili e metodi che possono poi essere referenziati in qualsiasi punto del codice JSP. I metodi diventano metodi della servlet quando la pagina viene tradotta.

Esempio:

```
<%!
    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

Si usano i delimitatori `<%=` e `%>` per valutare espressioni Java. Risultato dell'espressione viene convertito in stringa inserito nella pagina al posto del tag.

Esempio:

```
// 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

Si usano i delimitatori `<%` e `%>` per aggiungere un frammento di codice Java eseguibile alla JSP (scriptlet). Lo scriptlet consente tipicamente di inserire logiche di controllo di flusso nella produzione della pagina. La combinazione di tutti gli scriptlet in una determinata JSP 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

Si usano i delimitatori `<%@` e `%>` per inserire comandi JSP da essere valutati a tempo di compilazione e non producono nessun output visibile. 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.

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

La direttiva page definisce una serie di attributi che si applicano all'intera pagina. Gli attributi di page sono i seguenti:

- `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.
- `buffer="none|8kb|sizekb"` dimensione in KB del buffer di uscita.
- `autoFlush="true|false"` dice se il buffer viene svuotato automaticamente quando è pieno. Se il valore è false viene generata un'eccezione quando il buffer è pieno.
- `isThreadSafe="true|false"` indica se il codice contenuto nella pagina è thread-safe. Se vale false le chiamate alla JSP vengono serializzate.
- `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.

```
<%@ 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" ]  
%>
```

La direttiva include serve ad includere il contenuto del file specificato. La sintassi è

```
<%@ include file="localURL" %> .
```

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

▼ 8.2 - Built-in objects

Built-in objects

Le specifiche JSP definiscono 9 oggetti built-in utilizzabili senza dover creare istanze. Rappresentano utili riferimenti ai corrispondenti oggetti Java veri e propri presenti nella tecnologia servlet. Sono i seguenti:

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

Oggetto page

L'oggetto page rappresenta l'istanza corrente della servlet. Ha come tipo l'interfaccia `HTTPJspPage` che discende da `JspPage`, la quale a sua volta estende `Servlet`. Può quindi essere utilizzato per accedere a tutti i metodi definiti nelle servlet.

Oggetto config

Contiene la configurazione della servlet (parametri di inizializzazione).

Metodi di config:

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

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 di provenienza, URL, headers, cookie, parametri, ecc.). I metodi disponibili sono gli stessi visti per le servlet.

Oggetto response

Oggetto 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). I metodi disponibili sono gli stessi visti per le servlet.

Oggetto out

Oggetto legato all'I/O della pagina JSP. È uno stream di caratteri e rappresenta lo stream di output della pagina.

I metodi dell'oggetto out sono:

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

Oggetto session

Oggetto che 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). I metodi disponibili sono gli stessi visti per le servlet.

Oggetto application

Oggetto che 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: fornisce la versione di JSP Container, garantisce l'accesso a risorse server-side, permette accesso ai parametri di inizializzazione relativi all'applicazione, consente di gestire gli attributi di un'applicazione.

Oggetto pageContext

Oggetto che fornisce informazioni sul contesto di esecuzione della pagina JSP. Rappresenta l'insieme degli oggetti built-in di una JSP, quindi

consente l'accesso a tutti gli oggetti impliciti e il trasferimento del controllo ad altre pagine.

Oggetto exception

Oggetto connesso alla gestione degli errori. Non è automaticamente disponibile in tutte le pagine ma solo nelle Error Page (quelle dichiarate con l'attributo `errorPage` impostato a true).

Esempio:

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

▼ 8.2 - Azioni

Azioni

Le azioni sono comandi JSP tipicamente per l'interazione con altre pagine JSP, servlet, o componenti JavaBean. Sono previsti 6 tipi di azioni definite dai seguenti tag:

- `useBean` : istanzia JavaBean e gli associa un identificativo.
- `getProperty` : ritorna una proprietà indicata come oggetto.
- `setProperty` : imposta valore di una proprietà 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.

Esempio:

```

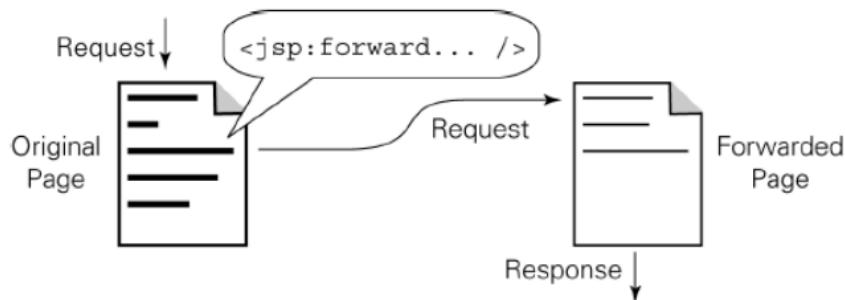
<html>
  <body>
    <jsp:useBean id="myBean" class="it.unibo.deis.my.HelloBean"/>
    <jsp:setProperty name="myBean" property="nameProp" param="na
    <jsp:setProperty name="myBean" property="nameProp" value="val
    Hello, <jsp:getProperty name="myBean" property="nameProp"/>!
  </body>
</html>

```

Azioni: forward

Sintassi: `<jsp:forward page="localURL" />` .

Consente trasferimento del controllo dalla pagina JSP corrente ad un'altra pagina sul server locale. Oggetti request, response e session della pagina d'arrivo sono gli stessi della pagina chiamante, ma viene istanziato un nuovo oggetto pageContext.



È anche possibile generare dinamicamente attributo page

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

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"/>
  ...

```

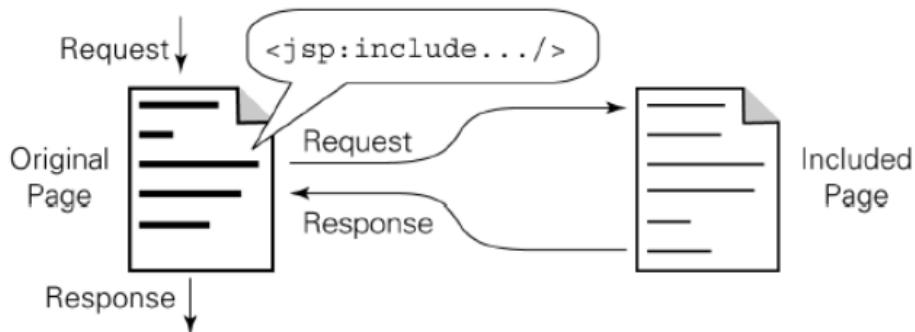
```
<jsp:param name="parNameN" value="parValueN"/>  
</jsp:forward>
```

Nota: forward è possibile soltanto se non è stato emesso alcun output.

Azioni: include

Sintassi: `<jsp:include page="localURL" flush="true" />`.

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



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

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

▼ 8.2 - Modello a componenti

JSP e 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.

Per questo JSP consente anche uno sviluppo basato su un modello a componenti.

JavaBeans

JavaBeans è il modello di "base" per componenti Java. Un JavaBean è una classe Java dotata di alcune caratteristiche particolari:

- Classe public.
- Ha un costruttore public di default.
- Espone proprietà tramite getter e setter. Tutti i tipi di proprietà devono avere metodi setter e getter chiamati `getProp()` e `setProp()`. Questo non vale per i booleani, i quali per la lettura presentano metodi chiamati `isProp()`.
- Espone eventi con metodi di registrazione che seguono regole precise.

Componenti e container

I componenti vivono all'interno di contenitori che gestiscono il tempo di vita dei singoli componenti e i collegamenti fra i componenti e resto del sistema.

Un contenitore per JavaBean prende il nome di bean container, ed è in grado di interfacciarsi con i bean utilizzando Java Reflection, che fornisce strumenti di introspezione e di dispatching.

JSP e JavaBeans

JSP prevede una serie di tag per agganciare un bean e utilizzare le sue proprietà all'interno della pagina. Questi si dividono in 3 tipi:

- Tag per creare un riferimento al bean (creazione di un'istanza).
- Tag per impostare il valore delle proprietà del bean.
- Tag per leggere il valore delle proprietà del bean e inserirlo nel flusso della pagina.

Esempio:

```

<jsp:useBean id="user" class="RegisteredUser" scope="session"/>
<jsp:useBean id="news" class="NewsReports" scope="request">
    <jsp:setProperty name="news" property="category" value="fin."/>
    <jsp:setProperty name="news" property="maxItems" value="5"/>
</jsp:useBean>

<html>
    <body>
        <p>Bentornato
            <jsp:getProperty name="user" property="fullName"/>,
            la tua ultima visita è stata il
            <jsp:getProperty name="user" property="lastVisitDate"/>.
        </p>
        <p>
            Ci sono <jsp:getProperty name="news" property="newItems"/>
            nuove notizie da leggere.</p>
    </body>
</html>

```

I tag JSP per l'uso dei JavaBeans sono i seguenti:

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

Inizializza e crea il riferimento al bean. Id è il nome con cui l'istanza del bean verrà indicata nel resto della pagina.

Per default ogni volta che una pagina JSP viene richiesta e processata viene creata un'istanza del bean con scope di default page. Con l'attributo scope è possibile estendere la vita del bean oltre la singola richiesta.

Scope	Accessibilità	Tempo di vita
page	Solo la pagina corrente	Fino a quando la pagina viene completata o fino al forward
request	La pagina corrente, quelle incluse e quelle a cui si fa forward	Fino alla fine dell'elaborazione della richiesta e restituzione della risposta
session	Richiesta corrente e tutte le altre richieste dello stesso client	Tempo di vita della sessione
application	Richiesta corrente e ogni altra richiesta che fa parte della stessa applicazione	Tempo di vita dell'applicazione

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

Produce come output il valore della proprietà del bean avente come id beanName.

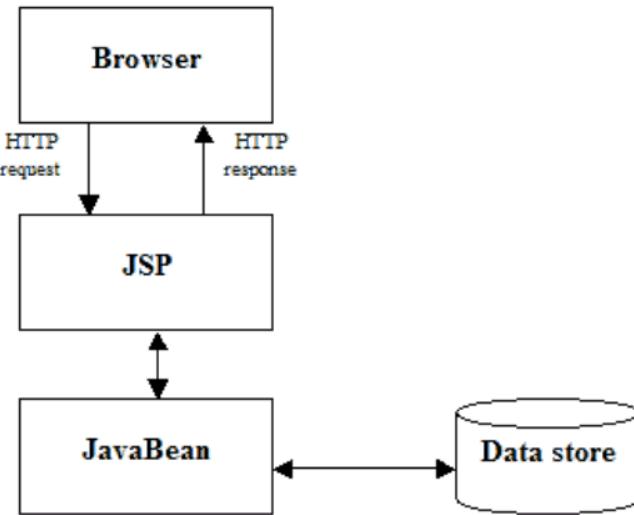
- `<jsp:setProperty name="beanName" property="propName" value="propValue"/>`

Consente di modificare il valore delle proprietà propName del bean con id beanName.

I tag per JavaBeans non supportano proprietà indirizzate, però un bean è un normale oggetto Java, è quindi possibile accedere a variabili e metodi associati. Ad esempio:

```
<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>
```

L'architettura J2EE a due livelli costituita da JSP per il livello di presentazione e JavaBean per il livello di business logic viene denominata Model 1.



Taglib

La direttiva taglib permette di aggiungere tag JSP non standard, caricando una libreria. La sintassi è `<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>`.

L'attributo uri fa riferimento ad un file xml, con estensione tld (tag library descriptor), che contiene informazioni sulle classi che implementano i tag, mentre l'attributo prefix indica il prefisso da utilizzare nei tag che fanno riferimento alla tag library.

Per definire una tag library occorrono due elementi:

- File TLD (Tag Lib Definition) che specifica i singoli tag e a quale "classe" corrispondono.

È un file XML che specifica:

- I tag che fanno parte della libreria.
- I loro eventuali attributi.
- "Body" del tag (se esiste).
- Classe Java che gestisce il tag.

- Le classi che effettivamente gestiscono i tag

Un esempio di file TLD è il seguente:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion> // versione della libreria
  <jspversion>1.1</jspversion>
  <shortname>hellolib</shortname> // nome della libreria
  <uri>hellobdir</uri> // directory in cui si trovano i file .class
  <tag>
    <name>helloWorld</name> // nome del tag
    <tagclass>helloTagClass</tagclass> // classe che gestisce il tag
    <bodycontent>empty</bodycontent> // body del tag (vuoto)
    <attribute>
      <name>who</name>
      <required>true</required> // attributo "who" obbligatorio
    </attribute>
  </tag>
</taglib>
```

Per utilizzare la libreria appena creata:

```
// inclusione
<%@ taglib uri="hellolib.tld" prefix="htl" %>
// utilizzo
<htl:helloWorld who="Mario">
```

Per la classe Java del tag, questa deve estendere TagSupport nel caso di tag semplici (è la classe base, ne esistono altre per tag più complessi) e deve implementare i metodi:

- `doStartTag()` : utilizza l'oggetto out restituito da PageContext per scrivere nell'output della pagina e se tag non ha nessun «body» deve ritornare come valore la costante SKIP_BODY.

- `doEndTag()`: restituisce usualmente la costante EVAL_PAGE che indica che, dopo il tag, prosegue la normale elaborazione della pagina.
- `setAttrName()`
- `getAttrName()`

Esempio:

```
public class helloTagClass extends TagSupport {

    private String who;

    public int doStartTag() throws JspException {
        try pageContext.getOut().println("Hello"+who+"<br>");
        catch(Exception e)
            throw new JspException( "taglib:" + e.getMessage() );
        return SKIP_BODY;
    }

    public int doEndTag() {
        return EVAL_PAGE;
    }

    public void setWho(String value) {
        who = value;
    }

    public String getWho() {
        return who;
    }
}
```

▼ 9.0 - JavaScript

▼ 9.1 - Componenti di JavaScript

Introduzione a JavaScript

JavaScript è un linguaggio di scripting sviluppato per dare interattività lato cliente alle pagine HTML. Nella pratica è lo standard client-side per implementare pagine dinamiche, meglio definite pagine attive.

JavaScript vs Java

Al di là del nome, JavaScript e Java sono due linguaggi completamente diversi. Alcune differenze principali sono le seguenti:

- JavaScript è interpretato e non compilato.
Il codice JavaScript viene eseguito da un interprete contenuto all'interno del browser.
- JavaScript è object-based ma non class-based, ovvero esiste il concetto di oggetto ma non di classe.
- JavaScript è debolmente tipizzato.

Sintassi del linguaggio

La sintassi di JavaScript è modellata su quella del C con alcune varianti significative, in particolare:

- 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.

Inserimento di JavaScript in una pagina HTML

HTML prevede un apposito tag per inserire script; la sua sintassi è

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

Nell'uso del tag <script> abbiamo due possibilità:

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

```
<script language="Javascript" src="nomefile.js"></script>
```

- Script interno: lo script è contenuto direttamente nel tag script.

```
<script type="text/javascript">  
    alert("Hello World!");  
</script>
```

Lo script può essere inserito sia nell'intestazione che nel body. Una variabile o qualsiasi altro elemento JavaScript può essere però richiamato solo se caricato in memoria, dunque ciò che si trova nell'header è visibile a tutti gli script del body, mentre quello che si trova nel body è visibile solo agli script che lo seguono.

Gestire l'assenza di JavaScript

Ci sono browser che non gestiscono JavaScript, ad esempio alcuni browser dei cellulari, dunque HTML prevede il 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>
```

Componenti di JavaScript

Variabili

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

```
var nomevariabile
```

Le variabili non hanno un tipo, ma lo assumono dinamicamente in base al dato a cui vengono agganciate.

Esiste lo scope globale e quello locale (ovvero dentro una funzione) ma, a differenza di Java, non esiste lo scope di blocco.

Valori speciali

Ad ogni variabile può essere assegnato il valore `null` che rappresenta l'assenza di un valore.

Una variabile non inizializzata ha invece un valore indefinito `undefined`.

Tipi primitivi

JavaScript prevede i tipi primitivi numeri, booleani e stringhe.

Per quanto riguarda i numeri questi sono rappresentati in formato floating point a 8 byte, non c'è distinzione fra interi e reali, esiste il valore speciale `NaN` (not a number) per le operazioni non ammesse (ad esempio, radice di un numero negativo) ed esiste il valore `infinite` (ad esempio, per la divisione per zero).

Oggetti

Gli oggetti in JavaScript vengono creati usando l'operatore `new`:

```
var o = new Object()
```

Gli oggetti sono tipi composti che contengono un certo numero di proprietà (attributi). Ogni proprietà ha un nome e un valore, e si accede alle proprietà con l'operatore `.`. Le proprietà non sono definite a priori, ma possono essere aggiunte dinamicamente; appena viene assegnato un valore ad una proprietà la proprietà comincia ad esistere.

Le costanti oggetto (object literal) sono oggetti con proprietà che vengono definite al momento della creazione. Per farlo si racchiude fra parentesi graffe un elenco di attributi nella forma nome:valore:

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

Array

Gli array vengono istanziati con

```
new Array([dimensione])
```

Sono tipi composti i cui elementi sono accessibili mediante un indice numerico. Non hanno una dimensione prefissata, infatti l'argomento

[dimensione] è opzionale. Possono contenere elementi di tipo eterogeneo.

Si possono creare e inizializzare usando delle costanti array (array literal) delimitate da parentesi quadre:

```
var a = [val1, val2, val3, ...]
```

Oggetti e array

Gli oggetti in realtà sono array associativi strutture composite i cui elementi sono accessibili mediante un indice di tipo stringa (nome) anziché attraverso un indice numerico. Si può quindi utilizzare anche una sintassi analoga a quella degli array o mischiarle:

```
var o = new Object();
o["x"] = 7;
o.y = 8;
o["tot"] = o.x + o["y"];
alert(o.tot);
```

Stringhe

In JavaScript le stringhe sono sequenze arbitrarie di caratteri in formato UNICODE a 16 bit e sono immutabili come in Java.

Esiste la possibilità di definire costanti stringa (string literal) delimitate da apici singoli o doppi. È possibile la concatenazione con l'operatore `+` e la comparazione con gli operatori `<, >, =, <=` e `!=`.

In JavaScript le stringhe sono dati di tipo primitivo che sembrano oggetti, infatti è possibile invocare metodi su una stringa o accedere ai suoi attributi, ad esempio:

```
var s = "ciao";
var n = s.length;
var t = s.charAt(1);
```

Espressioni regolari

JavaScript ha un supporto per le espressioni regolari (regular expressions) che sono un tipo di dato nativo del linguaggio.

Una espressione regolare può essere creata in due modi:

```
var r = /expression/ // metodo 1  
var r = new RegExp("expression") // metodo 2
```

Tipi valore e riferimento

In JavaScript si può distinguere tra tipi valore e riferimento.

In particolare i numeri e i booleani sono tipi valore, mentre stringhe, array e oggetti sono tipi riferimento.

Funzioni

Le funzioni in JavaScript possono essere definite utilizzando la parola chiave function, ad esempio:

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

Una funzione in JavaScript ammette parametri che sono privi di tipo, e restituisce un valore il cui tipo non viene definito.

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

```
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.

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. Ad esempio:

```
// costrutture
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);
    }
}

// istanziazione oggetto
var r = new Rectangle(5, 4);
```

È possibile inoltre, dopo aver definito un costruttore, aggiungere proprietà e metodi statici. Ad esempio:

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

// creazione variabile static
Circle.PI = 3.14159;
```

Operatori

Esistono alcuni operatori tipici di JavaScript:

- `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 `!=`).

Strutture di controllo

`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
```

Oggetto globale

In JavaScript esiste un oggetto globale implicito, al quale appartengono tutte le variabili e le funzioni definite in una pagina.

Questo oggetto espone anche 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.

▼ 9.2 - Cosa si può fare con JavaScript

Cosa si può fare con JavaScript

Con JavaScript si possono fare essenzialmente quattro cose:

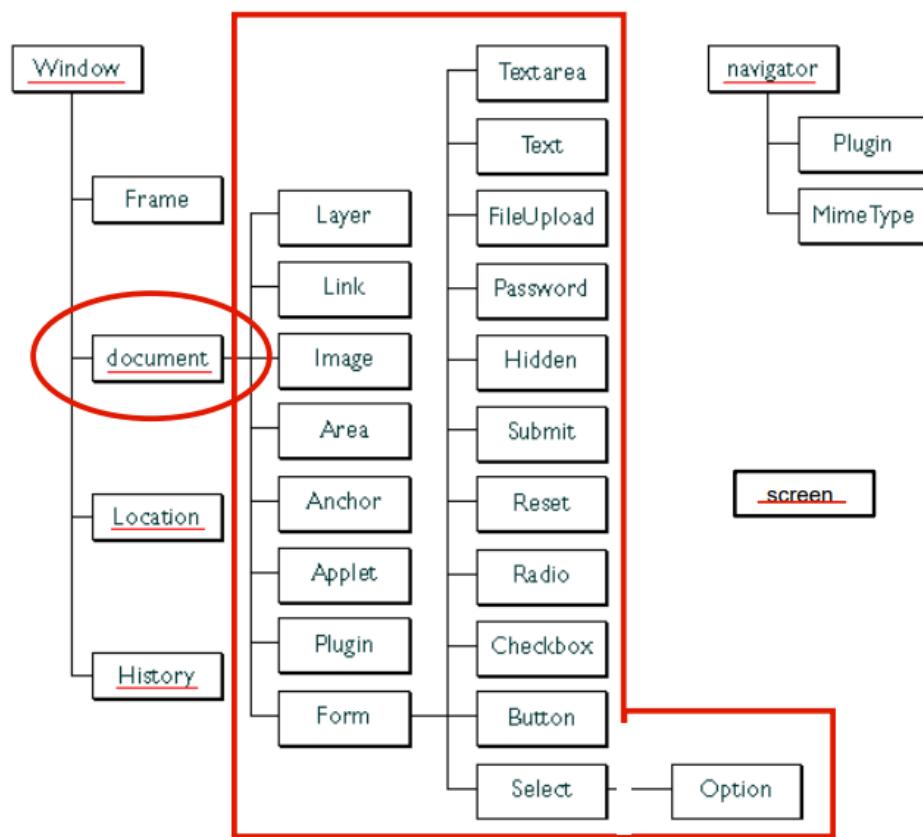
- Costruire dinamicamente parti della pagina in fase di caricamento.
- Rilevare informazioni sull'ambiente (tipo di browser, dimensione dello schermo, ecc.).

- Rispondere ad eventi generati dall'interazione con l'utente.
- Modificare dinamicamente il DOM (si parla in questo caso di Dynamic HTML o DHTML).

Browser Objects

Per interagire con la pagina HTML, JavaScript utilizza una gerarchia di oggetti predefiniti denominati Browser Objects e DOM Objects.

La gerarchia che ha come radice document corrisponde al DOM.



Costruzione dinamica della pagina

La più semplice modalità di utilizzo di JavaScript consiste nell'inserire nel corpo della pagina script che generano dinamicamente parti della pagina HTML. La pagina corrente è rappresentata dall'oggetto `document`, e per scrivere nella pagina si utilizzano i metodi `document.write()` e `document.writeln()`.

Rilevazione di informazioni sull'ambiente

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

Proprietà	Descrizione
<code>appCodeName</code>	Nome in codice del browser (poco utile)
<code>appName</code>	Nome del browser (es. Microsoft Internet Explorer)
<code>appVersion</code>	Versione del Browser (es. 5.0 (Windows))
<code>cookieEnabled</code>	Cookies abilitati o no
<code>platform</code>	Piattaforma per cui il browser è stato compilato (es. Win32)
<code>userAgent</code>	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 invece di ricavare informazioni sullo schermo, ed espone alcune utili proprietà tra cui `width` e `height` che permettono di ricavarne le dimensioni.

Modello ad eventi e interattività

JavaScript consente di associare script agli eventi causati dall'interazione dell'utente con la pagina HTML. L'associazione avviene mediante attributi collegati agli elementi della pagina HTML, e gli script prendono il nome di gestori di eventi (event handlers).

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 buttoni 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, buttoni, 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, buttoni, 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 per agganciare un gestore di evento ad un evento è la seguente:

```
<tag eventHandler="JavaScript Code">
```

È possibile inserire più istruzioni in sequenza, ma è meglio definire delle funzioni in testata che fungono da event handlers, ad esempio:

```
<head>
<script type="text/javascript">
    function compute(f) {
        if (confirm("Sei sicuro?"))
            f.result.value = eval(f.expr.value);
        else alert("Ok come non detto");
    }
</script>
</head>
<body>
<form>
    Inserisci un'espressione:
    <input type="text" name="expr" size=15 >
    <input type="button" value="Calcola" onClick="compute(this.form)"
    <br/>
    Risultato:
    <input type="text" name="result" size="15" >
</form>
</body>
```

Esplorare il DOM: Document

Il punto di partenza per accedere al Document Object Model (DOM) della pagina è l'oggetto `document`, il quale 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"]
```

In base all'equivalenza tra array associativi e oggetti vista in precedenza la seconda forma può essere scritta anche come `document.nomelink`.

Sull'oggetto `document` possono anche essere chiamati i seguenti metodi:

- `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.

Alcune proprietà di document sono invece le seguenti:

- `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.

Form

Un documento può contenere più oggetti form, i quali possono essere referenziati con il loro nome o mediante il vettore `forms[]` esposto da `document`. Gli elementi del form possono invece essere referenziati con il loro nome o mediante il vettore `elements[]`. Ogni elemento ha una proprietà `form` che permette di accedere al form che lo contiene. Per ogni elemento del form esistono proprietà corrispondenti ai vari attributi (`id`, `name`, `value`, `type`, `className` ...).

Un oggetto form presenta le seguenti 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.

Controlli di un form

Ogni tipo di controllo (widget) che può entrare a far parte di un form è rappresentato da un oggetto JavaScript (`Text`, `Checkbox`, `Radio`, `Button`, `Hidden` ecc.).

Proprietà comuni ai vari widget sono:

- `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.

Nello specifico, per quanto riguarda l'oggetto Text si hanno le seguenti proprietà e metodi aggiuntivi:

- `defaultValue`: valore di default.
- `maxLength`: numero massimo di caratteri.
- `readOnly`: sola lettura/lettura e scrittura.
- `size`: dimensione del controllo.
- `select()`: seleziona una parte di testo.

Per Checkbox e Radio si ha:

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

JavaScript e JQuery

JQuery è una libreria JavaScript (sviluppata da terzi) pensata appositamente per semplificare la vita del programmatore Web.

Prima di JQuery, gli sviluppatori tendevano a creare il proprio “framework JavaScript”; ciò permetteva loro di lavorare su specifici bug senza perdere tempo nel debugging di feature comuni. Ciò ha portato alla realizzazione da parte di gruppi di sviluppatori di librerie open source e gratuite. Con JQuery, lo sviluppatore usa API JavaScript preconfezionate pronte all'uso.

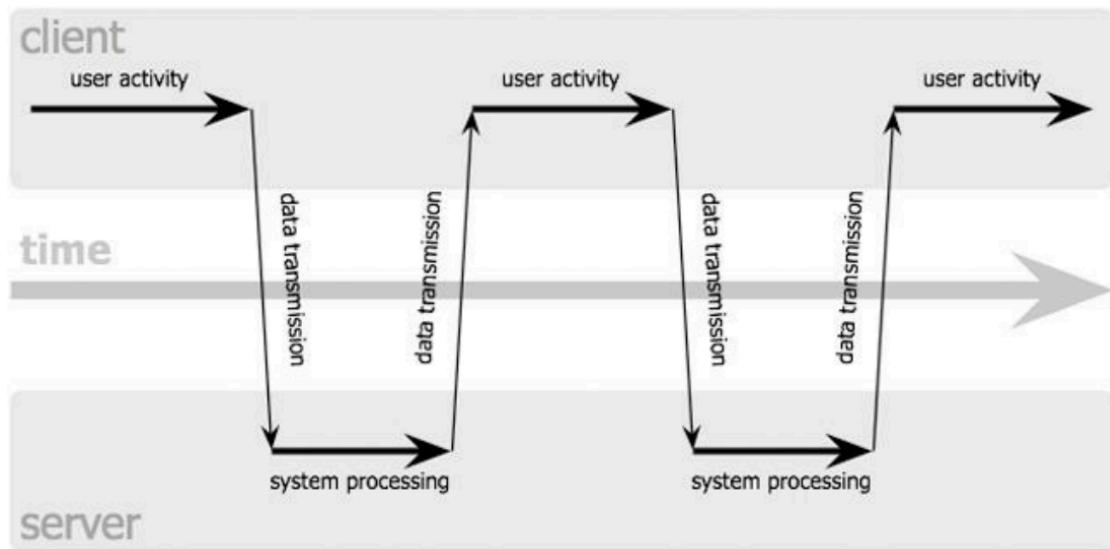
▼ 10.0 - AJAX

▼ 10.1 - Ajax

Introduzione a AJAX

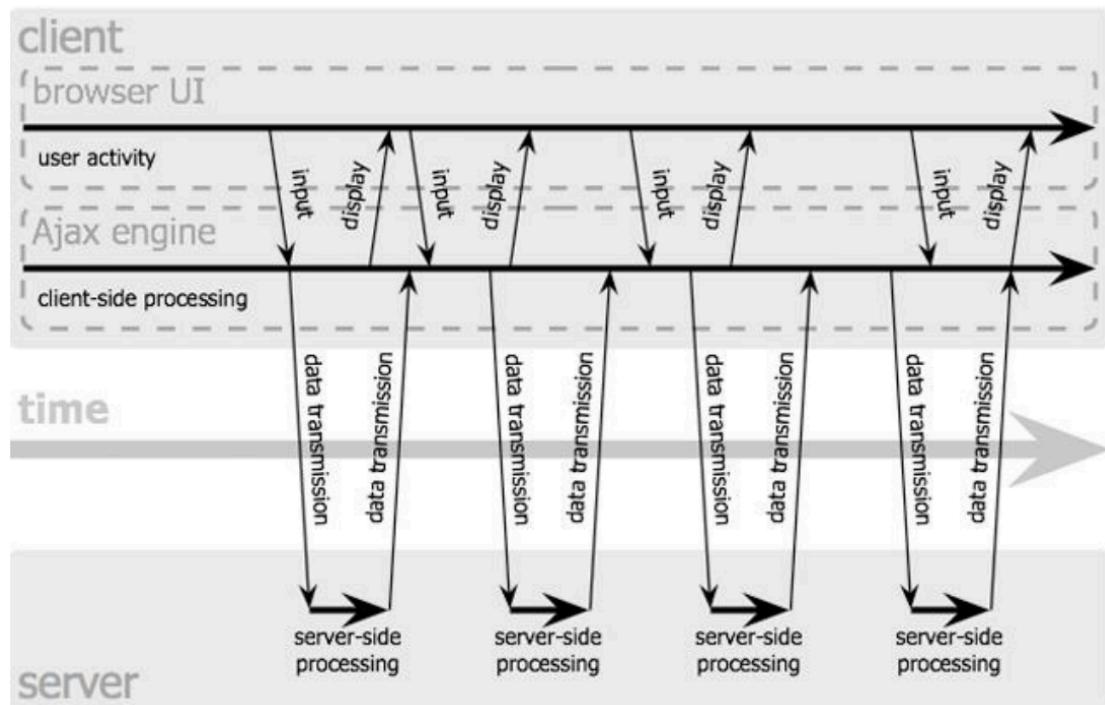
Le applicazioni Web tradizionali espongono un modello di interazione rigido in cui è necessario refresh della pagina da parte del server per la

gestione di qualunque evento. È dunque ancora modello sincrono: l'utente effettua una richiesta e deve attendere la risposta da parte del server.



Il modello di interazione AJAX è nato per superare queste limitazioni. L'idea alla base di AJAX è quella di consentire agli script JavaScript di interagire direttamente con il server.

L'elemento centrale è l'utilizzo dell'oggetto JavaScript `XMLHttpRequest`, il quale consente di ottenere dati dal server senza necessità di ricaricare l'intera pagina e realizza la comunicazione asincrona fra client e server.



XMLHttpRequest

L'oggetto XMLHttpRequest effettua la richiesta di una risorsa via HTTP al server Web. Può effettuare sia richieste GET che POST, e le richieste possono essere sia di tipo sincrono che asincrono.

Per creare un oggetto XMLHttpRequest si utilizza la seguente sintassi:

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

Alcuni dei metodi più utilizzati dell'oggetto XMLHttpRequest sono:

- `open()`

Inizializza la richiesta da formulare al server.

Lo standard W3C prevede 5 parametri, di cui 3 opzionali:

```
open(method, uri, [async], [user], [password])
```

L'uso più comune per AJAX ne prevede 3, di cui uno comunemente fissato:

```
open(method, uri, true)
```

Dove:

- `method`: stringa e assume il valore "get" o "post".
- `uri`: stringa che identifica la risorsa da ottenere (URL assoluto o relativo).
- `async`: valore booleano che deve essere impostato come true per indicare al metodo che la richiesta da effettuare è di tipo asincrono.

- `setRequestHeader(nomeheader, valore)`

Consente di impostare gli header HTTP della richiesta da inviare. Viene invocata più volte, una per ogni header da impostare, e per una richiesta GET gli header sono opzionali, mentre sono necessari per impostare la codifica utilizzata nelle richieste POST. È comunque importante impostare header connection di solito al valore close.

- `send(body)`

Consente di inviare la richiesta al server.

- `getResponseHeader(headername)`

Consente di leggere l'header HTTP corrispondente a headername che descrive la risposta del server. È utilizzabile solo nella funzione di callback e può essere invocato sicuramente in modo safe solo a richiesta conclusa (readystate=4).

- `getAllResponseHeaders()`

Consente di leggere gli header HTTP che descrivono la risposta del server. È utilizzabile solo nella funzione di callback e può essere invocato sicuramente in modo safe solo a richiesta conclusa (readystate=4).

- `abort()`

Consente l'interruzione delle operazioni di invio o ricezione.

Esempi di richieste AJAX:

```
// 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");

```

Lo stato e i risultati della richiesta vengono memorizzati dall'interprete JavaScript all'interno dell'oggetto XMLHttpRequest durante la sua esecuzione. Le proprietà comunemente supportate dai browser sono:

- `readyState`

Proprietà in sola lettura di tipo intero che consente di leggere in ogni momento lo stato della richiesta. Ammette 5 valori:

- 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.

- `onreadystatechange`

Proprietà che consente di registrare una funzione di callback che viene richiamata in modo asincrono ad ogni cambio di stato della proprietà ReadyState. La sintassi è:

```
xhr.onreadystatechange = nomefunzione
```

```
xhr.onreadystatechange = function(){istruzioni}
```

Per evitare comportamenti imprevedibili l'assegnamento va fatto prima del `send()`.

- `status`
Contiene un valore intero corrispondente al codice HTTP dell'esito della richiesta (in caso di successo 200, in caso di errore 403, 404, 500, ...).
- `statusText`
Contiene invece descrizione testuale del codice HTTP restituito dal server.
- `responseText`
Stringa che contiene il body della risposta HTTP. Disponibile solo a interazione ultimata (readystate=4).
- `responseXML`
Body della risposta convertito in documento XML se possibile, altrimenti null.ù

È importante notare che tramite AJAX, per motivi di sicurezza, è possibile fare richieste solo ad url appartenenti alla stesso dominio della richiesta. Per leggere dunque risorse da siti esterni occorre recuperarli tramite logica server-side (Servlet, JSP ecc.)

Gestione della compatibilità

Tutti i browser recenti supportano XMLHttpRequest come oggetto nativo. Versioni precedenti di IE lo supportano come oggetto ActiveX, solo dalla versione 4 e in modi differenti a seconda della versioni.

Se si vuole proprio essere compatibili con ogni versione di browser ancora installata nella pratica industriale si usano tecniche come la seguente:

```
function myGetXmlHttpRequest() {
  var xhr = false;
  var activeXopt = new Array("Microsoft.XmlHttp", "MSXML4.XmlHttp",
    "MSXML3.XmlHttp", "MSXML2.XmlHttp", "MSXML.XmlHttp" );
  // prima come oggetto nativo
  try xhr = new XMLHttpRequest();
  catch (e) {}
```

```

// poi come oggetto activeX dal più al meno recente
if (!xhr) {
    var created = false;
    for (var i = 0; i < activeXopt.length && !created; i++) {
        try {
            xhr = new ActiveXObject(activeXopt[i]);
            created = true;
        } catch (e) {}
    }
}
return xhr;
}

```

▼ 10.2 - JSON

Introduzione a JSON

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, inoltre non è semplicissimo da leggere e da mantenere.

JSON (JavaScript Object Notation) è un formato per lo scambio di dati, considerato molto più comodo di XML in quanto è leggero in termini di quantità di dati scambiati, molto semplice ed efficiente da elaborare da parte del supporto runtime al linguaggio di programmazione.

La sintassi JSON

La sintassi JSON si basa su quella delle costanti oggetto e array di JavaScript. Un oggetto JSON altro non è che una stringa equivalente a una costante oggetto di JavaScript.

Da stringa JSON a oggetto

JavaScript mette a disposizione la funzione `eval('espressione')` che invoca l'interprete per la traduzione della stringa passata come parametro.

Esempio:

```
var s = {"Paese": "Inghilterra", "AnnoFormazione": 1959, "TipoMusica": "Rock", "Membri": ["Paul", "John", "George", "Ringo"]};  
var o = eval('(' + s + ')');
```

L'uso di eval però presenta rischi, in quanto una stringa passata come parametro potrebbe contenere del codice malevolo. Di solito si preferisce dunque utilizzare parser appositi che traducono solo oggetti JSON e non espressioni JavaScript di qualunque tipo. Alcuni parser molto diffusi sono jabsorb per JavaScript e Google Gson per Java.

Jabsorb espone l'oggetto JSON con due metodi:

- `JSON.parse(strJSON)` : converte una stringa JSON in un oggetto JavaScript.
- `JSON.stringify(objJSON)` : converte un oggetto JavaScript in una stringa JSON.

Gson è invece una libreria java che consente di effettuare il parsing/deparsing di oggetti Java. Esempio di utilizzo:

```
// inizializzazione dell'oggetto Gson  
Gson g = new Gson();  
// serializzazione di un oggetto  
Person santa = new Person("Santa", "Claus", 1000);  
g.toJson(santa)  
// deserializzazione di un oggetto  
Person peterPan = g.fromJson(json, Person.class);
```

▼ 11.0 - React

Introduzione a React

React.js è una libreria JavaScript per la creazione di interfacce utente Web. È costruita sul linguaggio JavaScript, pertanto qualsiasi codice scritto in React.js esegue all'interno del browser. Ne consegue che React.js non è uno strumento per lo sviluppo lato back-end delle Web application, ma è in grado di interagire con tecnologie di backend quali Python/Flask, Ruby on Rails, Java/Spring, PHP, etc. tramite API.

L'approccio React in sintesi

React.js si ispira alla metodologia di sviluppo delle interfacce utenti del tipo "Single Page Application (SPA)". Una SPA è un'applicazione Web che interagisce col browser per modificare pagine Web in modo dinamico in funzione dei dati che arrivano dal back-end. Si contrappone all'approccio classico in cui il browser carica nuove pagine in seguito all'interazione dell'utente.

Lo sviluppo di una pagina Web avviene attraverso la scrittura di cosiddetti componenti i quali interagiscono con le API della libreria React.js che, a loro volta, manipolano il DOM per creazione di elementi di interfaccia utente.

Nel fare ciò React.js ha introdotto il concetto di Virtual DOM. Al verificarsi di un evento, invece di manipolare il DOM del browser, viene manipolato un virtual DOM che è una copia esatta del DOM del browser e si trova in memoria centrale. La manipolazione del Virtual DOM è più leggera di quella del DOM del browser. Lavorando con il Virtual DOM, React.js sarà in grado di inviare al DOM del browser solo le modifiche strettamente necessarie, rendendo così più leggero, efficiente e veloce il processo di rendering della pagina.

Linguaggio JSX

In React si fa uso del linguaggio JSX (JavaScript XML), il quale permette di mescolare JavaScript e html, consentendo di scrivere facilmente tag HTML all'interno di codice JavaScript e di piazzarli all'interno del DOM senza l'uso di metodi quali `createElement()` e/o `appendChild()`.

Bisogna notare che l'uso di JSX non è obbligatorio, ma di sicuro semplifica la vita dello sviluppatore. Un esempio di uso di JSX a confronto del non uso è il seguente:

```
// con JSX
const myelement = <h1>I Love JSX!</h1>;
ReactDOM.render(myelement, document.getElementById('root'));

// senza JSX
const myelement = React.createElement('h1', {}, 'I do not use JSX!');
ReactDOM.render(myelement, document.getElementById('root'));
```

```
// lista con JSX
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>;
ReactDOM.render(listElement, document.getElementById("container"));
// lista senza JSX
var item1 = React.DOM.li({ className: "item-1", key: "key-1" }, "Item 1");
var item2 = React.DOM.li({ className: "item-2", key: "key-2" }, "Item 2");
var item3 = React.DOM.li({ className: "item-3", key: "key-3" }, "Item 3");
var itemArray = [item1, item2, item3];
var listElement = React.DOM.ul({ className: "list-of-items" }, itemArray);
ReactDOM.render(listElement, document.getElementById("container"))
```

Variabili, costanti e funzioni

Un esempio di utilizzo di variabili/costanti in JSX è il seguente:

```
const nome = 'Giuseppe Verdi';
const element = <h1>Hello, {nome}</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

All'interno di `{}` è possibile inserire qualsiasi tipo di espressione o funzione che restituisca un valore.

Interprete di JSX

Il browser non è in grado di interpretare nativamente costrutti scritti in JSX, in quanto non scritti in linguaggio JavaScript. È quindi necessario aggiungere all'interno della pagina un riferimento a un cosiddetto pre-compilatore in grado di trasformare JSX in linguaggio javascript. Un esempio è Babel, un compilatore che supporta la traduzione in JavaScript di codice espresso in altri linguaggi, tra cui appunto JSX.

```
<head>
  ...

```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babelcore/5.8.24/browserify.js">
</head>
```

I 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. Ogni componente ha un ruolo ben definito dal punto di vista di ciò che rappresenta graficamente e si fa carico di gestire le interazioni dell'utente su quella particolare sezione di interfaccia.

Esistono due tipi di componenti, i componenti class e quelli function. Entrambi i tipi di componenti restituiscono codice HTML attraverso l'istruzione return. Inoltre il nome di un componente deve cominciare con una lettera maiuscola, altrimenti React lo tratterebbe come un normale tag HTML.

Esempio di componente di tipo function:

```
function Car() {
  return <h2>I am a Car!</h2>;
}

ReactDOM.render(<Car />, document.getElementById('root'));
```

Il vincolo di un componente di tipo function è quello di dover restituire l'elemento di cui fare rendering attraverso la parola chiave return.

Esempio di componente di tipo class:

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}

ReactDOM.render(<Car />, document.getElementById('root'));
```

Per creare un componente di tipo class, occorre creare una classe che estenda da `React.Component` e implementi obbligatoriamente il metodo `render()`. Così come per i componenti di tipo function, occorre che questo metodo restituisca l'elemento da renderizzare attraverso la parola chiave `return`.

È possibile creare un componente di tipo class anche utilizzando l'istruzione `React.createClass`:

```
var Car = React.createClass({
  render: function() {
    return <h2>Hi, I am a Car!</h2>;
  }
});
```

Il concetto di props

Sia per le funzioni che per le classi è possibile specificare delle proprietà ed assegnare a queste determinati valori. In React, le props assumono valori immutabili per i quali non è prevista alcuna alterazione, utili ad esempio per configurare il componente. L'oggetto che contiene queste proprietà prende il nome di props, e in fase di rendering è possibile accedere alle props di un componente richiamandole come fossero attributi di un tag HTML.

Nel caso di function l'oggetto props viene passato come parametro alla funzione:

```
function Car(props) {
  return <h2>I am a {props.colore} Car!</h2>;
}

ReactDOM.render(<Car colore="red"/>, document.getElementById('root'));
```

Per quanto riguarda le class l'oggetto props è built-in, quindi per invocarne l'uso occorre servirsi della parola chiave `this`:

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car. My name is {this.props.nome}</h2>;
}
```

```
    }
}

ReactDOM.render(<Car nome="Saetta McQueen"/>, document.getElementById('root'))
```

Il concetto di state

Esiste un altro modo per rappresentare le proprietà di un componente di tipo classe: lo state. Tutti i componenti di tipo classe, quelli di tipo function sono stateless, possiedono un oggetto built-in che prende il nome di state. A differenza delle props, le proprietà definite all'interno dell'oggetto state non sono immutabili, anzi, state è pensato proprio per contenere proprietà che nel tempo cambieranno. Quando una delle proprietà all'interno di state cambia valore, viene invocata la rirenderizzazione del relativo componente.

Analogamente a tutti i linguaggi ad oggetti, anche per il tipo di componente class è possibile definire un costruttore. Questo viene invocato prima del rendering e serve a:

- Inizializzare lo stato del componente.
- Inizializzare la gestione degli eventi.

Lo stato di un componente in React è invocabile tramite la parola chiave this:

```
class Car extends React.Component {
  constructor() {
    super(); // necessario, altrimenti "this" non può essere utilizzato
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a {this.state.color} Car!</h2>;
  }
}

ReactDOM.render(<Car />, document.getElementById('root'));
```

Per il costruttore è possibile imbattersi anche nella seguente sintassi:

```
constructor(props) {  
  super(props);  
  this.state = {...};  
}
```

L'oggetto state di un componente classe può essere modificato attraverso la funzione `setState()`, la quale può essere invocata solamente all'interno del componente. L'invocazione di tale funzione scatena una reazione da parte della libreria React, la quale provvederà a modificare lo stato e a re-invocare la funzione `render()` della classe.

Vediamo l'utilizzo del metodo `setState()` nell'esempio del lancio di un dado:

```
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 = <small>Lancia il dado cliccando <br />  
      sul pulsante sottostante</small>;  
    } else valore = <span>{this.state.numeroEstratto}</span>;  
  
    return (  
      <div className="card">
```

```

<p className="card_number">{valore}</p>
<button className="card_button" onClick={() => this.lanciaDado()}>
  Lancia il Dado</button>
</div>
);
}
}

```

È consigliato costruire componenti senza stato, infatti la tipica applicazione React è realizzata come una gerarchia di componenti: 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.

Da notare che in JSX il nome di un evento viene scritto in camel case e l'event handler viene invocato come stringa tra graffe. La funzione event handler, inoltre, può essere definita con o senza un parametro `e`, il quale indica un evento sintetico. Per una corretta invocazione dell'handler dell'evento occorre che l'oggetto chiamante sia il componente, ci sono dunque due alternative per fare ciò:

- All'interno del costruttore, forzare bind di this del metodo a this del componente.
- Invocare l'handler come un'arrow function.

Vediamo nella pratica le due alternative:

```

// bind
class Interruttore extends React.Component {
  constructor(props) {
    super(props);
    this.state = {acceso: true};
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({acceso: !this.state.acceso})
  }
}

```

```

...
render() {
  return (
    <button onClick={this.handleClick}>
      {this.state.acceso ? 'Acceso' : 'Spento'}
    </button>
  );
}
}

// arrow function
class Interruttore extends React.Component {
  constructor(props) {
    super(props);
    this.state = {acceso: true};
  }

  handleClick() {
    this.setState({acceso: !this.state.acceso})
  }
  ...

  render() {
    return (
      <button onClick={() => this.handleClick()}>
        {this.state.acceso ? 'Acceso' : 'Spento'}
      </button>
    );
  }
}

```

Form

In React, gli elementi HTML di cui è composto un form funzionano in un modo leggermente differente. La motivazione sta nel fatto che gli elementi form

mantengono naturalmente uno stato interno in base all'input dell'utente.

Esempio di gestione degli eventi in un form:

```
class EsempioForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
    console.log('onChange: lo stato ora vale ' + event.target.value);
  }

  handleSubmit(event) {
    alert('E\' stato inserito un nome: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Nome:
          <input type="text"
            value={this.state.value}
            onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Invocazione risorsa su un server

In React si possono effettuare HTTP request in diversi modi. Uno dal semplice utilizzo fa uso delle Fetch API fornite nativamente da javascript. Le Fetch API forniscono un'interfaccia js per accedere e manipolare parti della pipeline HTTP. Mettono a disposizione, inoltre, un metodo che fornisce un modo semplice e logico per recuperare le risorse in modo asincrono. Nella pagina seguente viene mostrata la composizione di una request HTTP di tipo POST all'interno di un event handler. Si noti che FormData è un'interfaccia javascript nativa supportata da tutti i browser.

Esempio di invio richiesta di tipo POST tramite Fetch API:

```
class MyForm extends React.Component {
  constructor() {
    super();
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleSubmit(event) {
    event.preventDefault();
    const data = new FormData(event.target); // interfaccia js
    fetch('/api/form-submit-url', {
      method: 'POST',
      body: data
    });
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label htmlFor="username">Enter username</label>
        <input id="username" name="username" type="text" />
        <label htmlFor="email">Enter your email</label>
        <input id="email" name="email" type="email" />
        <label htmlFor="birthdate">Enter your birth date</label>
        <input id="birthdate" name="birthdate" type="text" />
      </form>
    );
  }
}
```

```
        <button>Send data!</button>
    </form>
);
}
}
```

▼ 12.0 - Tecnologie e componenti server-side

Nel progetto di applicazioni Web in Java, due modelli di ampio uso e di riferimento: Model 1 e Model 2. Model 1 è un modello semplice e sta diventando obsoleto, Model 2 è invece un design pattern più complesso e articolato che separa chiaramente livello presentazione dei contenuti dalla logica. È usualmente associato con paradigma Model-View-Controller (MVC):

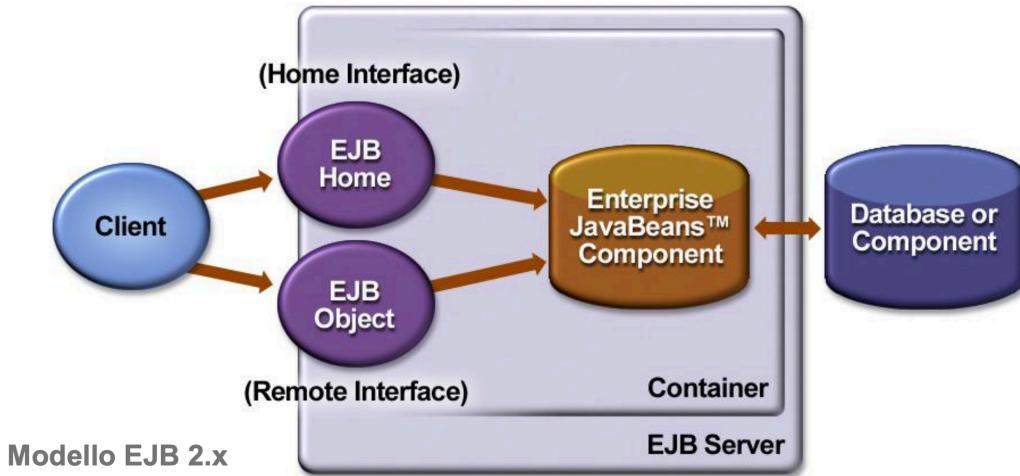
- Model rappresenta il livello dei dati, incluse operazioni per accesso e modifica.
- View – si occupa di rendering dei contenuti di model.
- Controller – definisce comportamento dell'applicazione.

In Java Model 2 tipicamente controller come EJB Session Bean o servlet, e view come JSP.

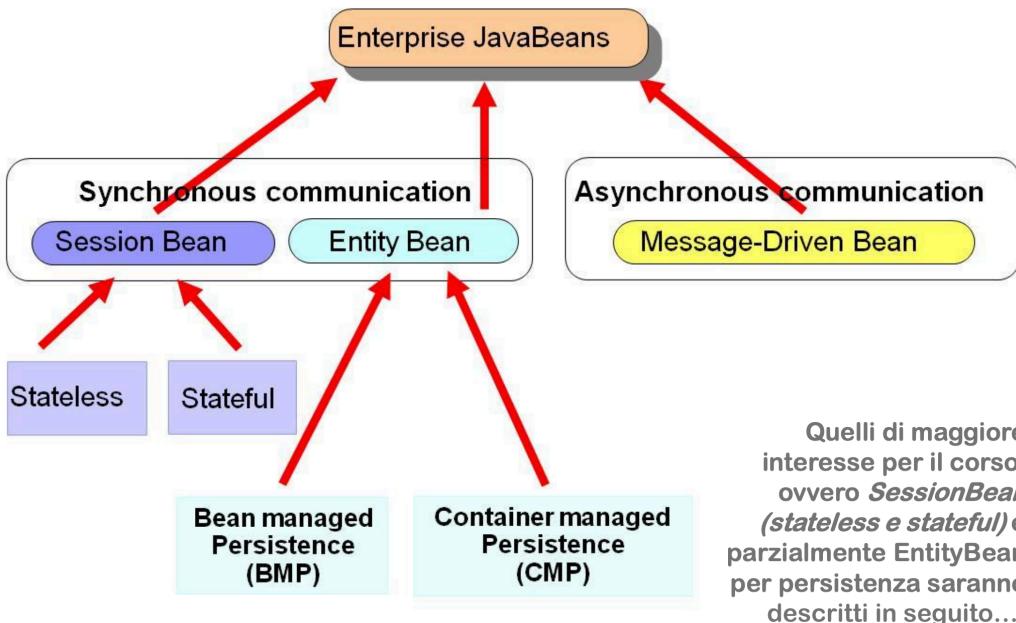
Java Model 2 è un modello a Container. Il Container si occupa di gestire i seguenti aspetti: pooling e concorrenza, transazionalità, gestione delle connessioni a risorse, persistenza, messaggistica e sicurezza.

Architettura EBJ

Idea di base: container pesante attivo all'interno di un EJB Server. Cliente può interagire remotamente con componente EJB tramite interfacce ben definite passando SEMPRE attraverso container.



I principali componenti EJB sono i seguenti:



- Session Bean

I session bean lavorano tipicamente per un singolo cliente e non sono persistenti. Non rappresentano dati in un DB, anche se possono accedere/modificare questi dati.

Possono essere stateless se eseguono una richiesta e restituiscono risultato senza salvare alcuna informazione di stato relativa al cliente, oppure stateful se mantengono stato specifico per un cliente.

Ciclo di vita di uno stateless session bean:

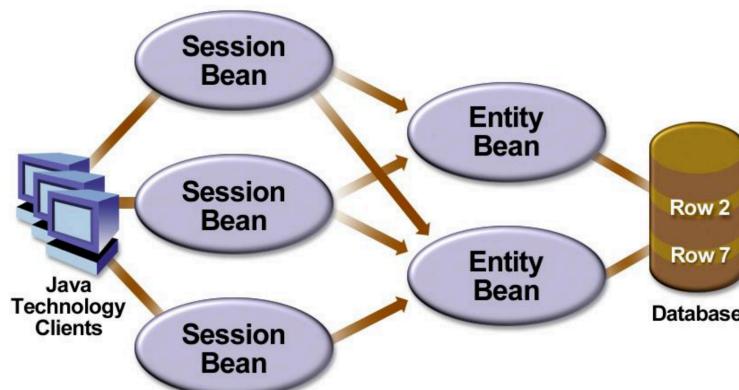
- No state (non istanziato; stato iniziale e terminale del ciclo di vita).
- Pooled state (istanziato ma non ancora associato ad alcuna richiesta cliente).
- Ready state (già associato con una richiesta EJB e pronto a rispondere ad una invocazione di metodo).

Per la gestione della concorrenza degli stateless session bean (i session bean non possono essere concorrenti per definizione) il Containter utilizza il metodo dell'Activation, il quale gestisce la coppia oggetto EJB + istanza di bean stateful:

- Passivation: disassociazione fra stateful bean instance e suo oggetto EJB, con salvataggio dell'istanza su memoria tramite serializzazione.
- Activation: recupero dalla memoria tramite deserializzazione dello stato dell'istanza e riassociazione con oggetto EJB.

- Entity Bean

Forniscono una vista ad oggetti dei dati mantenuti in un database. Il tempo di vita non è infatti connesso alla durata delle interazioni con i clienti, ma permangono nel sistema fino a che i dati esistono nel database, e nella maggior parte dei casi i componenti sono sincronizzati con i relativi database relazionali. L'accesso è condiviso tra clienti differenti.



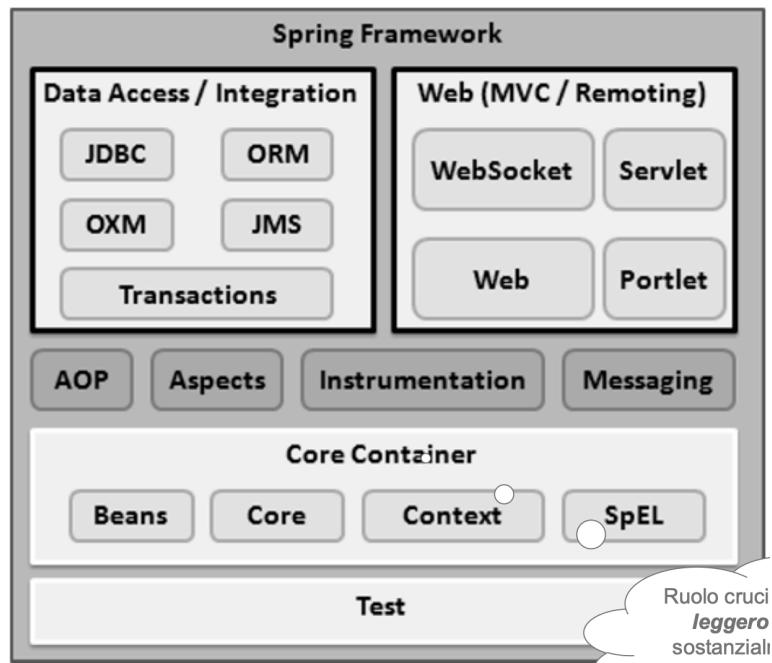
- **Message-Driver Bean (MDB)**

Svolgono il ruolo di consumatori di messaggi asincroni, vengono attivati in seguito all'arrivo di un messaggio e non possono essere invocati direttamente dai clienti, ma questi vi interagiscono tramite l'invio di messaggi verso le code o i topic per i quali questi componenti sono in ascolto (listener).

Spring

Spring è l'implementazione di modello a container leggero per la costruzione di applicazioni Java SE e Java EE.

Una delle funzionalità chiave di Spring è che è un framework modulare. Architettura a layer, possibilità di utilizzare anche solo alcune parti in isolamento.

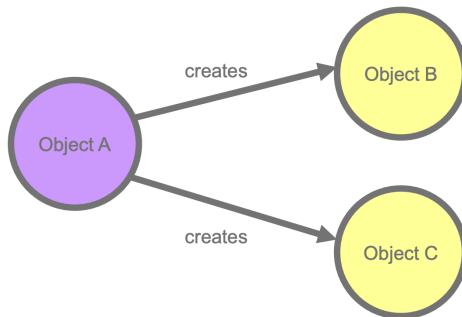


I soli moduli di interesse centrale per il corso sono:

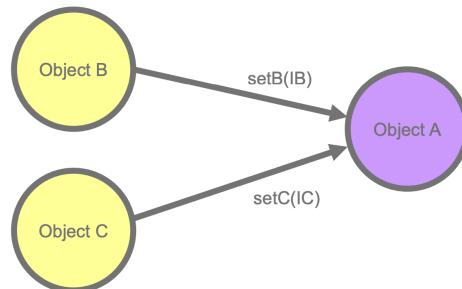
- Core Package: parte fondamentale del framework. Consiste in un container leggero che si occupa di Inversion of Control (Dependency Injection). L'elemento fondamentale è BeanFactory.
- MVC Package: ampio supporto a progettazione e sviluppo secondo architettura Model-View-Controller (MVC) per applicazioni Web.

Dependency Injection

Senza Dependency Injection un oggetto/componente deve esplicitamente istanziare gli oggetti/componenti di cui ha necessità, vi è dunque accoppiamento stretto tra oggetti/componenti.



Con Dependency Injection vengono creati gli oggetti/componenti quando necessario e vengono passati agli oggetti/componenti che li devono utilizzare quando necessario.



BeanFactory

L'oggetto BeanFactory è responsabile della gestione dei bean che usano Spring e delle loro dipendenze. L'oggetto BeanFactory viene creato dall'applicazione e, una volta creato, legge un file di configurazione e si occupa di fare l'injection.

Ogni bean deve avere un nome unico all'interno della BeanFactory contenente, al fine di permettere a questa di trovarlo.

```
<beans>
    <bean id="injectRef" class="InjectRef">
```

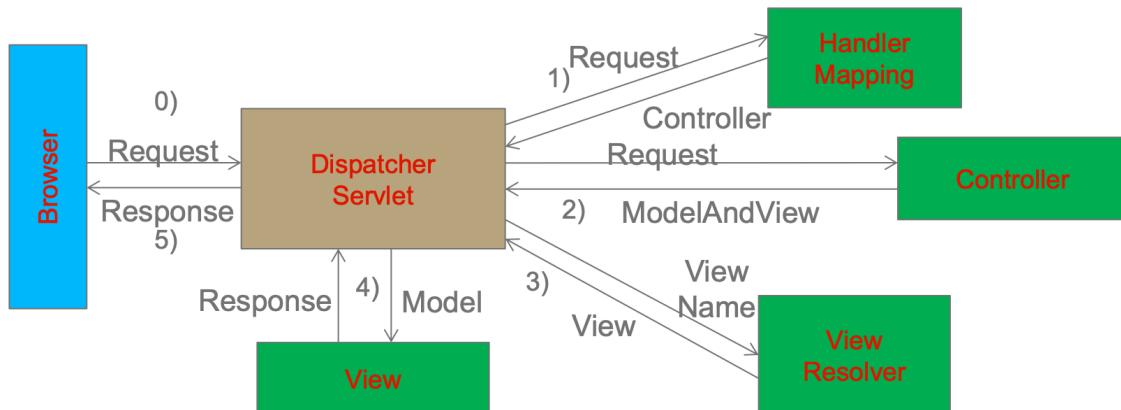
```

<property name="oracle">
    <ref local="oracle"/>
</property>
</bean>
</beans>

```

DispatcherServlet e Controller

Spring è progettato attorno a una servlet centrale che fa da dispatcher delle richieste (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 e una view. Cerca un View Resolver opportuno tramite cui scegliere View e creare HTTP Response.



Esistono diverse interfacce di Controller da poter implementare per realizzare i propri controller.

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.

▼ 13.0 - Node.js e cenni di HTTP/3

Node.js

Node.js è un modello basato sull'asincronicità progettato per estrema concorrenza e scalabilità. Al posto di thread, usa un event loop con stack al fine di ridurre fortemente overhead di context switching. Nella pratica vengono utilizzate delle callback richiamate al termine delle operazioni.

È utile fare ciò, soprattutto per applicazioni che effettuano operazioni sulla rete, per via del fatto che tali operazioni impiegano solitamente molti cicli di CPU, e se fatte in maniera sincrona rischiano di rallentare di molto il programma.

Operation	CPU Cycles
L1	3 cycles
L2	14 cycles
RAM	250 cycles
DISK	41 000 000 cycles
NETWORK	240 000 000 cycles

L'event loop lavora nel seguente modo per richiamare le funzioni di callback al termine delle operazioni:

```
while (true) {
  if (!eventQueue.isEmpty()) {
    eventQueue.pop().call()
  }
}

// when op finishes
eventQueue.push(eventHandler)
```

Node.js gestisce richieste concorrenti e non bloccanti in un unico thread, evitando i costi di context switching e di I/O bloccanti.

Node.js utilizza Javascript engine sia lato browser che servitore.

Stile di programmazione thread vs callback:

Threads

```
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!');
```

Works for **non-blocking**
calls in both styles

Callbacks

```
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!'); // Wrong!
```

Il core di Node 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: chiunque può creare un modulo Node con funzionalità aggiuntive e pubblicarlo in npm.

Il pattern di settaggio del listener e di emissione di un evento è il seguente:

```
myEmitter.on('myEvent', function(param1, param2) {
  console.log('myEvent occurred with ' + param1 + ' and ' +
  + param2 + '!');
});
myEmitter.emit('myEvent', 'arg1', 'arg2');
```

HTTP/3

HTTP/3, a differenza di HTTP/1 e 2, non è costruito al di sopra di TLS che usa UDP, ma di QUIC, che usa UDP. In questo modo è possibile avere più stream (richieste) contemporanei client server e più veloci (meno handshake e più performance).

▼ 14.0 - Web socket e JSF

Introduzione alle Web Socket

Limiti dell'HTTP tradizionale

Nel modello di interazione HTTP ci sono alcuni limiti, come nel modello di ricezione dei dati dal server solo quando questo ha aggiornamenti (es. chat).

Per implementare una comunicazione del genere si può far uso delle seguenti tecniche:

- Polling

Il client fa richieste al server a intervalli prefissati e il server risponde immediatamente. È realizzabile con le tecnologie viste finora, ad esempio in JavaScript, ma è una soluzione ragionevole solo quando la periodicità di aggiornamento è nota e costante, inefficiente invece se il server non ha dati da trasferire.

- Long polling

Il client invia una richiesta e il server attende fino a che non ha dati da inviare. Quando il client riceve la risposta, reagisce inviando immediatamente una nuova richiesta.

Ogni request/response si appoggia a una nuova connessione. Realizzabile in AJAX.

- Streaming/forever response

Il client manda una richiesta iniziale e il server attende fino a che non ha dati da inviare. Il server risponde inviando aggiornamenti tramite risposte parziali su una connessione mantenuta sempre aperta.

È una connessione di tipo half-duplex, in quanto solo il server invia dati al client e non viceversa. Realizzabile in AJAX.

- Connessioni multiple

Si effettua il long polling su due connessioni HTTP separate, una utilizzata per il long polling tradizionale, mentre l'altra per inviare i dati dal client verso il server.

Questa tecnica prevede però un complesso coordinamento e un overhead di due connessioni per ogni cliente.

Web Socket: principali caratteristiche

Per implementare questo tipo di connessione in maniera semplice ed efficiente esiste una tecnologia apposita, chiamata Web Socket. Le principali caratteristiche delle Web Socket sono le seguenti:

- Bi-direzionale: client e server possono scambiarsi messaggi quando desiderano.
- Full-duplex: non vi è nessun requisito di interagire solo come coppia request/response e di ordinamento dei messaggi.
- Unica connessione long running.
- Upgrade del protocollo HTTP, non vi è dunque la creazione di un nuovo protocollo e quindi il bisogno di una nuova infrastruttura.
- Uso efficiente di banda e CPU.

Elementi base del protocollo

Gli elementi base del protocollo delle web socket sono:

- Handshake: il cliente inizia la connessione e il servitore risponde accettando l'upgrade.
- Una volta stabilita la connessione, entrambe gli endpoint vengono notificati che la socket è aperta, ed entrambi possono inviare messaggi e chiudere la socket in ogni istante.

I dati vengono trasmessi con il minimo overhead in termini di dimensioni dell'header, e vi è la possibilità di frammentare un dato in più frame, utilizzando il bit FIN dell'header per indicare la fine del messaggio.

Web Socket API

La Web Socket API consente un approccio integrato con JavaScript lato cliente e JEE lato servitore. Gli endpoint WebSocket possono inviare/ricevere messaggi sotto forma di testo o binary.

Lato server

Per creare l'endpoint lato server occorre creare la seguente classe:

```
@ServerEndpoint("/actions")
public class WebSocketServer {
```

```

@OnOpen
public void open(Session session) {...}

@OnClose
public void close(Session session) {...}

@OnError
public void onError(Session session, Throwable throwable) {...}

@OnMessage
public void handleMessage(Session session) {...}
}

```

Per l'invio di messaggi occorre ottenere un oggetto `Session`, disponibile come parametro in molti metodi (es. `@OnOpen` e `@OnMessage`), ottenere da questo un `RemoteEndpoint`, tramite i metodi `Session.getBasicRemote` e `Session.getAsyncRemote`, e richiamare su di essi i metodi `sendText(String text)`, `sendBinary(ByteBuffer data)` e `sendPing(ByteBuffer appData)`.

Nel caso in cui un'istanza di endpoint è connessa a più peer, come nel caso di applicazioni di chat, è possibile utilizzare il metodo `getOpenSessions()` della classe `Session` per inviare un messaggio a tutti i peer connessi.

```

@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 la ricezione dei messaggi invece 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());
    }
}
```

Il container lato server crea una istanza della classe endpoint per ogni connessione, quindi si possono usare variabili di istanza per salvare lo stato del cliente. Inoltre, il metodo `Session.getUserProperties` restituisce una map modificabile per memorizzare 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 informazioni comuni a tutti i client, si possono anche utilizzare variabili di classe static, ma in questo caso è responsabilità dello sviluppatore assicurare che l'accesso sia thread-safe.

Uso di encoder e decoder

Java API per WebSocket forniscono supporto per la conversione di messaggi WebSocket in oggetti Java e viceversa tramite encoder e decoder.

Per quanto riguarda gli encoder occorre implementare una di queste interfacce: `Encoder.Text<T>` per messaggi testuali e `Encoder.Binary<T>` per messaggi binary.

Queste interfacce specificano il metodo di encode, dunque occorre implementarne una per ogni tipo Java custom che si vuole inviare come messaggio WebSocket. Successivamente bisogna aggiungere il nome delle classi encoder al parametro opzionale della annotazione `ServerEndpoint` e usare il metodo

`sendObject(Object data)` di `RemoteEndpoint.Basic` o di `RemoteEndpoint.Async` per inviare il messaggio.

```
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;
    }
}
```

```

// ... 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 e non occorre gestire problematiche di concorrenza.

Per i decoder bisogna implementare in modo analogo le interfacce

`Decoder.Text<T>` per messaggi testuali e `Decoder.Binary<T>` 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...
        }
    }
}

```

Anche i decoder sono associate ad un sola connessione con un solo peer, quindi non bisogna gestire la concorrenza.

Lato client

Lato client occorre creare un oggetto WebSocket tramite il costruttore

```
WebSocket(url, [protocols]) .
```

Alcune proprietà di questo oggetto sono:

- `bufferedAmount` : sola lettura, numero di byte di dati accodati.
- `onclose` : listener all'evento di chiusura della connessione.
- `onerror` : listener all'evento di errore sull'uso della WebSocket.
- `onmessage` : listener all'evento di ricezione di un messaggio dal server.
- `onopen` : listener all'evento di connessione aperta.
- `protocol` : sola lettura, sub-protocol selezionato dal servitore.

- `readyState` : sola lettura, stato corrente della connessione (WebSocket.CONNECTING 0, WebSocket.OPEN 1, WebSocket.CLOSING 2, WebSocket.CLOSED 3).
- `url` : sola lettura, URL assoluto associato.

Tra i metodi invece troviamo:

- `close([code], [reason])` : chiude la connessione.
- `send(data)` : accoda nuovi dati per l'invio.

Le WebSocket in JavaScript presentano inoltre un insieme di eventi ai quali è possibile agganciarsi usando `addEventListener()` o assegnando un event listener alla proprietà `onNomeEvento`. Questi eventi sono:

- `close` : evento di chiusura connessione.
- `error` : evento di errore che ha prodotto la chiusura di WebSocket, ad esempio con mancato invio di un dato.
- `message` : evento associato alla ricezione di un messaggio dal server.
- `open` : evento di apertura di una connessione WebSocket.

JSF (Java Server Faces)

È possibile costruire un backing bean tramite annotazione `@ManagedBean`, la quale consente di registrare automaticamente il componente come risorsa utilizzabile all'interno del container JSF. È inoltre necessario inserire il Bean nel file `faces-config.xml`.

```
package hello;
import javax.faces.bean.ManagedBean;
@ManagedBean
public class Hello {
    final String world = "Hello World!";
    public String getworld() {
        return world; }
}
```

Poi è possibile la costruzione di pagina Web tramite XHTML utilizzando il backing bean.

```
<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>
```

In tecnologia JSF, è inclusa servlet predefinita, chiamata FacesServlet, che si occupa di gestire richieste per pagine JSF.