Tecnologie Web

@Francesca Guzzi

Flashcards

Introduzione

Il web nasce nel 1989 con l'idea di evolvere FTP (File Transfer Protocol), e scambiare documenti statici ipertestuali →

Può essere visto come un'applicazione sopra il modello TCP/IP, livello applicativo (livello 7 dello stack OSI).

Hypertext: documenti messi in relazione tramite collegamenti monodirezionali (link)

La rete può essere vista come un grafo e i documenti sono i nodi.

Server: eroga risorse ovvero documenti → passivo

Client: consente visualizzazione e navigazione → attivo → browser → interagisce con il server tramite http.

Il client chiede una risorsa al server tramite URL, e il server se dispone della risorsa gliela manda, altrimenti manda errore, fine interazione.

Il World Wide Web è quindi un ipertesto distribuito sulla rete: URL indirizza le risorse disponibili sui server, HTTP ne consente il trasferimento, HTML consente la rappresentazione di ipertesti.

> WWW = URL + HTTP + HTML

URL

Il primo meccanismo ideato per accedere ad una risorsa sul web sono gli URI (Uniform Resource Identifier) che forniscono un meccanismo semplice ed estendibile per identificare una risorsa: sono un mapping concettuale ad un'entità, non si riferisce necessariamente ad una particolare versione dell'entità esistente in un dato momento; il mapping può rimanere inalterato anche se cambia il contenuto della risorsa.

Gli URI seguono una sintassi standard, sono identificatori uniformi: stringhe con una sintassi definita, dipendente dallo schema:

<scheme>:<scheme-specific-part>

Esiste un sottoinsieme di URI che condivide una sintassi comune per rappresentare relazioni gerarchiche in uno spazio di nomi:

<scheme>://<authority><path>?<query> → query opzionale, per parametri

Esistono due sottoinsiemi del concetto di URI:

- URN (Uniform Resource Name), identifica una risorsa per mezzo di un nome che deve essere globalmente unico e restare valido per sempre in un particolare dominio di nomi (*namespace*), ad esempio il codice ISBN che identifica i libri;
- **URL** (Uniform Resource Locator), identifica una risorsa per mezzo del suo meccanismo di accesso primario, ad esempio la locazione nella rete piuttosto che il suo nome o altro.

Specifica il protocollo necessario per il trasferimento della risorsa stessa, e tipicalemte il nome dello schema corrisponde al protocollo utilizzato:

<protocol>://[username>:<password>@]<host>[:<port>][/<path>]

HTTP

HTTP (HyperText Transfer Protocol) è il protocollo di livello applicativo utilizzato per trasferire le risorse web da server a client. Gestisce sia le richieste (URL) inviate al server che le risposte inviate al client (pagine).

É un protocollo stateless: ne il server ne il client mantengono, a livello di protocollo, informazioni relative ai messaggi precedentemente scambiati.

Si tratta quindi di un protocollo **request-response**, **stateless**, **oneshot** (versione 1.0). É basato su TCP e richieste e risposte sono trasmesse usando uno stream TCP (no UDP) seguendo questo schema:

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

Essendo poco efficiente aprire e chiudere per ogni risorsa una connessione apposia, viene proposta HTTP 1.1, dove la stessa connessione può essere utilizzata per una serie di richieste e una serie corrispondente di risposte. Le connessioni 1.0 vengono dette non persistenti mentre quelle 1.1 vengono definite persistenti: il server lascia aperta la connessione TCP dopo aver spedito la risposta e può quindi ricevere le richieste successive; chiude la connessione quando viene specificato nell'header del messaggio (richiesta) oppure quando non è usata da un certo tempo (time-out).

Venne introdotta anche HTTP 1.1 con pipelining, che consente al cliente di inviare molteplici richieste prima di terminare la ricezione delle risposte: le risposte devono però essere date nello stesso ordine delle richieste, poichè non è specificato un metodo esplicito di associazione richiesta-risposta, non c'è nessun identificatore di messaggio e rischierebbero di mescolarsi sulla connessione.

C'è anche HTTP/2, retrocompatibile, con compressione dell'header, server push, request-response multiplexing.

Un messaggio HTTP è definito da due strutture:

- *Message header*, che contiene tutte le informazioni identificative del messaggio;
- *Message body*, che contiene i dati trasportati dal messaggio.

Gli header sono costituiti da insiemi di coppie (nome:valore) che specificano caratteristiche del messaggio trasmesso o ricevuto e possono essere: generali della trasmissione (data, codifica, versione, ecc); relativi all'entità trasmessa (content-type, content-lenght, data di scadenza, ecc); riguardo la richiesta effettuata; della risposta generata.

Request e response

Il protocollo utilizza messaggi in formato ASCII.

Le HTTP request hanno questa struttura:

1. *Request line* → descrive il tipo di richiesta da attuare, o il suo stato (successo o fallimento). Questa riga di partenza è sempre singola e contiene 3 campi:
   - Il tipo di comando (request)
   - La risorsa che il client vuole
   - La versione del protocollo HTTP
2. *Request headers (opzionale);*

3. *Request body (opzionale);*

```
// request line: contiene comandi (GET,POST..)
GET /somedir/page.html HTTP/1.1
// header lines
Host: www.unibo.it
Connection: close // chiude la connessione al termine della richiesta
User-agent: Chrome/37.0
Accept: text/html, image/gif,image/jpeg
Accept-language:fr
```

Tra i comandi di richiesta ritroviamo:

- *GET:* serve per chiedere una risorsa ad un server, ed è il metodo più frequente poichè utilizza l'URL, prevede il passaggio di parametri, la lunghezza massima dell'URL è limitata;
- *POST:* a differenza di GET, i dettagli per richiedere la risorsa non sono nell'URL ma nel body del messaggio, non ci sono limiti di lunghezza e si ha una trasmissione client → server che però non porta alla creazione di una risorsa sul server;
- *PUT:* chiede la memorizzazione sul server di una risorsa all'URL specificato, serve quindi a trasmettere client → server; a differenza del POST si ha la creazione di una risorsa: l'argomento di PUT è la risorsa che ci si aspetta facendo un GET con lo stesso nome in seguito;
- *DELETE:* richiede la cancellazione della risorsa all'URL specificato;
- *HEAD:* simile a GET, ma riceve solo gli header relativi senza body (per verificare gli URL);
- *OPTIONS:* richiede informazioni sulle opzioni disponibili;
- *TRACE:* invoca il loop-back remoto a livello applicativo del messaggio di richiesta, consente al client di vedere cosa è stato ricevuto dal server, viene usato nella diagnostica e testing.

Dopo aver ricevuto einterpretato una richiesta del client, il server risponde con un messaggio di risposta HTTP, così composto:

1. *Status line:* è la prima riga del messaggio ed è costituita da 3 campi → versione di protocollo, codice di stato, frase testuale associata al codice di stato;
2. *Response header;*
3. *Response body;*

```
// status line: protocollo, codice di stato, status phrase
HTTP/1.1 200 OK
// header lines: HTTP 1.0 -> server chiude la connessione al termine della richiesta
//               HTTP 1.1 -> server mantiene aperta la connessione a meno di conn:close
Connection: close
Date: Thu, 06 Aug 2008 12:00:15 GMT
Server: Apache/2.3.0 (Unix)
Last-Modified: Mon, 22 Jun 2008 …...
Content-Length: 6821
Content-Type: text/html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.51 Transitional//EN">
<html>...</html>
```

Lo status code è un numero di tre cifre, di cui la prima indica la classe della risposta e le altre due la risposta specifica; i più importanti sono:

- *1xx-Informational*: è una risposta temporanea alla richiesta durante il suo svolgimento (sconsigliata da HTTP1.0);

- *2xx-Success*: significa che la richiesta è stata ricevuta, compresa e accettata da parte del server;
- *3xx-Redirection*: il server ha ricevuto e capito la richiesta, ma sono necessarie ulteriori azioni da parte del client per completarla;
- *4xx-Client Error*: significa che la richiesta contiene una sintassi errata o non può essere soddisfatta, vi è quindi un errore da parte del client;
- *5xx-Server Error*: significa che il server non ha soddisfatto una richiesta apparentemente valida (tipicamente per un problema interno).

Cookie

I cookies 🍪 sono piccole strutture dati (collezioni di stringhe) che si muovono come un token, dal client al server e viceversa; possono essere generati sia dal client che dal server e vengono passati ad ogni request/response.

Vengono inviate da un web server al browser dell'utente che effettua per la prima volta l'accesso a un sito web, per essere memorizzati sul computer del client. Vengono poi ritrasmessi al server ad ogni successivo collegamento al sito, in modo tale che il server riconosca il PC dell'utente. È un tentativo di rendere il protocollo HTTP non-stateless, poichè i cookies permettono di mantenere uno stato delle interazioni tra client e server. Sono contenuti nell'header del messaggio HTTP (sia nella request che nella response).

Autenticazione e sicurezza

Tecniche di **autenticazione** HTTP:

1. Filtro su set di indirizzi IP;
2. Form per la richiesta di username e password;
3. HTTP Basic: tipo di autenticazione che consiste nell'andare ad inviare l'username e la password codificati in base64 (codifica facilmente reversibile, deprecato da tempo);
4. HTTP Digest: più avanzato rispetto ad HTTP Basic, applica una funzione hash al nome utente e alla password prima di inviarli in rete con l'uso di valori *nonce* (*number only used once*) per prevenire attacchi replay.

HTTPS è un protocollo per la comunicazione sicura che consiste nella comunicazione (tramite il protocollo HTTP) all'interno di una connessione criptata, tramite crittografia asimmetrica, dal **Transport Layer Security (TLS)** o dal suo predecessore **Secure Sockets Layer (SSL)** fornendo come requisiti chiave:

- Confidenzialità → protezione della privacy;
- Integrità → dei dati scambiati dalle parti comunicanti;
- Autenticità → un'autenticazione del sito web visitato.

Quindi ad ogni interazione client/server sia il client che il server utilizzano un canale di trasporto TLS, garantendo le proprietà di sicurezza sopra elencate.

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. Un proxy deve interpretare e, se necessario, riscrivere le request prima di inoltrarle; di solito un proxy serve per:

- Filtrare le richieste → bloccare o permettere alcune richieste solo a specifici URL;
- Fare caching → permettere di scaricare in locale le risorse web per poi renderle disponibili per successive richieste da parte di altri client o dallo stesso client che l'ha voluta inizialmente nel caso di richieste multiple.

Gateway: server che agisce da intermediario per altri server, si occupa di instradare i pacchetti in due o più reti a cui è connesso. I gateway instradano i pacchetti verso altri gateway finchè non possono fornirli alla destinazione finale attraverso una rete fisica. Al contrario dei proxy, il gateway riceve le request come se fosse il server originale e il client non è in grado di identificare che response proviene da un gateway.

Tunnel: programma applicativo che agisce come "blind relay" tra due connessioni; una volta attivo non partecipa alla comunicazione HTTP.

Cache

Web caching → tecnica che consente di memorizzare copie temporanee di documenti Web (come pagine HTML, immagini ecc.) al fine di ridurre l'uso della banda ed il carico sul server. L'obiettivo è usare i documenti in cache per le successive richieste qualora alcune condizioni siano verificate. Esistono diversi tipi di web cache:

- **User Agent Cache** → mantiene una cache delle pagine visitate dall'utente, cache mantenuta dallo User Agent (tipicamente il browser);
- **Proxy Cache** → mantenuta dal Proxy server, ne esistono 2 tipi:
  - Forward Proxy Cache → serve per ridurre la necessità di banda, poichè il proxy server intercetta il traffico e mette in cache le pagine; successive richieste non provocano il download di ulteriori copie delle pagine server;
  - Reverse Proxy Cache → detto anche *Gateway Cache*, opera per conto del server e consente di ridurre il carico computazionale delle macchine → i client non sono capaci di distinguere se le pagine arrivano dal server o dal gateway.

HTTP Cache → per la gestione dell'aggiornamento della cache il protocollo HTTP, non prevedendo alcun tipo di cache interna, implementa 3 meccanismi:

- Freshness → controllata lato server dal campo *"Expires"* del response-header che comunica entro quanto la risorsa diventa obsoleta, e lato cliente dalla direttiva CacheControl: max-age;
- Validation → può essere usato per controllare se un elemento in cache è ancora attuale e corretto, ad es. nel caso in cui sia in cache da molto tempo, ciò viene fatto attraverso delle richieste *HEAD*;
- Invalidation → è normalmente un effetto collaterale di altre request che hanno attraversato la cache. Se per esempio viene mandata una POST, una PUT o una DELETE a un URL il contenuto della cache deve essere e viene automaticamente invalidato.

HTML

HyperText Markup Language è un linguaggio di codifica del testo del tipo a marcatori (markup) e viene utilizzato per descrivere le pagine che costituisocno i nodi dell'ipertesto.

Oltre a descriverne il contenuto, HTML associa significati grafici agli oggetti che definisce.

I tag HTML sono usati per definire il mark-up di elementi HTML: sono racchiusi in <> e normalmente accoppiati, con start tag ed end tag.

Un linguaggio di mark-up è composto da: un insieme di istruzioni dette tag che rappresentano le caratteristiche del documento, una grammatica che ne regola l'utilizzo e una semantica che ne definisce il dominio di applicazione. I linguaggi di mark-up possono essere procedurali (Tex,Latex) o dichiarativi: il mark-up descrive la struttura di un documento testuale identificandone i componenti (SGML,HTML,XML) → content object organizzati in modo gerarchico.

Un elemento può essere dettagliato mediante attributi, coppie "nome=valore" contenute nello start tag:

```
<input type='submit' value="NAME">Ok</input>
```

L'Internet Media Type rappresenta il tipo di contenuto di un messaggio: classifica i tipi di contenuto in base a tipo/sottotipo, ad esempio *text/html*.

Struttura

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

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

Poi troviamo l'header identificato dal tag ***<head>*** che contiene elementi non visualizzati dal browser (informazioni di servizio) ovvero *<title>*,*<base>*,*<link>*,*<script>*,*<style>* e *<meta>* (metadati di informazioni utili ad applicazioni esterne o al browser)*:* gli elementi di tipo *<meta>* si distinguono in due tipi, *http-equiv* o *name:* rispettivamente gli elementi del primo tipo possono essere *refresh, expires o content type*, quelli del secondo *author, description, copyright, keywords, date* (informazioni importanti ma non critiche).

```
<head>
  <meta http-equiv="Content-Type"
      content="text/html;
      charset=iso-8859-1">
  <meta name="description"
      content="Documentation about HTML">
  <meta name="keywords"
      content="HTML, tags, commands">
  <title>Impariamo l'HTML</title>
  <link href="style.css"
      rel=stylesheet type="text/css">
  </head>
```

Il tag ***<body>*** delimita il corpo del documento e contiene la parte che viene mostrata dal browser: può contenere intestazioni, strutture di testo, aspetto del testo, elenchi e liste, tabelle, form, collegamenti ipertestuali, immagini, media e script. Dal punto di vista del layout ritroviamo 3 grandi categorie: elementi "block-level", elementi "inline" e liste. Un'altra distinzione è quella tra elementi rimpiazzati (*replaced elements*) e non: gli elementi rimpiazzati sono quelli di cui il browser conosce le dimensioni intrinseche (*<img>*,*<input>*,*<textarea>*,*<select>*).

Tag

Gli heading, ovvero i titoli, usano i tag *<h1>*,*<h2>*,..,*<h6>* in importanza decrescente.

Tra i contenitori di testo ritroviamo:

- *<p>* è un elemento di blocco e lascia spazio prima e dopo la propria chiusura;
- * * è un elemento di blocco, non lascia spazio ma va a capo;
- ** è un elemento inline.

Per quanto riguarda gli stili di testo, i tag che svolgono questa funzione vengono suddivisi in fisici e logici:

- Tag fisici → definiscono lo stile del carattere in termini grafici (ad esempio corsivo, grassetto, sottolineato);
- Tag logici → forniscono informazioni sul ruolo svolto dal contenuto, e in base a questo adottano uno stile grafico (ad esempio **,*<acronym>*, ecc..);

Il tag *<hr>* serve ad inserire una riga di separazione; il tag *<font>* permette di formattare il testo (insieme ad altri tag stilistici).

Per quanto riguarda le liste:

- *<ul>* → liste non ordinate (puntate): gli elementi della lista vengono definiti mediante il tag *<li>* (list item); l'attributo *type* definisce la forma dei punti (disc, circle e square);

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

- <*ol*> → liste ordinate (numerate): l’attributo *type* ammette 5 valori (1,a,A,i,I);

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

- <*dl*> → liste di definizione

```
<dl>
  <dt>TERMINE1</dt>
  <dd>Definizione di Termine1</dd>
  <dt>TERMINE2</dt>
  <dd>Definizione di Termine2</dd>
...
</dl>
```

Il tag *<table>* racchiude la tabella, con tag per le celle di testata (*<th>),* celle del contenuto (*<td>)* e per le righe (*<tr>);*

```
<table border="1" >
    <caption align=“top”>
    <em>A test table with merged cells</em></caption>
    <tr>
        <th rowspan="2"></th>
        <th colspan="2">Average</th>
        <th rowspan="2">Red<br/>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>
```

Il link è il costrutto base di un ipertesto: è costituito da due estremi, detti ancore, e da una direzione di percorrenza: link = source anchor → destination anchor. Le ancore si esprimono con il tag *<a>*.

```
...
<p>Per maggiori informazioni leggete il
<a href="chapter2.html#section2 ">
secondo paragrafo del capitolo 2</a>.
Guardate anche questa
<a href="../images/forest.gif">
mappa della foresta incantata.</a></p>
...
```

```
...
<h1>Capitolo 2</h1>
<h2><a name="section1">Paragrafo 1</a></h2>
<p>Testo del primo paragrafo...</p>
<h2><a name="section2">Paragrafo 2</a></h2>
<p>Testo del secondo paragrafo...</p>
...
```

Un *<form>* (modulo) è una sezione che contiene elementi di controllo (bottoni, checkbox, radio button, liste, caselle di inserimento testo) che l’utente può utilizzare per inserire dati o interagire.

```
<form action="http://site.com/bin/adduser" method="post">
  <p>
    Nome: <input type="text" name="firstname">
```

```
    <!-- In questo caso input di testo, può essere "file" o qualsiasi altro
    elemento di controllo (checkbox, radio, button) -->
  </p>
</form>
```

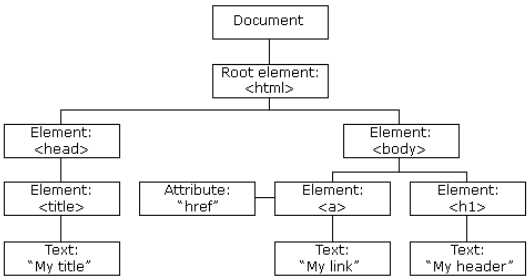
Dentro i form possono essere inserite anche liste di opzioni (volendo a scelta multipla → attributo *multiple* = *"multiple" e selected = "selected"* tenendo premuto ctrl durante la scelta):

```
<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>
```

Le liste lunghe possono anche essere organizzate in gruppi con il tag <*optgroup*>. Con il tag <*fieldset*> è possibile creare gruppi di campi a cui si può attribuire un nome utilizzando il tag <*legend*>. Il tag <*label*> permette di associare un'etichetta ad un qualunque controllo di un form (attributo id).

DOM

Una pagina HTML può essere rappresentata come una struttura ad albero, che prende il nome di **DOM**: *Document Object Model*. Quando un browser riceve una pagina HTML, ne fa il parsing e costruisce la struttura ad albero del DOM.




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

CSS

I file CSS (*Cascading Style Sheets)* sono fogli di stile multipli che agiscono uno dopo l'altro e indicano caratteristiche tipografiche e di layout.

Separano contenuto e presentazione: HTML → contenuto e struttura, diventa riutilizzabile; CSS → presentazione, anche su dispositivi o media diversi, riduce i tempi di scaricamento delle pagine e pulisce il codice HTML. Tra le caratteristiche principali dei CSS:

- Controllo sia dell'autore sia del lettore di un documento HTML;
- Indipendenti dalla specifica collezione di elementi ed attributi HTML;

I file CSS possono essere interni (stili inline, nell'header <*style*>) oppure esterni: uno o più file separati, referenziati con *href file.css* oppure *@import url*.

```
<head>
  <link rel="stylesheet" href="file.css" type="text/css">
  <!-- In alternativa si può utilizzare il tag '<style>' ma è ormai deprecato -->
  <style type="text/css">@import url(file.css);</style>
</head>
<body>
```

```
  <!-- E' possibile anche definire degli stili specifici in linea, ovviamente
  l'operazione è altamente sconsigliata! -->
  <h1 style="color: red"> Titolo1 </h1>
</body>
```

```
/* File file.css  */
BODY { color: red}
H1 { color: blue}
```

La regola CSS è composta da *selettore {dichiarazione}*:

Il selettore può essere universale (*{..}*), o per tipo, classi, identificatori;

I selettori sono gerarchici: per rappresentare una pagina HTML il browser deve riuscire ad applicare ad ogni elemento uno stile. Questa attribuzione può essere diretta o indiretta → l'elemento "eredita" lo stile dall'elemento che lo contiene: uno stile applicato ad un blocco esterno si applica anche ai blocchi in esso contenuto (non si ereditano le proprietà di formattazione dei box model).

La dichiarazione è composta da *proprietà:valore*.

Le proprietà possono essere singole o shorthand: alcune proprietà (ad esempio i margini) possono essere raggruppate → anzichè *margin-left, margin-top*, ecc un'unica proprietà seguita dai 4 valori.

I valori possono essere numeri, grandezze, percentuali, URL, stringhe, colori. Le grandezze possono anche essere relative:

- em: relativa alla dimensione del font in uso;
- px: pixel, dipende dal dispositivo di output.

Lo standard CSS definisce un insieme di regole di risoluzione dei conflitti che prende il nome di **cascade**, la cui logica di risoluzione si basa su tre elementi:

- Origine:
  - Autore
  - Browser
  - Utente
- Specificità del selettore;
- Ordine di dichiarazione.

L'unica eccezione è la clausola *!important* che ha sempre la precedenza sulle altre.

CSS definisce una sessantina di proprietà che ricadono nei seguenti gruppi: colori e sfondi, margini, caratteri e testo, box model, liste, display ed elementi floating, posizionamento e tabelle.

Web Dinamico

Il modello che abbiamo analizzato finora, basato su ipertesti distribuiti, ha una natura statica: pagine che vengono preparate staticamente a priori, con evidenti limiti, nessuna dinamicità. Prendiamo come esempio una pagina web statica, a cui vogliamo aggiungere un form per la ricerca:

```
<html>
  <head>
    <title> Ricerca dinosauri </title>
  </head>
  <body>
    <p>Enciclopedia dei dinosauri - Ricerca</p>
    <form method="GET" action="http://www.dino.it/cerca">
      <p>Nome del dinosauro
      <input type="text" name="nomeTxt" size="20">
```

```
      <input type="submit" value="Cerca" name="cercaBtn">
      </p>
    </form>
  </body>
</html>
```

Ritroviamo un'invocazione HTTP di tipo *GET*: il web server non è in grado di interpretare immediatamente URL con query → estensione specifica → *Common Gateway Interface* (**CGI**), standard per interfacciare applicazioni esterne con web server, può essere scritto in qualsiasi linguaggio.

La comunicazione tra CGI e web server può avvenire in 4 modi: variabili d'ambiente, parametri sulla linea di comando, standard input e standard output. Con il metodo *GET* il server passa il contenuto della form al programma CGI come se fosse da linea di comando di una shell; il metodo *POST* non aggiunge nulla all'URL specificata da *ACTION*, quindi la linea di comando nella shell contiene solo il nome del programma CGI, i dati del form vengono inviati al CGI tramite standard input. Il programma CGI elabora i dati in ingresso e passa i dati al server tramite stdout, il server preleva i dati dallo standard output e li invia al client incapsulandoli in messaggio HTTP.

Per creare una pagina web dinamica efficiente una soluzione è quella di separare gli aspetti di contenuto da quelli di presentazione, ad esempio utilizzando database relazionali (nel caso dell'esempio dell'enciclopedia) e quindi realizzando diversi programmi CGI, che usano il DB per ricavare le informazioni utili per la costruzione della pagina.

Le CGI sono però terribili dal punto di vista delle prestazioni: ogni volta che una di esse viene invocata crea un processo che viene distrutto alla fine; possono essere poco robuste, molto laboriose e garantire scarsa sicurezza, per questo si utilizzano Java Servlet.

La soluzione migliore è quella di realizzare un contenitore per le funzioni server-side: **Application Server**, che fornisce servizi di interfacciamento con il web server, gestione del tempo di vita, intefacciamento con il database e gestione della sicurezza, il tutto in una soluzione modulare. Modello a contenimento → molte funzionalità non controllate direttamente ma delegate ad una "entità supervisore" → **CONTAINER**: componenti trasportabili, riutilizzabili e mobili che forniscono "automaticamente" molte delle funzioni per supportare il servizio applicativo verso l'utente (supporto al ciclo di vita, sistema di nomi, qualità del servizio). Alcuni esempi di container sono le servlet, .NET, PHP, Ruby.

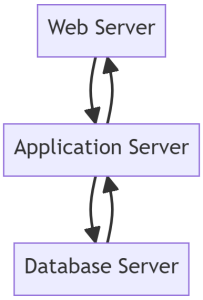
L'application server si avvale delle sessioni server-side (tecnica di base assieme ai cookie → storage lato cliente) per gestire lo **STATO** → l'interazione tra client e server può essere di due tipi:

- stateful: esiste stato dell'interazione (ogni messaggio può essere messo in relazione con i precedenti);
- stateless: non si tiene traccia dello stato → progettato con operazioni idempotenti → producono sempre lo stesso risultato indipendentemente dal numero di messaggi ricevuti dal server stesso.

Non tutte le applicazioni possono fare a meno dello stato (es. autenticazione, login) → dove c'è personalizzazione delle richieste web c'è stato:

- stato di esecuzione: rappresenta un avanzamento in una esecuzione, per sua natura è uno stato volatile;
- stato informativo persistente: mantenuto in una struttura persistente come un database (ad esempio per ordini eCommerce);
- stato di sessione: insieme dei dati che caratterizzano un'interazione con uno specifico utente → la sessione rappresenta lo stato associato ad una sequenza di pagine visualizzate da un utente → conversazione: sequenza di pagine di senso compiuto univocamente definita dall'insieme di pagine che la compongono e insieme delle interfacce di I/O tra le pagine (flusso della conversazione).

L'application server fa parte dell'**architettura a 3 tier**:



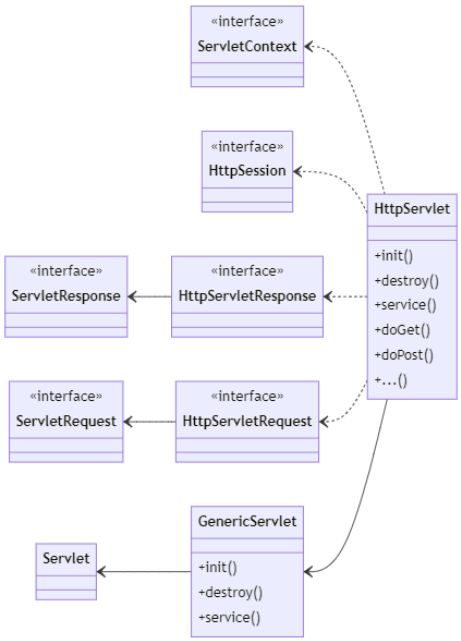
Questi tre servizi possono stare sullo stesso HW o su macchine separate → distribuzione verticale; orizzontalmente su ogni livello è possibile replicare il servizio su diverse macchine → distribuzione orizzontale.

Il web server è stateless, facile da replicare a differenza dell'application server (stato di sessione difficile da replicare) e del database server (stateful, replicazione delicata).

Servlet

Una Servlet è una classe Java che fornisce risposte a richieste HTTP; in termini più generali fornisce un servizio comunicando con il client mediante protocolli di tipo request/response. Le servlet estendono le funzionalità di un web server generando contenuti dinamici, ed eseguono direttamente in un web container.
Dopo la richiesta GET o POST tramite HTTP, la URL viene mappata su una web app che elabora la richiesta, genera la risposta e la ritorna al client.

Gli oggetti di tipo *Request* rappresentano la chiamata al server effettuata dal Client, mentre gli oggetti di tipo *Response* rappresentano le informazioni restituite al client in risposta ad una request.




```
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet{

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

Ridefiniamo *doGet()* e implementiamo la logica di risposta a HTTP GET, producendo in output un testo HTML → all'arrivo di una richiesta HTTP il Servlet Container crea gli oggetti request e response e li passa alla servlet.

Il servlet container controlla e supporta automaticamente il **ciclo di vita di una servlet**. Di solito c'è una sola istanza di servlet e un thread assegnato ad ogni richiesta http per servlet. Più thread quindi condividono la stessa istanza di una servlet —> concorrenza.
In alternativa si può usare il modello single-threaded (**deprecato!!!**) che indica al container di creare un'istanza della servlet per ogni richiesta concorrente.

Metodi ciclo di vita servlet: (definiti nella classe *GenericServlet*)

- *init()*: inizializza la servlet;
- *service()*: viene chiamato ad ogni HTTP request → chiama doGet o doPost → *doGet() throws ServletException*;
- *destroy()*: disattiva la servlet.

Metodi

Metodi di creazione della *Response*:

```
void setStatus(int statusCode) // 200 OK, 404 Page not found, ...
void sendError(int sc) // invia un errore con codice
void sendError(int code, String message) // invia un errore con codice e messaggio
void setHeader(String headerName, String headerValue) // imposta un header arbitrario
void setDateHeader(String name, long millisecs) // imposta la data
void setIntHeader(String name, int headerValue) // imposta header con un valore intero
void setContentType // configura il content-type (si usa sempre)
void setContentLength // utile per la gestione di connessioni persistenti
void addCookie(Cookie c) // consente di gestire i cookie nella risposta
void sendRedirect // imposta location header e cambia status code (forza ridirezione)
PrintWriter getWriter() // mette a disposizione uno stream di caratteri
ServletOutputStream getOuputStream() // mette a disposizione uno stream di byte
```

Metodi per accedere alla *Request*:

```
String getParameter(String parName) // restituisce il valore di un parametro
String getContextPath() // restituisce le informazioni sulla parte di URL
String getQueryString() // restituisce la stringa di query
String getPathInfo() // per ottenere il path
String getPathTranslated() // per ottenere informazioni sul path nella forma risolta
String getHeader(String name) // restituisce il valore di un header individuato da name
Enumeration getHeaders(String name) // restituisce valori sotto forma di enum di stringhe
Enumeration getHeaderNames() // elenca i nomi di tutti gli header presenti nella richiesta
int getIntHeader(String name) // valore di un header convertito in intero
long getDateHeader(String name) // valore della parte date di header, convertita in long
String getRemoteUser() // nome di user se la servlet ha autenticazione, null altrimenti
String getAuthType() // nome schema di autenticazione usato per proteggere la servlet
boolean isUserInRole(String role) // true se l'utente è associato al ruolo specificato
Cookie[] getCookies() // restituisce array di cookie che il client ha inviato alla request
InputStream getInputStream() // consente di leggere il body della richiesta
```

Un esempio di *doGet()* con request parametrica, implementato tramite Servlet:

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

Risposta HTTP:

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

Deployment

Un'applicazione web deve essere installata e questo processo prende il nome di deployment, (*web.xml)* che comprende:

- Definizione del runtime enviroment di una Web Application;
- La mappatura delle URL sulle servlet;
- Definizione di impostazioni di default (welcome page, pagine di errore ecc..);
- Configurazione della sicurezza.

Il deployment viene rappresentato tramite gli archivi web (file .war), ovvero file jar con una struttura particolare che contiene file di configurazione XML, con elementi descrittivi e l'elenco delle servlet attive sul server, mappate con i loro rispettivi URL. Una servlet accede ai propri parametri di configurazione mediante l'interfaccia *ServletConfig*, alla quale si può accedere tramite o il metodo *init()* o *getServletConfig()*.
Estendiamo quindi l'esempio rendendo parametrico il titolo della pagina HTML e ridefinendo il metodo *init()* e *doGet()*:

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

```
import java.io.*
import java.servlet.*
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet {

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

Servlet Context

Ogni Web application esegue in un contesto: corrispondenza 1:1 tra una Web-app e suo contesto; l'interfaccia *ServletContext* è la vita del contesto della Web Application da parte della servlet. I parametri di inizializzazione del contesto sono definiti all'interno di elementi *context-param* in *web.xml*.

Si può ottenere un'istanza di tipo ServletContext all'interno della servlet utlizzando il metodo *getServletContext()*.

Attributi di contesto → sono accessibili a tutte le servlet e funzionano come variabili "globali":

```
// Scrittura
ServletContext ctx = getServletContext();
ctx.setAttribute(“utente1”, new User(“Giorgio Bianchi”));
// 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

HTTP è un protocollo stateless, non fornisce in modo nativo meccanismi per il mantenimento dello stato. Le applicazioni web però hanno spesso bisogno di stato → vengono definite due tecniche per mantenere traccia delle informazioni di stato:

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

Cookie: 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 sul file associato);
- un valore booleano per definirne il livello di sicurezza.

La classe *Cookie* modella il cookie HTTP:

```
// Creazione
Cookie c = new Cookie("MyCookie", "test");
c.setSecure(true);
c.setMaxAge(-1);
c.setPath("/");
response.addCookie(c);

// Lettura
Cookie[] cookies = request.getCookies();
if(cookies != null) {
  for(int i=0; i < cookies.length; i++) {
  Cookie c = cookies[i];
  out.println("Un cookie: " + c.getName()+"="+c.getValue());
  }
}
```

Sessione: la sessione Web è un'entità gestita dal web container. È condivisa fra tutte le richieste provenienti dallo stesso client: consente di mantenere, quindi, informazioni di stato (di sessione).

Può contenere dati di varia natura ed è identificata in modo univoco da un session ID. Viene usata dai componenti di una Web application per mantenere lo stato del client durante le molteplici interazioni dell'utente con quest'ultima. Il session ID è usato per identificare le richieste provenienti dallo stesso utente e mapparle sulla corrispondente sessione.

Ci sono due tecniche per trasmettere l'id:

- includerlo in un cookie (session cookie) → sappiamo però che non sempre i cookie sono attivati nel browser;
- inclusione del session ID nella URL → URL rewriting;

```
// Ottengo la sessione, true mi indica che se la sessione non esiste, viene creata
HttpSession session = request.getSession(true);
```

```
// Gestione contenuto di una sessione
Cart sc = (Cart)session.getAttribute(“shoppingCart”);
sc.addItem(item);
session.setAttribute(“shoppingCart”, new Cart());
session.removeAttribute(“shoppingCart”);
Enumeration e = session.getAttributeNames();
while(e.hasMoreElements())
  out.println(“Key; “ + (String)e.nextElements());
```

Altri metodi fondamentali:

```
String getId() // restituisce l’ID di una sessione
boolean isNew() // ci indica se la sessione è nuova o no
void invalidate() // permette di invalidare (distruggere) una sessione
long getCreationTime() // restituisce da quanto tempo è attiva una sessione
long getLastAccessedTime() // da informazioni su quando è stata utilizzata l’ultima volta
```

Scope

Gli oggetti di tipo *ServletContext, HttpSession, HttpServletRequest* forniscono metodi per immagazzinare e ritrovare oggetti nei loro rispettivi ambiti (scope). Lo scope è definito dal tempo di vita (lifespan) e dall’accessibilità da parte delle servlet.

Ambito	Interfaccia	Tempo di vita	Accessibilità
Request	*HttpServletRequest*	Fino all’invio della risposta	Servlet corrente e ogni altra pagina inclusa o in forward
Session	*HttpSession*	Lo stesso della sessione utente	Ogni richiesta dello stesso client
Application	*ServletContext*	Lo stesso dell’applicazione	Ogni richiesta alla stessa Web app anche da client diversi e per servlet diverse

Gli oggetti scoped forniscono i seguenti metodi per immagazzinare e ritrovare oggetti nei rispettivi ambiti (scope):

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

Include: includere risorse web (altre pagine, statiche o dinamiche) può essere utile quando si vogliono aggiungere contenuti creati da un’altra risorsa:

- Inclusione di risorsa statica → includiamo un’altra pagina nella nostra (es. banner)
- Inclusione di risorsa dinamica → la servlet inoltra una request ad una componente web che la elabora e restituisce il risultato, che viene incluso nella pagina prodotta dalla servlet.

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

Per includere o inoltrare una risorsa si ricorre a un oggetto di tipo `RequestDispatcher`

```
RequestDispatcher rd = getServletContext().getRequestDispatcher("/inServlet");
// inclusione
```

```
rd.include(request, response);
// forward
rd.forward(request, response);
```

Java Server Pages - JSP

Le JSP sono l'altra componente base della tecnologia J2EE, relativamente alla parte web. In realtà esse vengono trasformate in servlet dai container; con le JSP si estende l'HTML con codice Java custom. Quando viene effettuata una richiesta ad una JSP:

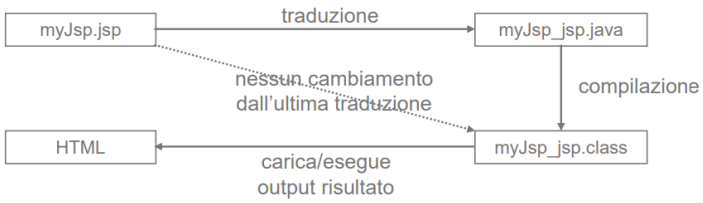
1. la parte HTML viene direttamente trascritta sullo stream di output;
2. il codice Java viene eseguito sul server per la generazione del contenuto HTML dinamico;
3. la pagina HTML così formata (parte statica + parte generata dinamicamente) viene restituita al client.

Ogni volta che incontra un tag `<%...%>` valuta l'espressione Java contenuta al suo interno e inserisce al suo posto il risultato dell'espressione:

```
<html>
  <body>
    <% String visitor=request.getParameter("name");
    if (visitor == null) visitor = "World"; %>
    Hello, <%= visitor %>!
  </body>
</html>
```

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

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



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

Le JSP non rendono inutili le servlet, bensì rendono molto semplice presentare documenti HTML o XML all'utente, e facilitano la progettazione grafica e l'aggiornamento delle pagine.

Tag JSP

Le parti variabili della pagina sono contenute all'interno di tag speciali. Ritroviamo due tipi di sintassi:

1. **Scripting-oriented tag** → sono definite da delimitatori entro cui è presente lo scripting (self-contained), sono di quattro tipi:
   - `<%! %>` dichiarazione;
   - `<%= %>` espressione;
   - `<% %>` scriptlet;
   - `<%@ %>` direttiva.
2. **XML-oriented tag** → equivalenti ai delimitatori visti nelle scripting-oriented tag:
   - `<jsp:declarationdeclaration</jsp:declaration>` dichiarazione;

- `<jsp:expressionexpression</jsp: expression>` espressione;
- `<jsp:scriptletjava_code</jsp:scriptlet>` scriptlet;
- `<jsp:directive.dir_type dir_attribute />` direttiva.

Dichiarazione: si usano i delimitatori di dichiarazione per dichiarare variabili e metodi.

```
<%! 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;
  }
%>
```

Espressione: si usano i delimitatori di espressione per valutare espressioni Java il cui risultato viene convertito in stringa e inserito nella pagina al posto del tag.

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

Scriptlet: si usano per aggiungere un frammento di codice Java eseguibile dalla JSP; consentono tipicamente di inserire logiche di controllo di flusso nella produzione della pagina.

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

Direttiva: comandi JSP valutati a tempo di compilazione, non producono output visibile. I più importanti sono:

- Page: definisce una serie di attributi che si applicano all’intera pagina

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

- Include: include un altro documento

  ```
  <%@ include file=“myHeaderFile.html” %>
  ```

- Taglib: carica una libreria di custom tag

```
<%@ uri="tagLibraryURI" prefix="tagPrefix"%>
```

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:

1. ***page***

```
/* gli oggetti page rappresentano l'istanza corrente della servlet e possono essere utilizzati per accedere a tutti i metodi def
initi nelle servlet */

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

2. ***config***

```
// contiene la configurazione della servlet (parametri di inizializzazione)

String getInitParameterName() // Da tutti i nomi dei parametri di inizializzazione
String getInitParameter(String name) // Restituisce il valore del parametro dato il nome
```

3. ***request***

```
/* rappresenta la richiesta alla pagina JSP (è il parametro di request passato al metodo service() della servlet */

String getParameter(String name) // Da il valore di un parametro individuato dal nome
Enumeration getParameterNames() // Restituisce l'elenco dei nomi dei parametri
String getHeader(String name) // Restituisce il valore di un header individuato dal nome
Enumeration getHeaderNames() // Restituisce l'elenco dei nomi degli headers
Cookie[] getCookies() // Restituisce un array di oggetti cookie che client ha inviato alla request
```

4. ***response***

```
// rappresenta la risposta che viene restituita al client

void setHeader(String headerName, String headerValue) // Imposta header
void setDateHeader(String name, long millisecs) // Imposta data
addHeader, addDateHeader, addIntHeader // Aggiungono nuova occorrenza di un dato header
setContentType // Determina content-type
addCookie // Consente di gestire i cookie nella risposta
PrintWriter getWriter() // Restituisce uno stream di caratteri (un'istanza di PrintWriter)
ServletOutputStream getOuputStream() // Restituisce uno stream di byte (un'istanza di ServletOutputStream)
```

5. ***out***

```
// stream di caratteri e rappresenta lo stream di output della pagina

void print(String str) // Stampa la stringa sullo stream di output
boolean isAutoFlush() // Dice se output buffer è stato impostato in modalità autoFlush
getBufferSize() // Restituisce le dimensioni del buffer
getRemaining() // Indica quanti byte liberi ci sono nel buffer
void clearBuffer() // Ripulisce il buffer
void flush() // Forza l'emissione del contenuto del buffer
void close() // Fa flush e chiude stream
```

6. ***session***

```
// fornisce informazioni sul contesto di esecuzione della JSP in termini di sessione utente

void setAttribute(String name, attributo) //
void setMaxInactiveInterval(int interval) // Setta l'intervallo massimo di inattività
String getID() // Restituisce ID di una sessione
boolean isNew() // Restituisce vero se la sessione è nuova
void invalidate() // Permette di invalidare (distruggere) una sessione
long getCreationTime() // Ci dice da quanto tempo è attiva la sessione (in ms)
long getLastAccessedTime() // Ci dice quando è stata utilizzata l'ultima volta
```

7. ***application***

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

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

8. ***pageContext***

   L'oggetto *pageContext* fornisce informazioni sul contesto di esecuzione della pagina JSP. Rappresenta l'insieme degli oggetti built-in di una JSP e consente l'accesso a tutti gli oggetti impliciti e ai loro attributi; consente il trasferimento del controllo ad altre pagine; viene poco usato in caso di scripting, è più utile per costruire custom tag.

9. ***exception***

```
/* rappresenta l'eccezione che non viene gestita da nessun blocco catch; non è disponibile in tutte le pagine ma solo nelle Erro
r Page (quelle dichiarate con l'attributo errorPage impostato a true) */

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

Azioni

Sono comandi che vengono tipicamente utilizzati per l'interazione con altre pagine JSP, Servlet, o componenti javaBean; sono espresse usando sintassi XML. Esistono 6 tipologie di azioni definite dai seguenti tag:

1. *useBean*: istanzia JavaBean e gli associa un identificativo;
2. *getProperty*: ritorna property indicata come oggetto;
3. *setProperty*: imposta valore della property indicata per nome;
4. *include*: include nella JSP contenuto generato dinamicamente da un'altra pagina locale, trasferisce temporaneamente il controllo ad un'altra pagina

```
<!— flush stabilisce se il buffer della pagina corrente debba essere flushed -->
<jsp:include page="localURL" flush="true">
<jsp:param name="parName1" value="parValue1"/>
<jsp:param name="parNameN" value="parValueN"/>
</jsp:include>
```

5. *forward*: consente il trasferimento del controllo dalla pagina JSP corrente ad un'altra pagina sul server locale; è possibile usare forward soltanto se non è stato emesso alcun output!

```
<jsp:forward page="localURL" />
<!-- E' possibile generare dinamicamente attributo page -->
<jsp:forward page='<%="message"+statusCode+".html"%>'/>
<!-- E' possibile specificare parametri -->
<jsp:forward page="localURL">
<jsp:param name="parName1" value="parValue1"/>
<jsp:param name="parNameN" value="parValueN"/>
</jsp:forward>
```

6. *plugin*: genera contenuto per scaricare plug-in Java se necessario.

Java Beans

Un Java bean è il modello di "base" per componenti Java, il più semplice. Non è altro che una classe Java dotata di alcune caratteristiche particolari:

- classe public;
- ha un costruttore public di default (senza argomenti);
- espone proprietà, sotto forma di coppie di metodi di accesso (*accessors*) costruiti secondo le regole get e set. Ad esempio `void setLenght(int Value)` e `int getLenght()` identificano proprietà lenght di tipo int; se definiamo solo il metodo get abbiamo una proprietà di sola lettura, se la proprietà è di tipo boolean bisognerà scrivere `void setEmpty(boolean value)` e boolean `isEmpty()`;
- espone eventi con metodi di registrazione che seguono regole precise.

Un oggetto bean si definisce così:

```
<jsp:useBean id="time" class="CurrentTimeBean" scope="page"/>
```

Lo scope di un oggetto bean può essere:

- *page (default)*: fino a quando la pagine viene completata o fino al forward;
- *request*: fino alla fine dell'elaborazione della richiesta e restituzione della risposta;
- *session*: tempo di vita della sessione;
- *application*: tempo di vita dell'applicazione.

Per quanto riguarda i metodi delle proprietà dei bean:

```
<jsp:getProperty name="beanId" property="propName"/>
// consente l'accesso alle proprietà del bean

<jsp:setProperty name="beanId" property="propName" value="propValue"/>
// consente di modificare il valore della proprietà del bean
```

Esempio di creazione di un bean:

```
public class CurrentTimeBean {
  private int hours;
  private int minutes;
  public CurrentTimeBean() {
    Calendar now = Calendar.getInstance();
    this.hours = now.get(Calendar.HOUR_OF_DAY);
```

```
    this.minutes = now.get(Calendar.MINUTE);
  }
  public int getHours() {
    return hours;
  }
  public int getMinutes() {
    return minutes;
  }
}
```

Custom tag

Le JSP permettono di definire tag personalizzati (custom tag) che estendono quelli predefiniti. Una taglib è una collezione di questi tag non standard, realizzata mediante una di classe Java:

```
<!-- La direttiva per includere la libreria di tag -->
<%@ uri="tagLibraryURI" prefix="tagPrefix"%>
<!-- Utilizzo: il prefisso definisce un namespace e quindi elimina eventuali omonimie date dallinclusione da più librerie-->
<htl:helloWorld who="Mario">
```

JavaScript

JavaScript è un linguaggio di scripting sviluppato per dare interattività lato client e alle pagine HTML. Permette di creare pagine Web attive e può essere inserito direttamente nel documento HTML; in pratica è lo standard "client-side" per implementare pagine attive.

Presenta le seguenti caratteristiche:

- è un linguaggio interpretato e non compilato;
- è object-based ma non class-based → esiste il concetto di oggetto e non esiste il concetto di classe;
- è debolmente tipizzato (weakly typed) → non è necessario definire il tipo di una variabile (ciò non significa che i dati non abbiano un tipo, sono le variabili a non averlo in modo statico)

La sintassi di JavaScript, come Java, è modellata su quella del C:

- è case-sensitive;
- le istruzioni sono terminate da ";", ma il terminatore può essere omesso se si va a capo;
- si commenta con "/* */" oppure monolinea "//";
- gli identificatori non possono iniziare con una cifra.

Il codice JavaScript viene eseguito da un interprete contenuto all'interno del browser. Uno script JavaScript viene inserito nella pagina HTML usando il tag `<script>` :

```
<html>
  <body>
    <p>Hello da JavaScript</p>
    <script>
        alert("Hello World!");
    </script>
  </body>
</html>
```

Variabili e tipi

Le **variabili** vengono dichiarate usando la parola chiave `var nomeVariabile;` e non hanno un tipo. 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`.

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

Per esprimere le costanti si utilizza la sintassi `literal`.

JavaScript prevede pochi **tipi primitivi**:

- numeri (*number*) → rappresentati sempre in formato *floating point*, esiste il valore `NaN` e `infinite`;
- boolean;

```
var v; // senza tipo
v = 15.7 // diventa di tipo number
v = true; // diventa di tipo boolean
```

E come **tipi di riferimento**:

- **Oggetti** → tipi composti che contengono un certo numero di proprietà (attributi), tali proprietà non sono definite a priori ma possono essere aggiunte dinamicamente

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

  Object() è un costruttore e non una classe, le classi non esistono!

  Le costanti oggetto (*object literal*) sono racchiuse fra parentesi graffe e contengono un elenco di attributi nella forma *nome:valore*

  ```
  var o = {x:7, y:8, tot:15};
  alert(o.tot);
  ```

- **Array** → tipi composti i cui elementi sono accessibili mediante un indice numerico (che parte da 0) e non hanno una dimensione prefissata (simili agli *ArrayList* di Java)

  ```
  var arr = new Array([dimensione]); // Dimensione è opzionale
  // Oppure
  var arr = [val, val2, ..., valN];
  ```

Le **stringhe** in JavaScript sono dati di tipo primitivo che sembrano oggetti (però non lo sono!). Sono sequenze arbitrarie di caratteri in formato UNICODE a 16 bit e sono immutabili come in Java. Ciò accade perchè sono gestite dall'oggetto wrapper String quando la variabile necessita di essere trattata come tipo di riferimento. Si possono concatenare con l'operatore + ed è possibile la comparazione con gli operatori di uguaglianza, disuguaglianza, maggioranza e minoranza.

```
var s = “ciao”;
varN = s.length;
varT = s.charAt(1);
```

Le ***regular expressions*** sono un tipo di dato nativo del linguaggio:

```
// costanti di tipo espressione regolare (regexp literal) con la sintassi:
/ expression /

// oppure creata mediante costruttore RegExp()
var r = /[abc]/; // Vuol dire qualsiasi carattere che sia a o b o c
var r = new RegExp("[abc]");
```

Per riassumere, numeri e booleani sono tipi valore (tipi primitivi), mentre array e oggetti sono tipi di riferimento (oggetti); le stringhe invece sono un caso particolare, perchè pur essendo un tipo primitivo si comportano come un tipo di riferimento.

Funzioni

Le funzioni in JavaScript ammettono parametri privi di tipo e restituiscono un valore il cui tipo non viene definito. Sono considerati oggetti e possono essere assegnate a una variabile, si definiscono così:

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

// per assegnarle ad una variabile invece costanti funzione: function literal
var sum = function(x,y) {
    return x + y;
}

// oppure create mediante il costruttore Function() (per questo sono equivalenti agli oggetti: function object
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, e si può usare la parola chiave `this` per accedere a tale oggetto:

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

Operatori e strutture di controllo

JavaScript ammette tutti gli **operatori** presenti in C e in Java, oltre ad alcuni operatori tipici come `delete` che elimina le proprietà di un oggetto, `void`, `typeof`, `===` ovvero identità stretta diverso da uguaglianza (puntano lo stesso oggetto) e non identità `!==`.

Per quanto riguarda le **strutture di controllo** ammette if/else, switch, do/while e for proprio come in C e Java. Vi è però un ulteriore operatore che permette di scorrere le proprietà di un oggetto (e quindi anche di un array) con la sintassi

```
for (variable in object) statement
```

JavaScript ed HTML

HTML prevede l'apposito tag `<script>` per eseguire codice JavaScript; il commento `<!-- codice -->` serve per gestire la compatibiltà con i browser, qualora non dovessero gestire JavaScript difatti il contenuto del tag viene ignorato, e per gestire contenuti alternativi in questo caso HTML prevede un tag `<noscript>`:

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

Esistono due tipi di script:

```
// script esterno
// il tag contiene il riferimento ad un file con estensione .js che contiene lo script
<script src="nomefile.js"></script>

// script interno
// contenuto direttamente nel tag
<script>
    alert("Hello World!");
</script>
```

Con JavaScript si possono fare essenzialmente 4 cose:

- costruire dinamicamente parti della pagina in fase di caricamento;
- rilevare informazioni sull'ambiente (tipo di browser, dimensione schermo, ..);
- rispondere ad eventi generati dall'interazione con l'utente;
- modificare dinamicamente il DOM (*Dynamic HTML* o *DHTML*)

Per quanto riguarda la **costruzione dinamica della pagina** bisogna tener presente che gli script contenuti nel corpo della pagina vengono eseguiti solo una volta durante il caricamento della pagina e quindi non si ha interattività con l'utente. L'uso più comune è invece quello di generare pagine diverse in base al tipo di browser o alla risoluzione dello schermo. La pagina corrente è rappresentata dall'oggetto *document* e per scrivere nella pagina si utilizzano `document.write()` o `document.writeln()`.

Per accedere alle **informazioni del browser** si utilizza l'oggetto *navigator*:

```
navigator.appCodeName // Nome in codice del browser (poco utile)
navigator.appName // Nome del browser
navigator.appVersion // Versione del browser
navigator.cookieEnabled // Dice se i cookies sono abilitati
navigator.platform // Piattaforma per cui il browser è stato compilato (es.
Win32)
navigator.userAgent // Stringa passata dal browser come header user-agent (es. "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1;)"
)

// un esempio:
<html>
    <body>
        <script>
            document.write('Hello '+navigator.appName+'!<br>');
            document.write('Versione: '+navigator.appVersion+'<br>');
            document.write('Piattaforma: '+navigator.platform);
        </script>
    </body>
</html>
```

Invece l'oggetto *screen* permette di ricavare informazioni sullo schermo:

```
<html>
    <body>
        <script>
            document.write('Schermo: '+screen.widht+'x'+screen.height+' pixel<br>');
    </body>
</html>
```

Modello ad eventi ed interattività

Per avere una reale interattività bisogna utilizzare il meccanismo degli **eventi**. JavaScript consente di associare script agli eventi causati dall'interazione dell'utente con la pagina HTML. L'associazione avviene mediante attributi collegati agli elementi della pagina HTML, e gli script prendono il nome di *event handlers*.

```
<tageventHandler="JavaScript Code">

// un esempio:
<input type="button" value="Calculate" onClick="alert("Calcolo")"/>

// è possibile inserire più istruzioni in sequenza, ma è meglio definire delle funzioni
```

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

Vedi → lista eventi

Esempio: calcolatrice

```
<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>
```

Il punto di partenza per accedere al *Document Object Model (**DOM**)* della pagina è l'oggetto *document*, che espone 4 collezioni di oggetti per gli elementi di primo livello:

1. *anchors[]*
2. *forms[]*
3. *images[]*
4. *links[]*

L'accesso agli elementi delle collezioni può avvenire per indice (`document.links[0]` o per nome `document.links["nomelink"]`.

Per quanto riguarda i metodi e le proprietà:

```
// 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'

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

Form

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

```
document:
document.nomeForm;
document.forms[n];
document.forms["nomeForm"];

// gli elementi del form vengono referenziati con il loro nome o col vettore elements[]
document.nomeForm.nomeElemento;
document.forms[n].elements[n];
document.forms["nomeForm"].elements["nomeElem"];
```

Per ogni elemento del form esistono i vari attributi *id, name, value, type, className, ecc.*

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

// o in alternativa possiamo scrivere
onclick = "this.form.text1.value = this.form.name">
```

Proprietà dei form:

- *action* → riflette l'attributo action;
- *elements* → vettore contenente gli elementi del form;
- *lenght* → numero di elementi del form;
- *method* → riflette l'attributo method;
- *name* → nome del form;
- *target* → riflette l'attributo target;

Metodi:

Eventi:

- *reset()* → resetta il form;
- *submit()* → esegue il submit.
- *onreset* → quando il form viene resettato;
- *onsubmit* → quando viene eseguito il sumbit.

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

```
<input type ="text">
<input type ="checkbox">
<input type ="radio">
<input type ="button"> // o in alternativa  <button>
<input type ="hidden">
<input type ="file">
<input type ="password">
<input type ="submit">
<input type ="reset">
<textarea>
```

Uno degli utilizzi più frequenti di JavaScript è nell'ambito della **validazione dei campi di un form**. Generalmente si valida un form in due momenti:

- durante l'inserimento utilizzando l'evento `onChange()` sui vari controlli;
- al momento del sumbit utilizzando l'evento `onClick()` del bottone sumbit o l'evento `onSubmit()` del form.

Alcuni esempi:

```
<head>
    <script type="text/javascript">
        function qty_check(item, min, max)
        {
            returnVal = false;
            if (parseInt(item.value) < min) or
                (parseInt(item.value) > max)
              alert(item.name+"deve essere fra "+min+" e "+max);
            else returnVal = true;
            return returnVal;
        }
        function validateAndSubmit(theForm)
        {
            if (qty_check(theform.quantità,0,999))
            { alert("Ordine accettato"); return true; }
            else
            { alert("Ordine rifiutato"); return false; }
        }
    </script>
</head>
<body>
    <form name="widget_order"
      action="lwapp.html" method="post">
        Quantità da ordinare
        <input type="text" name="quantità"
            onchange="qty_check(this,0,999)">
        <br/>
        <input type="submit" value="Trasmetti l'ordine"
          onclick="validateAndSubmit(this.form)">
    </form>
</body>

/* oppure in alternativa
<form name="widget_order"
  action="lwapp.html" method="post"
    onSubmit="return qty_check(this.quantita,0,999)">
    ...
  <input type="submit" />
    ...
</form>  */
```

```
<head>
  <script>
    function upperCase()
    {
        var val = document.myForm.firstName.value;
        document.myForm.firstName.value = val.toUpperCase();
        val = document.myForm.lastName.value;
        document.myForm.lastName.value = val.toUpperCase();
    }
  </script>
</head>
<body>
  <form name="myForm">
    <b>Nome: </b>
        <input type="text" name="firstName" size="20" /><br/>
    <b>Cognome: </b>
        <input type="text" name="lastName" size="20"/>
    <p><input type="button" value=" Maiuscolo"
        onClick="upperCase()"/></p>
  </form>
<body>
```

AJAX

Applicazioni Web tradizionali espongono un modello di interazione rigido (*click*, *wait*, and *refresh*) sincrono: l'utente effettua una richiesta e deve attendere la risposta da parte del server. Il modello di interazione **AJAX** (**A**synchronous **J**avascript **A**nd **X**ml) è un'estensione di JavaScript nata per superare queste limitazioni permettendo di definire un **modello asincrono**. L'elemento centrale è l'utilizzo dell'oggetto JavaScript `XMLHttpRequest` che consente di ottenere dati dal server senza necessità di ricaricare l'intera pagina.

Tipica sequenza AJAX:

1. si verifica un evento determinato dall'interazione utente-pagina web;
2. l'evento fa eseguire una funzione JavaScript in cui:
   - si istanzia un oggetto `XMLHttpRequest` e si configura
   - si effettua chiamata asincrona al server;
3. il server elabora la richiesta e risponde al client;
4. il browser invoca la funzione di callback che elabora il risultato e aggiorna il DOM.

Oggetto *XMLHttpRequest*

Questo oggetto effettua richieste di una risorsa via HTTP a server Web. Può effettuare sia richieste *GET* che *POST* che possono essere di tipo:

- **sincrono** → blocca flusso di esecuzione del codice JavaScript
- **asincrono** → NON interrompe il flusso di esecuzione dello script né le operazioni dell'utente sulla pagina → **quindi thread dedicato**;

I browser recenti supportano `XMLHttpRequest` come oggetto nativo, mentre la lista dei metodi è diversa da browser a browser (in genere useremo quelli presenti in Safari, comuni a tutti i browser)

```
var xhr = new XMLHttpRequest();

// METODI:

open(method, uri, true); // inizializza la richiesta da formulare al server - method può assumere il valore "get" o "post"

setRequestHeader(nomeheader, valore); // consente di impostare gli header HTTP della richiesta da inviare
// viene invocata per ogni header da impostare
```

```
// per una richiesta GET gli header sono opzionali, invece obbligatori per le POST
// impostare header connection di solito al valore close

send(body); // consente di inviare la richiesta al server

// ESEMPI:

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

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

Stato e risultati della richiesta vengono memorizzati all'interno dell'oggetto stesso durante la sua esecuzione.

Le **proprietà** comunemente supportate dai vari browser sono:

- *readyState* → proprietà in sola lettura di tipo intero che consente di leggere in ogni momento lo stato della richiesta → ammette 5 valori ovvero `0: uninitialized` `1:open` `2:sent` `3:receiving` `4:loaded`;
- *onreadystateChange* → come si è detto l'esecuzione del codice non si blocca sulla `send()` in attesa dei risultati, quindi per gestire la risposta si deve adottare un approccio ad eventi → registrare una funzione di callback che viene richiamata in modo asincrono ad ogni cambio di stato della proprietà

  ```
  var xhr = new XMLHttpRequest();
  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 (200 in caso di successo);
- *statusText* → contiene una descrizione testuale del codice HTTP restituito dal server;

  ```
  // esempio
  if ( xhr.status != 200 )
    alert( xhr.statusText );
  ```

- *responseText* → body della risposta HTTP, disponibile solo a interazione ultimata (`readystate==4`);
- *responseXML* → body della risposta convertito in XML (se possibile) → consente navigazione via javascript → può essere null;

I **metodi di ricezione della risposta** sono nella funzione di callback e possono essere invocati in modo safe solo a richiesta conclusa (`readystate==4`).

- *getAllResponseHeaders();*
- *getResponseHeader(header_name);*

Vantaggi e svantaggi

> si guadagna in espressività, ma si perde la linearità dell'interazione

Le richieste AJAX permettono all'utente di continuare a interagire con la pagina ma non necessariamente lo informano di cosa stia succedendo, e possono durare troppo. Di solito si agisce su due fronti:

- rendere visibile l'andamento della chiamata → barre di scorrimento, info utente..
- interrompere le richieste che non terminano in tempo utile per sovraccarichi del server o momentanei timeout → metodo `abort()` → consente l'interruzione delle operazioni di invio o ricezione.

  > Attenzione: non ha senso invocare `abort()` dentro la funzione di callback → se *readyState* non cambia, il metodo non viene richiamato e *readyState* non viene modificato quando la risposta si fa attendere

  Si crea un'altra funzione da far richiamare in modo asincrono al sistema mediante il metodo `setTimeOut(funzioneAsincronaPerAbortire, timeout)`

  Un esempio fatto con Ajax:

  Scegliamo un nome da una lista e mostriamo i suoi dati

```
<html>
  <head>
    <script src="selectmanager_xml.js"></script>
  </head>
  <body>
    <form action=""> Scegli un contatto:
    <select name=”manager“
      onchange="showManager(this.value)">
      <option value=”Carlo11">Carlo Rossi</option>
      <option value=”Anna23">Anna Bianchi</option>
      <option value=”Giovanni75">Giovanni Verdi</option>
    </select></form>
    <b><span id="companyname"></span></b><br/>
      <span id="contactname"></span><br/>
      <span id="address"></span>
      <span id="city"></span><br/>
      <span id="country"></span>
  </body>
</html>
```

  Ipotizziamo che i dati sui contatti siano contenuti in un database → server riceve la request, legge e restituisce file XML con i dati richiesti

  selectmanager_xml.js

```
var xmlHttp;
function showManager(str) {
  xmlHttp=new XMLHttpRequest();
  var url="getmanager_xml.jsp?q="+str;
  xmlHttp.onreadystatechange=stateChanged;
  xmlHttp.open("GET",url,true);
  xmlHttp.send(null);
}
function stateChanged() {
  if (xmlHttp.readyState==4) {
    var xmlDoc=xmlHttp.responseXML.documentElement;
    var compEl=xmlDoc.getElementsByTagName("compname")[0];
    var comName = compEl.childNodes[0].nodeValue;
    document.getElementById("companyname").innerHTML=
    compName;
    ...
  }
}
```

Dall'esempio però si vede come l'utilizzo di XML come formato di scambio client-server porta a generazione e utilizzo di quantità di byte piuttosto elevate e non ottimizzate → esiste una soluzione più efficiente? Si

JSON

JSON è l'acronimo di (**J**ava**S**cript **O**bject **N**otation) è un formato per lo scambio di dati molto più comodo di XML perchè:

- è leggero in termini di quantità di dati scambiati;
- semplice ed efficiente da elaborare da parte del supporto runtime al linguaggio di programmazione;

- semplice da leggere per il programmatore

La sua sintassi si basa sulla notazione usata per le costanti oggetto (*object literal*) e le costanti array (*array literal*) in JavaScript:

```
// in Javascript è possibile creare un oggetto in base a una costante oggetto
var Beatles = {
  "Paese" : "Inghilterra",
  "AnnoFormazione" : 1959,
  "TipoMusica" : "Rock"
}
// del tutto equivalente a:
var Beatles = new Object();
Beatles.Paese = "England";
Beatles.AnnoFormazione = 1959;
Beatles.TipoMusica = "Rock";

// analogamente per gli array
var Membri = ["Paul","John","George","Ringo"];
// che equivale a
var Membri = new Array("Paul","John","George","Ringo");

// possiamo quindi avere anche oggetti che contengono array
var Beatles = {
  “Paese" : “Inghilterra",
  “AnnoFormazione" : 1959,
  “TipoMusica" : "Rock",
  "Membri" : ["Paul","John","George","Ringo"]
}

// ed infine array di oggetti
var Rockbands = [
    {
        "Nome" : "Beatles",
        “Paese" : “Inghilterra",
        “AnnoFormazione" : 1959,
        “TipoMusica" : "Rock",
        "Membri" : ["Paul","John","George","Ringo"]
    },
    {
        "Nome" : "Rolling Stones",
        “Paese" : “Inghilterra",
        “AnnoFormazione" : 1962,
        “TipoMusica" : "Rock",
        "Membri" : ["Mick","Keith","Charlie","Bill"]
    }
]
```

Un **“oggetto JSON”** non è altro che una stringa equivalente a una costente oggetto di JavaScript → la sintassi di JSON è un sottoinsieme di JavaScript → tramite la funzione `eval()` possiamo trasformare una stringa JSON in un oggetto → la sintassi della stringa passata a `eval()` deve essere `'(espressione)'`: dobbiamo quindi racchiudere la stringa JSON fra parentesi tonde

```
// stringa JSON
var s ='{ "Paese" : "Inghilterra",
"AnnoFormazione" : 1959, "TipoMusica" : "Rock",
"Membri" : ["Paul","John","George","Ringo"]}';

var o = eval('('+s+')');
```

Esempio completo:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
```

```
  <head>
    <title> Esempio JSON </title>
    <script>
        var s ='{ "Paese" : "Inghilterra", "AnnoFormazione": 1959,
                    "TipoMusica" : "Rock", "Membri" :["Paul","John","George","Ringo"]}';
        var o = eval('('+s+')');
    </script>
  </head>
  <body>
    <p onclick='alert(o.Paese)'>
      Clicca
    </p>
  </body>
</html>
```

All'uso di `eval()` però di solito è preferibile utilizzare parser appositi che traducono solo oggetti JSON:

- *Google GSON* - https://github.com/google/gson;
- *jabsorb* - https://github.com/Servoy/jabsorb;

JSON e AJAX

Ad esempio in una interazione client-server in cui il cliente vuole trasferire un oggetto JSON:

Lato client

1. si crea un oggetto JavaScript e si riempiono le sue proprietà
2. si usa `JSON.stringify()` (metodo del parser jabsorb) per convertire l'oggetto in stringa JSON
3. si usa la funzione `encodeURIComponent()` per convertire la stringa in formato utilizzabile in una richiesta HTTP
4. si manda la stringa al server tramite `XMLHttpRequest`

Dopo la ricezione:

10. si converte stringa JSON in un oggetto JavaScript usando `JSON.parse()`
11. si usa l'oggetto

Lato server

5. si decodifica la stringa JSON e la si trasforma in oggetto Java con apposito parser (sempre su www.json.org)
6. si elabora l'oggetto
7. si crea un nuovo oggetto Java che contiene i dati della risposta
8. si trasforma l'oggetto Java in stringa JSON usando il suddetto parser
9. si trasmette stringa JSON al client nel corpo della risposta HTTP `response.out.write(strJSON);`

Volendo utilizzare *GSON* al posto di *jabsorb*

```
// inizializzazione dell'oggetto Gson
Gson g = new Gson();

// serializzazione di un oggetto
Person santa = new Person("Santa", "Claus", 1000);
g.toJson(santa);

// deserializzazione
Person peterPan = g.fromJson(json, Person.class);
```

Riassumendo

- AJAX consente gestione asincrona;

- Aggiunge un nuovo elemento al modello a eventi;
- L'uso di `XmlHttpRequest` rappresenta una modalità alternativa per gestire gli eventi remoti -> abbiamo quindi
  - Una modalità per gestire gli eventi a livello locale
  - Due modalità per gestire gli eventi remoti (postback e XmlHttpRequest)
- Si può utilizzare solo AJAX per eliminare i caricamenti di pagina (*Single Page Applications*)

React.js

React.js è una libreria javascript per la creazione di interfacce utente Web e rientra tra gli strumenti utili per il cosiddetto sviluppo "**front-end**" di web application. Essendo costruito sul linguaggio javascript, qualsiasi codice scritto in React.js **esegue all'interno del browser**. Come altre tecnologie front-end, React.js permette di invocare anche API lato server; React NON interagisce direttamente con database o qualsiasi altra sorgente dati che si trovi su back-end. Può interagire con alcune tecnologie di back-end come Python/Flask, Ruby on Rails, Java/Spring, PHP ed altre.

React.js si ispira alle interfacce utenti del tipo "***Single Page Application***" (**SPA**), ovvero applicazioni web che interagiscono col browser per *modificare* pagine web in modo dinamico in funzione dei dati che arrivano dal back-end; la SPA può infatti essere vista come un contenitore all'interno del quale la pagina web evolve dinamicamente. I componenti di una pagina web interagiscono con le API della libreria React.js che, a loro volta, manipolano il DOM per creazione di elementi di interfaccia utente.

Solo che anzichè manipolare il DOM del browser, React manipola un ***virtual DOM*** che è una copia esatta di quello del browser, ma rende più leggero e veloce il processo di rendering della pagina, effettuando sul DOM del browser solo le modifiche strettamente necessarie.

Esempio con ***React Element***:

```
<!DOCTYPE html>
<html>
  <head>
      <meta charset="UTF-8" />
      <title>Primi passi con React</title>
      <script src="https://unpkg.com/react@15/dist/react.js"></script>
      <script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.24/browser.js"></script>
  </head>
  <body>
      <div id="root"></div>
      <script type="text/babel">
        const elem = <p>Hello <strong>React</strong>!</p>;
        <!-- sintassi JSK (Javascript XML) -->
        ReactDOM.render(elem, document.getElementById('root'));
        <!-- è stato definito un React Element e successivamente è stato chiesto al DOM
         del browser di visualizzare l'elemento in una specifica posizione attraverso
         l'istruzione ReactDOM.render -->
      </script>
  </body>
</html>
```

Un React element è un oggetto semplice e immutabile che descrive cosa si vuole visualizzare sullo schermo.

Esempio con ***React Component***:

```
<!DOCTYPE html>
<html>
  <head>
```

```
      <meta charset="UTF-8" />
      <title>Primi passi con React</title>
      <script
      src="https://unpkg.com/react@15/dist/react.js"></script>
      <script src="https://unpkg.com/react-dom@15/dist/reactdom.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/babelcore/5.8.24/browser.js"></script>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/babel">
    class HelloWorld extends React.Component{
      render() { return <p>Hello <strong>React</strong>!</p>;}
    };
    ReactDOM.render(<HelloWorld />,
    document.getElementById('root'));
    </script>
  </body>
</html>
```

I React Component sono oggetti complessi e dinamici, che ricevono input dall'esterno e forgiano l'elemento grafico da restituire. Concettualmente sono simili alle funzioni JavaScript: possono accettare in input dati arbitrari (sotto il nome di *props*) e restituiscono elementi React che descrivono cosa dovrebbe apparire sullo schermo.

Il risultato finale sul browser è identico a quello dell'esempio precedente.

Linguaggio JSX

Il linguaggio JSX (JavaScript XML) permette di scrivere tag HTML all'interno di codice JavaScript e piazzarlo all'interno del DOM senza l'uso di metodi quali `createElement()` e/o `appendChild()`.

```
<!-- negli esempi precedenti: -->
const elem = <p>Hello <strong>Reacy</strong>!</p>;
return <p>Hello<strong>React</strong>!</p>;

<!-- un altro esempio di JSX -->
const myelement = <h1> I love JSX! </h1>;
ReactDOM.render(myelement, document.getElementById('root'));

<!-- creazione di una lista -->
const listElement = <ul className="list-of-items">
    <li className="item-1" key="key-1">Item 1</li>
    <li className="item-2" key="key-2">Item 2</li>
    <li className="item-3" key="key-3">Item 3</li>
  </ul>

ReactDOM.render(listElement, document.getElementById("container"));

<!-- uso di variabili, costanti e funzioni -->
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 -->
```

Il browser ovviamente non è in grado di interpretare nativamente i costrutti JSX; è necessario aggiungere all'interno della pagina un riferimento a un pre-compilatore in grado di trasformare JSK in linguaggio javascript → **Babel**:

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

React Components

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 reagire (react) agli eventi che si verificano sulla pagina e aggiornare se stessa di conseguenza.

Ogni “mattoncino” ha un ruolo ben definito per ciò che rappresenta graficamente e gestisce le interazioni utente su quella particolare sezione di interfaccia.

I componenti sono pezzi di codice indipendenti e riusabili → richiamano il concetto di funzioni in javascript. Esistono due tipi di componenti:

- Componenti di tipo *“class”*

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

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

- Componendi di tipo *“function”*

```
// Il nome del componente deve cominciare con la lettera
// maiuscola, altrimenti React lo tratta come un normale tag HTML
function Car() {
    return <h2>I am a Car!</h2>;
}
// la funzione deve restituire l'elemento di cui fare il renderin
g
// attraverso il return

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

Sia per le funzioni che per le classi è possibile specificare delle proprietà (***props***) ed assegnare a queste determinati valori, immutabili per i quali non è prevista alcuna alterazione → utili per configurare il componente.

```
// uso delle props nelle funzioni
function Car(props) {
  return <h2>I am a {props.colore} Car!</h2>;
}
ReactDOM.render(<Car colore="red"/>, document.getElementById('root'));

// uso di props nelle classi
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'));
```

Per la creazione della classe è anche possibile utilizzare il metodo factory:

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

Tutti i componenti di tipo classe possiedono un oggetto built-in che prende il nome di ***state*** → a differenza delle props, le proprietà definite nell’oggetto state NON sono immutabili → pensato per contenere proprietà che variano nel tempo → quando una delle proprietà cambia valore, il componente viene ri-renderizzato.

Componenti *function* → props immutabili → stateless (non hanno state)

Componenti *class* → props, e state (NON immutabili) → anche metodo factory

Analogamente a tutti i linguaggi ad oggetti, anche per il componente *class* è possibile definire un costruttore:

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {brand: "Ford", model: "Mustang", color: "red", year:1964};
  }
  // è necessario invocare il metodo super() per usare this
  render() {
    return (
      <div>
      <h1>My {this.state.brand}</h1>
      <p> It is a {this.state.color} {this.state.model} from {this.state.year} </p>
      </div>
    );
  }
}
ReactDOM.render(<Car />, document.getElementById('root'));
```

All'interno di un componente è possibile fare riferimento ad altri componenti:

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

class Garage extends React.Component { // componente "contenitore"
  render() {
    return (
      <div>
      <h1>Who lives in my Garage?</h1>
      <Car />
      </div>
    );
  }
}
// è sufficiente fare il rendering del componente contenitore -> react farà il rendering dei componenti contenuti
ReactDOM.render(<Garage />, document.getElementById('root'));
```

Per aggiornare l'oggetto *state* si utilizza la funzione `setState()` che farà modificare lo stato alla libreria React che re-invocherà la funzione `render()`.

Esempio → 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() {
    // modifica del valore della prop numeroEstratto
    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>;
      // react element che farà da display
    }
    return (
      <div className="card" >
      <p className="card__number">{valore}</p>
      <button className="card__button" onClick={() => this.lanciaDado()}>Lancia il Dado</button>
      </div>
      // sull'evento onClick la funzione che gestisce l'evento viene invocata in modalità arrow
    )
  }
}
```

Non tutti i componenti devono avere uno state, anzi è consigliato usare componenti stateless.

Applicazione React → gerarchia di componenti → componenti ai vertici mantengono lo state dell'applicazione → passano le props ai figli

Esempio: rubrica con una serie di contatti → cercare i contatti tramite un campo di ricerca → filtrare la lista dei contatti in base al testo digitato → due componenti → *SearchBar* e *ContactList* → contenuti nel componente più esterno *App* → *App* aggiornerà lo state con `setState()` → passerà il nuovo valore di state ai figli tramite props → quando lo riceveranno verrà invocata `render()`

Gestione degli eventi

```
class App extends React.Component {
  constructor(props) {
    super(props);
  }
  // handler dell'evento
  handleClick() {
    console.log('Pulsante premuto - Evento ${e.type}');
    // il parametro e è un evento sintetico
  }

  render() {
    return (
      // sintassi JSX, event handler invocato come stringa tra graffe
      <button onClick={this.handleClick} >Pulsante</button>
    )
  }
}
```

> Attenzione: se l'handler dell'evento deve fare accesso allo *state* del componente, occorre apportare accorgimenti al codice di gestione dell'evento.

Per una corretta invocazione dell'handler dell'evento, occorre che l'oggetto che lo invoca sia il componente → keyword `this` con due opzioni:

- all'interno del costruttore, forzare `bind` di `this` del metodo a `this` del componente
- invocare l'handler come una arrow function → opzione più usata

```
// esempio prima opzione

class Interruttore extends React.Component {
  constructor(props) {
    super(props);
    this.state = {acceso: true};
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    // Al “this” dell’handler viene assegnato il “this” del componente
    this.setState({acceso: !this.state.acceso}
    // in alternativa
    // this.setState(state => ({acceso: !state.acceso}));
  }
  render() {
    return (
      // Event handler invocato come stringa
      <button onClick={this.handleClick}>
      {this.state.acceso ? 'Acceso' : 'Spento'}
      </button>
    );
  }
}

// esempio seconda opzione
class Interruttore extends React.Component {
  constructor(props) {
    super(props);
    this.state = {acceso: true};
    handleClick() {
      this.setState({acceso:!this.state.acceso}
      // in alternativa
      // this.setState(state => ({acceso: !state.acceso}));
    }
  render() {
    return (
    // Invocazione dell'handler tramite arrow function
    <button onClick ={()=>this.handleClick()}>
    {this.state.acceso ? 'Acceso' : 'Spento'}
    </button>
    );
  }
}
```

Form

In React, gli elementi HTML di cui è composto un form funzionano in modo diverso: gli elementi form mantengono naturalmente uno stato interno → <*input*>,<*textarea*> e <*select*> mantengono e aggiornano il proprio stato in base all’input dell’utente → in React lo stato mutabile viene mantenuto dallo *state* del componente e aggiornato solo mediante `setState()`.

```
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();
    // previene l'esecuzione del comportamento predefinito
    // es. apertura nuova pagina
  }

  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 risorse su server

In React si possono effettuare HTTP request in diversi modi. Uno elegante (e dal semplice utilizzo) fa uso delle Fetch API fornite nativamente da JavaScript.

Le Fetch API forniscono un'interfaccia JavaScript per accedere e manipolare parti della pipeline HTTP (ad esempio richieste e risposte) e mettono anche a disposizione un metodo che fornisce un modo semplice e logico per recuperare le risorse in modo asincrono.

Esempio: composizione di una request HTTP POST all'interno di un event handler

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

  handleSubmit(event) {
    event.preventDefault();
    const data = new
    FormData(event.target);
    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" />
      <button>Send data!</button>
      </form>
    );
  }
}
```

Librerie e framework alternativi

▼ **JQuery**

La libreria più utilizzata e conosciuta dagli sviluppatori, semplifica molto la gestione degli elementi DOM: selettori CSS3 e gestione semplificata delle richieste Ajax; codice compatibile con tutti i browser, molti plugin, estensione jQuery UI con drag&drop.

▼ **Angular**

Creato e mantenuto da Google, successore di AngularJS, serve per realizzare Single Page Application; implementa il pattern MVVM (Model-View-ViewModel) e si basa su jQuery Lite; non utilizza Javascript, bensì TypeScript, linguaggio di programmazione sviluppato da Microsoft basato su Javascript.

▼ **Vue.js**

Framework js per lo sviluppo di SPA, adotta il pattern MVVM, pensato come strumento più facile per i principianti, ciò a discapito della completezza di funzionalità.

▼ **Meteor**

Consente agli sviluppatori di creare con lo stesso codice sia applicazioni Web che mobile; le modifiche possono essere inoltrate direttamente ai client grazie al protocollo proprietario Distributed Data Protocol (DDP); funziona su una base Node.js pertanto può essere impiegato sia per sviluppo front-end che back.end.

▼ **Backbones**

Non è un vero e proprio framework ma un ottimo strumento per modellare e strutturare il codice; non fornisce un framework completo ma va abbinato ad altre librerie come underscore.js e jquery; nato per sviluppare SPA e adotta il patter Model-View-Presenter (MVP).

MVC e Java Model

L'**architettura model-view-controller** è adatta per applicazioni web interattive ed è composta da:

- **Model:** rappresenta il livello dei dati, incluse operazioni per accesso e modifica. Deve notificare view associate quando il modello viene modificato e deve supportare:
  - possibilità per view di interrogare stato di model
  - possibilità per controller di accedere alle funzionalità incapsulate dal model;
- **View:** si occupa del rendering dei contenuti del model. Accede ai dati e specifica come essi debbano essere rappresentati (di solito implementata da JSP):
  - aggiorna la presentazione dei dati quando il modello cambia
  - gira input dell'utente verso il controller
- **Controller:** definisce il comportamento dell'applicazione (di solito implementato da servlet o da EJB Session Bean, ecc..)
  - fa dispatching di richieste utente e seleziona view per presentazione
  - interpreta input utente e lo mappa su azioni che devono essere eseguite da model (in una GUI stand-alone, input come click e selezione menu; in una applicazione web richieste http get/post);

Il **Java Model 1** e il **Java Model 2** sono due modelli di sviluppo di applicazioni Java.

Il Java Model 1 è un modello di sviluppo di applicazioni Java che prevede l'utilizzo di servlets e JSP (JavaServer Pages) per la creazione di applicazioni web dinamiche. In questo modello, le servlets sono utilizzate per gestire le richieste del client e le JSP per generare il contenuto HTML da visualizzare nel browser del client.

Il Java Model 2, anche noto come modello di sviluppo MVC (Model-View-Controller), è un modello di sviluppo di applicazioni Java che prevede l'utilizzo di componenti separati per la gestione dei dati (modello), la visualizzazione dei dati (vista) e la gestione delle richieste del client (controller). In questo modello, il controller gestisce le richieste del client, il modello gestisce i dati e la vista visualizza i dati al client.

Entrambi i modelli sono stati ampiamente utilizzati per lo sviluppo di applicazioni web dinamiche in Java, ma il Java Model 2 è stato progettato per offrire una maggiore separazione dei componenti e una maggiore flessibilità nello sviluppo di applicazioni complesse.

EJB

Enterprise JavaBeans (EJB) è una tecnologia Java che fornisce un framework per la creazione di componenti distribuiti per applicazioni aziendali. Gli EJB sono utilizzati principalmente per lo sviluppo di applicazioni enterprise con un alto livello di affidabilità e scalabilità.

Gli EJB sono componenti di software che possono essere utilizzati all'interno di un'applicazione Java per gestire le attività di business, come ad esempio l'accesso ai dati, la gestione della transazione, la gestione della sicurezza e altre funzionalità comuni alle applicazioni enterprise. Gli EJB vengono eseguiti in un container EJB, che fornisce servizi di runtime per la gestione delle risorse, la gestione delle transazioni e la gestione della sicurezza.

I principali componenti di EJB sono:

1. Session Bean: è un componente che viene utilizzato per eseguire attività di business a livello di sessione, come ad esempio il recupero di dati da un database o il processing di un ordine. Non sono persistenti, hanno vita tipicamente breve e vengono persi in caso di failure di EJB. Esistono due tipi di session bean: stateful e stateless.
2. Message-Driven Bean: è un componente che viene utilizzato per gestire messaggi asincroni inviati a una coda JMS (Java Message Service); non possono essere invocati direttamente dai clienti ma si attivano alla ricezione di un messaggio. Sono privi di stato.
3. Entity Bean: è un componente EJB che rappresenta un oggetto di dominio, come ad esempio un cliente o un ordine, e viene utilizzato per gestire l'accesso ai dati in un database. Sono persistenti e long-lived, nella maggior parte dei casi sincronizzati con i relativi database relazionali; vi è un accesso condiviso per clienti differenti. Esistono due tipi di entity bean: container-managed e bean-managed.
4. Interceptor: sono componenti che possono essere utilizzati per eseguire il codice prima o dopo un metodo di una session bean o di un entity bean. Gli interceptor possono essere utilizzati per gestire la logica di business comune a diverse sessioni o entity bean.
5. Container EJB: è un componente di runtime che fornisce servizi di gestione delle risorse, gestione delle transazioni e gestione della sicurezza per gli EJB. Il container EJB viene utilizzato per eseguire gli EJB e fornire loro l'accesso ai servizi di runtime.

Differenza tra Session Bean ed Entity Bean:

Session Bean → rappresenta un processo di business; una istanza per cliente, short-lived (vita del bean = vita del cliente); transient; non sopravvive a crash del server; può avere proprietà transazionali.

Entity Bean → rappresenta dati di business; istanza condivisa fra clienti multipli, long-lived (vita del bean = vita dei dati nel database); persitente; sopravvive a crash del server; sempre transazionale.

I bean di tipo sessione e message-driven sono classi Java ordinarie.

Il tipo di bean viene specificato da una annotation:

- `@Stateless`
- `@Stateful`
- `@MessageDriven`

```
// EJB3.0 Stateless Session Bean: Bean Class
@Stateless
public class Payroll Bean implements Payroll {
  public void setTaxDeductions(int empId, int deductions) {
  ...
  }
}
// EJB 3.0 Stateless Session Bean: Interfaccia remota
@Remote
```

```
public interface Payroll {
  public void setTaxDeductions(int empId, int deductions);
}
// EJB 3.0 Stateless Session Bean:
// Alternativa: @Remote annotation applicata direttamente alla classe bean
@Stateless @Remote
public class PayrollBean implements Payroll {
  public void setTaxDeductions(int empId, int deductions)
  {...}
}
```

Stateless Session Bean

Ogni EJB container mantiene un insieme di istanze del bean pronte per servire le richieste client. Non esiste stato di sessione da mantenere tra richieste successive, infatti ogni invocazione di metodo è indipendente dalle precedenti.

Resource Pooling: pooling dei componenti server-side da parte del container EJB (instance pooling). L'idea di base è di evitare di mantenere un'istanza separata di ogni EJB per ogni cliente, e si applica a stateless session bean e a message-driven bean.

Ciclo di vita:

- No state: non istanziato, stato iniziale 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.

L'istanza del bean nel pool riceve un riferimento a `javax.ejb.EJBContext`. EJBContext fornisce un'interfaccia per il bean per comunicare con l'ambiente EJB. Quando il bean passa in stato ready, EJBContext contiene anche informazioni sul cliente che sta utilizzando il bean, oltre che il riferimento al proprio EJB stub (utile per passare riferimenti ad altri bean).

> Ricordiamo che il fatto di essere stateless è dato semplicemente tramite l'annotazione

Analogamente i Message-driven Bean non mantengono stato della sessione e il container effettua pooling in maniera analoga. L'unica differenza è che ogni EJB container contiene molti pool, ciascuno dei quali composto da istanze con la stessa destination JMS (Java message service)

Stateful Session Bean

Nel caso di stateful session bean, l'**activation** permette una gestione della coppia *oggetto EJB + istanza di bean stateful*, attraverso le due operazioni:

- Passivation → dissociazione tra stateful bean instance e suo oggetto EJB, con salvataggio dell'istanza su memoria (serializzazione); processo del tutto trasparente per il cliente.
- Activation → recupero della memoria (deserializzazione) dello stato dell'istanza e riassociazione con l'oggetto EJB; la procedura di activation può essere associata anche all'invocazione di metodi di callback sui cambi di stato nel ciclo di vita di uno stateful session bean.

Abbiamo quindi le due annotation:

- `@javax.ejb.PostActivate` → associa l'invocazione del metodo a cui si applica immediatamente dopo l'attivazione di un'istanza;
- `@javax.ejb.PrePassivate` → associa l'invocazione del metodo a cui si applica prima dell'azione di passivation

Sono spesso utilizzate per la chiusura/apertura di connessioni a risorse per gestione più efficiente.

> Per definizione i session bean non possono essere concorrenti, una singola istanza è associata ad un singolo cliente. Niente thread o keyword synchronized.

Un'esempio di Session Bean:

- OperationsBean → stateless session bean contenente la definizione di somma, divisione, ec..
- CalculatorBean → stateful session bean che
  1. seleziona operazione da svolgere in base a parametri forniti
  2. effettua l'operazione richiesta (non direttamente ma demandando ad un altro componente)
  3. mantiene il risultato parziale delle operazioni

Spring

Spring è una piattaforma di sviluppo di applicazioni Java che fornisce un insieme di librerie e strumenti per facilitare lo sviluppo di software basato su Kava Enterprise Edition; è un modello a container leggero (core su cui si agganciano solo i moduli di interesse) e fornisce 4 funzionalità chiave:

- **Inversion of Control (IoC) o Dependency Injection (DI):**

  Funzionalità che permette ai componenti di non dover cercare a runtime le risorse necessarie → dichiarano le risorse di cui hanno bisogno e il container si occupa di passarle al componente (*inversion of control*). Ciò rende il sistema disaccoppiato e manutenibile. Il container Spring usa la Dependency Injection per gestire i componenti che costruiscono l'applicazione, chiamati Spring Beans.

  L'idea fondamentale è quella di una factory per componenti utilizzabile globalmente che si occupa del ritrovamento di oggetti per nome e della gestione delle relazioni tra essi.

  Per far in modo che Spring inietti all'interno del componente una risorsa è necessario definire un costruttore oppure i metodi get/set della risorsa:

  ```
  public class SetterInjection {
    private Dependency dep;
    public void setMyDependency (Dependency dep) {
        this.dep = dep;
    }
  }
  ```

  Per utilizzare il componente definito, il client dovrà passare attraverso la Bean Factory → solitamente creata dall'applicazione nella forma di `XmlBeanFactory` leggendo un file di configurazione XML associato:

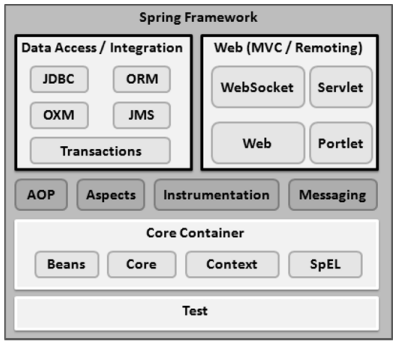
  ```
  XmlBeanFactory f = new XmlBeanFactory(new FileSystemResourse("bean.xml"));
  SomeBeanInterface b = (SomeBeanInterface) f.getBean("nameOfTheBean");
  ```

  Alcuni dei benefici principali dall'adozione di Dependency Injection sono:

  - flessibilità → va implementata solo la logica di business
  - possibilità e facilità di testing → senza bisogno di cambiare il sorgente
  - manutentibilità → possibilità di riutilizzo in diversi ambienti applicativi

- **Supporto alla persistenza:** offre un livello di astrazione generico per la gestione delle transazioni con DB (senza per forza dover lavorare dentro un EJB container)
- **Integrazione con Web tier:** offre un framework Model-View-Controller per applicazioni web, costruito sulle funzionalità base di Spring, con supporto per diverse tecnologie per la generazione di viste (oltre che Web Flow per navigazione a grana fine).
- **Aspect Oriented Programming:** tecnica che permette di separare in moduli diversi i vari aspetti dell'applicazione in modo da aumentarne la modularità e quindi la facilità di testing.

Architettura di Spring

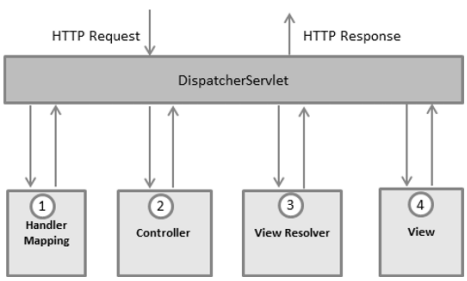


- **Core package:** parte fondamentale del framework, container leggero che si occupa di caricare le risorse dichiarate dai componenti e passarle ai componenti stessi.
- **MVC package:** ampio supporto a progettazione e sviluppo seguendo l'architettura Model-View-Controller per web app.

  → Spring offre supporto a componenti "controller", responsabili per interpretare la richiesta utente e interagire con l'applicazione → una volta che il controller ha ottenuto i risultati (model) decide a quale view fare forwarding del model → la view presenta all'utente i dati del model.
- **DAO package:** livello di astrazione che non rende più necessario codice ripetitivo per JDBC (Java Database Connectivity → API java che definisce come un client può accedere a un database) nè parsing di codici di errore database-specific.
- **ORM package:** livello di integrazione con soluzioni diffuse per Object-Relational Mapping (OR/M), soluzioni che permettono la conversione di dati tra database relazionali e linguaggi di programmazione object-oriented.
- **AOP package:** implementazione di aspect-oriented programming conforme allo standard AOP Alliance.

Spring DispatcherServlet

Progettato attorno a una servlet centrale che fa da dispatcher delle richieste, è completamente integrata con l'Inversion of Control container di Spring. La DispatcherServlet è vista come "Front Controller"



La DispatcherServlet è una normale servlet, in grado di:

1. Intercettare le HTTP request in ingresso che giungono al web container
2. Cercare un controller che sappia gestire la richiesta e invocarlo ricevendo un model e una view
3. Cercare un ViewResolver opportuno tramite cui scegliere una View e
4. Creare la HTTP Response.

Java Server Faces - JSF

La JSF è una tecnologia Java basata sul design pattern architetturale Model-View-Controller, il cui scopo è quello di semplificare lo sviluppo dell'interfaccia utente di una applicazione web; può quindi essere considerata un framework

per componenti lato server di interfaccia utente, che offre:

- ricche API per rappresentare i componenti e gestire il loro stato, gestire eventi, validare e convertire dati server-side, definire i percorsi di navigazione delle pagine ecc..
- ampia libreria di tag per aggiungere componenti a pagine Web e collegarli a componenti server-side
- costruita su Java Servlet e come alternativa a JSP

JSF Managed Bean

Sono basati su Java Beans e associati ai componenti dell'interfaccia utente in una pagina JSF. Definiscono un insieme di metodi che svolgono le funzioni come la convalida dei dati del componente, la gestione degli eventi, la gestione dei dati del modulo e la navigazione. Ci sarà un Managed Bean (o Backing Bean) per una pagina JSF. Seguono le regole standard dei JavaBeans:

- costruttore senza argomenti
- nessuna variabile di istanza public
- metodi “accessor” per evitare accesso diretto ai campi
- metodi get e set

Per utilizzarli si usa l’annotazione @*ManagedBean* che va a registrare automaticamente il componente come risorsa utilizzabile dal container JSF

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

Esempio di pagina (*beanhello.xhtml*)

```
<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>
```

I Managed Bean hanno 4 possibili scope:

- application → singola istanza per applicazione
- sessione → nuova istanza per ogni nuova sessione utente
- request → nuova istanza per ogni richiesta
- scopeless → acceduta anche da altri bean e soggetta a garbage collection come ogni oggetto java.

I JSF Managed Bean hanno inoltre metodi “action” che vengono invocati automaticamente in risposta ad un’azione dell’utente o un evento. Per quanto riguarda la configurazione, invece:

```
<managed-bean>
   <managed-bean-name>library</managed-bean-name>
```

```
    <managed-bean-class>com.oreilly.jent.jsf.library.model.Library</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
</managed-bean>
<managed-bean>
    <managed-bean-name>usersession</managed-bean-name>
    <managed-bean-class>com.oreilly.jent.jsf.library.session.UserSession</managed-beanclass>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
<managed-bean>
    <managed-bean-name>loginform</managed-bean-name>
    <managed-bean-class>com.oreilly.jent.jsf.library.backing.LoginForm</managed-beanclass>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

Facelets

É un un potente linguaggio di dichiarazione di pagine web utilizzato per costruire viste JSF, utilizzando modelli HTML e per costruire alberi di componenti. Nella tecnologia JSF è inclusa la servlet predefinita → "FacesServlet" che si occupa di gestire le richieste per pagine JSF (necessita di mapping in web.xml)

Il tipico ciclo di vita di un'applicazione Facelet è:

1. Deployment dell'app su server; prima che arrivi la prima richiesta, l'applicazione è in stato non inizializzato (anche non compilato)
2. Quando arriva una richiesta, viene creato un albero dei componenti contenuti nella pagina (messo in *FacesContext*), con validazione e conversione dati automatizzata
3. L'albero dei componenti viene popolato con valori dai backing bean, con possibile gestione eventi e handler
4. Viene costruita una view sulla base dell'albero dei componenti
5. Viene effettuato il rendering della vista al client, basato sull'albero di componenti
6. L'albero viene deallocato automaticamente
7. In caso di richieste successive (anche postback), l'albero viene ri-allocato

JSF e templating

Uno dei principali vantaggi del framework JSF è la possibilità di creare un modello di layout (template) che può essere riutilizzato in diverse pagine dell'applicazione. Ciò permette di mantenere un'interfaccia utente coerente, un look&feel uniforme e di ridurre la quantità di codice da scrivere.

Il templating in JSF viene generalmente realizzato utilizzando il tag `<ui:composition>`, che permette di definire il template. Essò può includere elementi fissi, come menu di navigazione o il footer, ed elementi variabili, come il contenuto della pagina che viene specificato attraverso il tag `<ui:define>`. Tra gli altri tag più utili ritroviamo:

- `<ui:insert>` → parte di un template in cui potrà essere inserito del contenuto
- `<ui:component>` → definisce un componente creato e aggiunto all'albero dei componenti
- `<ui:include>` → incapsula e riutilizza il contenuto per pagine multiple
- `<ui:param>` → per passare parametri a file chiuso

Un esempio (template e pagina che lo utilizza:

template.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
```

page.xhtml

```
<ui:composition template="/template.xhtml">
  <ui:define name="leftMenu">
```

```
      xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:head>
    <title>JSF Templating Example</title>
  </h:head>
  <h:body>
    <h:form>
      <h:panelGrid columns="2">
        <ui:insert name="leftMenu"/>
        <ui:insert name="content"/>
      </h:panelGrid>
    </h:form>
  </h:body>
</html>
```

```
    <h:panelGroup>
      <h:outputText value="Menu sinistro"/>
    </h:panelGroup>
  </ui:define>
  <ui:define name="content">
    <h:panelGroup>
      <h:outputText value="Contenuto della pagina 1"/>
    </h:panelGroup>
  </ui:define>
</ui:composition>
```

Navigation rule

Ogni regola di navigazione è come un flowchart con un ingresso e uscite multiple possibili. Vi è un singolo *<from-view-id>* per fare match con URI. Quando viene restituito il controllo, la stringa risultato viene valutata (successo, failure, verify, login..)

- *<from-outcome>* deve fare match con la stringa risultato
- *<to-view-id>* determina l'URI verso cui fare forwarding

```
<navigation-rule>
 <from-view-id>/login.xhtml</from-view-id>
 <navigation-case>
 <from-action>#{LoginForm.login}</from-action>
 <from-outcome>success</from-outcome>
 <to-view-id>/storefront.xhtml</to-view-id>
 </navigation-case>
 <navigation-case>
 <from-action>#{LoginForm.logon}</from-action>
 <from-outcome>failure</from-outcome>
 <to-view-id>/logon.xhtml</to-view-id>
 </navigation-case>
</navigation-rule>
```

Web Socket

Le Web Sockets sono un protocollo di messaggistica che permette una comunicazione asincrona e full-duplex su una singola connessione TCP. Non sono connessioni HTTP, nonostante lo utilizzino per avviare la connessione (handshake durante una *upgrade request*).

> *Full-duplex* → comunicazione bidirezionale in maniera contemporanea (es. telefono)

Il protocollo WebSocket permette una maggiore interazione tra browser e server, facilitando applicazioni che forniscono contenuti interattivi in tempo reale. Ciò è possibile fornendo un modo standard per il server di mandare contenuti al browser senza essere sollecitato dal client, permettendo ai messaggi di andare e venire tenendo la connessione aperta.

Le WebSocket nascono dalla necessità di superare i limiti del protocollo HTTP per quanto riguarda la comunicazione full-duplex, da cui derivano le seguenti soluzioni:

- **Polling:** il client fa polling a intervalli prefissati e il server risponde immediatamente.

  ▼ In pratica

Si tratta di una normale richiesta: “Ciao, sono qui, hai nuove informazioni da darmi?”. Per esempio, una volta ogni 10 secondi.

In risposta, come prima cosa, il server prende nota del fatto che il client è online, e poi invia un pacchetto di messaggi disponibili fino a quel momento.

Funziona, ma ci sono degli svantaggi:

1. I messaggi vengono trasferiti con un ritardo che può arrivare fino a 10 secondi (tra una richiesta e l’altra).
2. Anche se non ci sono messaggi, il server viene bombardato di richieste ogni 10 secondi, pure se l’utente è passato ad altre attività, o è inattivo. In termini di prestazioni, si tratta di un bel carico da gestire.

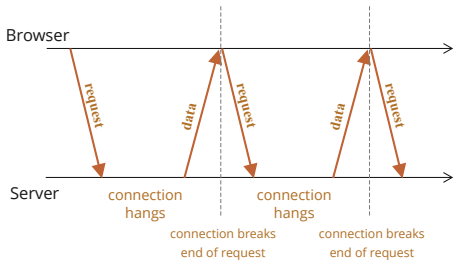
Quindi, se stiamo parlando di un servizio molto piccolo, l’approccio può essere percorribile, ma generalmente necessita di miglioramenti.

- **Long polling:** il client manda una richiesta iniziale e il server attende finchè non ha dati da inviare.

  Risulta davvero semplice da implementare, e recapita i messaggi senza alcun ritardo.

  Flusso:

  1. Viene inviata una richiesta al server.
  2. Il server non chiude la connessione fino a che ha un messaggio da inviare.
  3. Quando compare un messaggio, il server risponde alla richiesta con quest’ultimo.
  4. Il browser invia immediatamente una nuova richiesta.



  Con questo metodo, la situazione in cui il browser ha inviato una nuova richiesta e ha una connessione pendente con il server, è la situazione standard. La connessione viene ristabilita, solo quando viene consegnato un messaggio.

  L’unico contro è che ogni request/response si appoggia ad una nuova connessione. Se quest’ultima viene persa, poniamo il caso di un errore di rete, il browser invia una nuova richiesta.

  ▼ Bozza di una funzione lato client che effettua richieste long polling

```
async function subscribe() {
  let response = await fetch("/subscribe");

  if (response.status == 502) {
    // Lo status 502 indica un errore di timeout,
    // potrebbe succedere per connessioni pendenti da troppo tempo,
    // che il server remoto o un proxy chiudono
    // e quindi avviene una riconnessione
    await subscribe();
  } else if (response.status != 200) {
    // Un errore che andiamo a mostrare
    showMessage(response.statusText);
    // Riconnessione in un secondo
    await new Promise(resolve => setTimeout(resolve, 1000));
    await subscribe();
  } else {
    // Otteniamo e mostriamo il messaggio
    let message = await response.text();
    showMessage(message);
    // Chiamiamo subscribe() nuovamente per ottenere il prossimo messaggio
    await subscribe();
  }
}

subscribe();
// la funzione subscribe effettua un fetch e rimane in attesa della risposta, e dopo averla gestita richiama nuovamente se
stessa
```

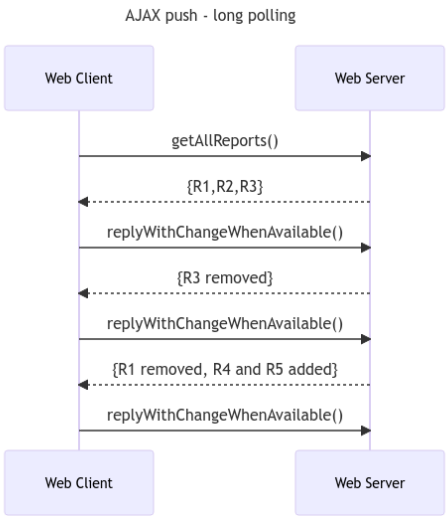
Esempio completo di una chat con long polling → https://it.javascript.info/long-polling#dimostrazione-una-chat

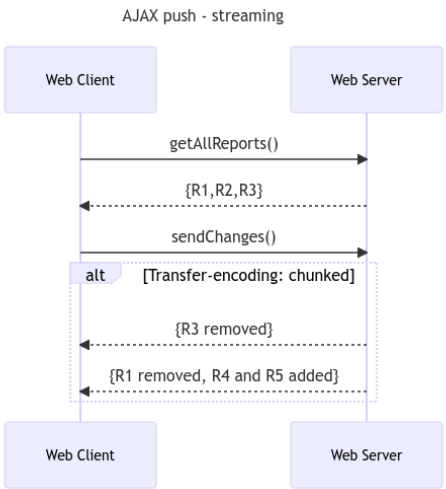
- **Streaming/forever response:** il client manda la richiesta iniziale e il server attende finchè ha dati da inviare.

  Il server risponse con streaming su una connessione mantenuta sempre aperta per aggiornamenti push (risposte parziali). Risulta half-duplex → solo server to client (essenzialmente il client manda una prima richiesta *HTTP GET* e poi è solo il server a inviare dati quando gli va)

- **Multiple connections:** consiste nel long polling su due connessioni HTTP separate: una per il long polling "tradizionale" e una per dati da client verso server. Ciò comporta però un coordinamento e una gestione delle connessioni abbastanza complesso, oltre che un overhead di due connessioni per ogni cliente.

Un implementazione migliore del long polling e delle connessioni multiple può essere fatta in **AJAX push**:



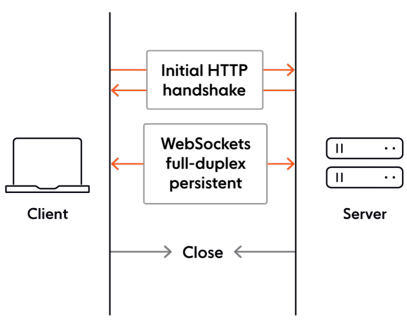


In sintesi quindi le **principali caratteristiche** delle WebSocket sono:

- Bi-direzionali → client e server possono scambiarsi messaggi quando desiderano
- Full-duplex → nessun requisito di interazione solo come coppia request/response e di ordinamento messaggi
- Unica connessione long running
- Visto come "upgrade" di HTTP
- Uso efficiente di banda e CPU → i messaggi possono essere del tutto dedicati a dati applicativi

Struttura del protocollo

Gli elementi base del protocollo sono l'handshake iniziale, quando il client comincia una connessione e il server risponde accettando l'upgrade, e una volta stabilita la connessione websocket entrambi gli endpoint vengono notificati che la socket è aperta e possono inviare messaggi e chiudere la socket in ogni istante.



```
// esempio di headers impostati dal browser per l'apertura di una websocket
1. GET /chat
2. Host: javascript.info
3. Origin: https://javascript.info
4. Connection: Upgrade
5. Upgrade: websocket
6. Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
7. Sec-WebSocket-Version: 13

// se il server acconsente allo switch in websocket risponde con un codice 101
1. 101 Switching Protocols
2. Upgrade: websocket
3. Connection: Upgrade
4. Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=
```

I dati (sia messaggi codificati in UTF-8 che binari) vengono trasmessi con minimo overhead in termini di header, e c'è possibilità di frammentazione in più frame.

L'approccio nel pratico viene integrato con **JavaScript lato client** e **JEE lato server** (uso di *annotations*).

JavaScript lato client

Per aprire una connessione websocket, usiamo il costruttore utilizzando il protocollo speciale `ws` :

```
let socket = new WebSocket("ws://javascript.info");
```

Esiste anche il protocollo criptato `wss://`, utilizzato per i websockets HTTPS → migliore da utilizzare perchè più affidabile.

Appena creato il socket, dovremmo rimanere in ascolto su di esso per gli eventi. Ce ne sono 4:

- `open` – connessione stabilita (established connection),
- `message` – dati ricevuti (data received),
- `error` – errore websocket (websocket error),
- `close` – connessione chiusa (connection closed).

...E nel caso volessimo inviare qualcosa al server, allora abbiamo il metodo `socket.send(data)` che si occupa di questo.

Un esempio:

```
let socket = new WebSocket("wss://javascript.info/article/websocket/demo/hello");

socket.onopen = function(e) {
  alert("[open] Connessione stabilita");
  alert("Invio al server");
  socket.send("Il mio nome è John");
};

socket.onmessage = function(event) {
  alert(`[message] Ricezione dati dal server: ${event.data}`);
};

socket.onclose = function(event) {
  if (event.wasClean) {
    alert(`[close] Connessione chiusa con successo, code=${event.code} reason=${event.reason}`);
  } else {
    // e.g. processo del server terminato o connessione già
    // in questo caso event.code solitamente è 1006
    alert('[close] Connection morta.');
  }
};

socket.onerror = function(error) {
```

```
  alert(`[error] ${error.message}`);
};
```

La comunicazione WebSocket è basata sui cosiddetti "frames" – frammenti di dati, che possono essere inviati da entrambe le parti e possono essere di tipi differenti.

- "text frames" – contengono dati in formato testuale che le parti inviano gli uni agli altri.
- "binary data frames" – contengono dati in formato binario che la parti inviano gli uni agli altri.
- "ping/pong frames" vengono usati per controllare la connessione, inviati dal server, il browser risponde vi risponde automaticamente.
- ci sono anche i "connection close frame" e altri frames di servizio.

Il metodo WebSocket `.send()` può inviare sia dati binari che testuali.

Una chiamata `socket.send(body)` permette che il `body` possa essere sia in formato stringa (default) che binario, incluso `Blob`, `ArrayBuffer`, etc. Non sono richieste configurazioni: basta inviarli in uno di questi formati.

```
var socket = new WebSocket("ws://server.org/wsendpoint");
socket.onmessage = onMessage;

function onMessage(event) {
  var data = JSON.parse(event.data);
  if (data.action === "addMessage") {
    // message processing
  }
  if (data.action === "removeMessage") {
    // message processing
  }
}

// chiusura della socket
socket.close(1000, "Work complete"); // 1000 è il code di default
```

Per ottenere lo stato della connessione, inoltre, c'è la proprietà `socket.readyState` con i valori:

- `0` – "CONNECTING": la connessione non è stato ancora stabilita,
- `1` – "OPEN": in comunicazione,
- `2` – "CLOSING": connessione in chiusura,
- `3` – "CLOSED": connessione chiusa.

JEE con annotations lato server

Lato server si utilizza la Java API for WebSocket (JSR-356) che presenta le seguenti caratteristiche:

- Gestione del ciclo di vita → `onOpen, onClose, onError`
- Comunicazione tramite messaggi → `onMessage, send`
- Possibilità di uso di sessione
- Encoder e decoder per formattazione di messaggi (messaggi ↔ oggetti Java)

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

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

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

  @OnError
  public void onError(Throwable error) { ... }

  @OnMessage
  public void handleMessage(String message, Session session) {
    // actual message processing
    // gli endpoint websocket possono inviare/ricevere messaggi sotto forma di testo o binario
    // per l'invio è necessario ottenere un oggetto session dalla connessione (come in questo metodo qui)
    // e usarlo per ottenere un RemoteEndpoint
    Session.getBasicRemote;
    Session.getAsyncRemote; // restituiscono ->

    void RemoteEndpoint.Basic.sendText(String text);
  }
}
```

Node.js

Node.js è un ambiente di esecuzione per JavaScript che viene utilizzato per sviluppare applicazioni web. È basato sull'engine JavaScript V8 di Google ed è stato progettato per essere leggero, veloce e scalabile.

Node.js viene utilizzato principalmente per lo sviluppo di server-side e di applicazioni di rete, ma può anche essere utilizzato per creare strumenti di linea di comando, giochi e altre applicazioni.

Uno dei suoi vantaggi principali è la sua capacità di eseguire operazioni asincrone in modo efficiente, il che lo rende particolarmente adatto per le applicazioni che devono gestire un gran numero di richieste simultanee, come ad esempio le applicazioni di chat o i giochi online.

Node.js è progettato per estrema concorrenza e scalabilità, senza thread o processi dedicati.

Al posto dei thread, utilizza un **event loop** con stack che va a ridurre fortemente l'overhead di context switching: vi è un singolo thread che si occupa di eseguire ogni operazione.

Quando un'operazione deve essere svolta, Node "passa il compito" al sistema operativo che creerà dei thread appositi per svolgere tale richiesta. Una volta completata la richiesta, verrà chiamata una funzione di callback ovvero viene avvisato il programma che l'operazione è stata conclusa. Tutto ciò rende Node completamente asincrono, perché ogni operazione viene delegata all'OS che andrà poi ad avvisarne il completamento tramite la funzione di callback, il tutto mentre l'esecuzione del programma Node continua senza bloccarsi.

Utilizzando il concetto del **callback**, Node.js può elaborare un gran numero di richieste senza aspettare che una qualsiasi funzione restituisca il risultato; ad esempio, quando una funzione inizia a leggere un file, restituisce il controllo all'ambiente di esecuzione immediatamente, in modo che l'istruzione successiva possa essere eseguita; una volta che l'I/O del file viene completato, la funzione di callback viene chiamata per avvisare il completamento dell'operazione.

```
var fs = require("fs");

fs.readFile('inputfile.txt', readDoneCallback);
function readDoneCallback(error, data) {
   if (error)
      return console.error(error);
   console.log(data.toString());
}
console.log("End of Program execution");
```

La libreria "fs" viene caricata per gestire le operazioni relative al file system. La funzione readFile() è asincrona e il controllo ritorna immediatamente all'istruzione successiva nel programma, mentre la funzione continua a elaborare in

background. Viene passata una funzione di callback che viene richiamata
quando il task in esecuzione in background è terminato. Si noti che la funzione di callback prende 2 argomenti: err e file. Per convenzione, il primo argomento è un errore e i successivi sono i dati della risposta. Se un errore viene lanciato, starà a noi doverlo gestire in qualche modo, se invece non accade
nessun errore, allora il primo argomento dovrebbe essere nullo.

Funzionalità principali

Thread vs. Event-driven

In molti linguaggi di programmazione e framework tradizionali, le operazioni di
I/O sono bloccanti: bloccano il progresso di un thread in attesa di lettura da, ad esempio, hard drive o rete. In caso di I/O bloccante, un server "tradizionale" usa multi-threading per limitare l'attesa, si ha quindi:

- Thread pool
- Un thread per connessione

Ma comunque ogni thread passa maggior parte del tempo in attesa di I/O. Andare verso altissimi numeri di thread introduce overhead di context switching e significativo uso di memoria.
Node.js usa un approccio single-thread non-blocking I/O: ogni funzione che fa operazioni I/O viene gestita in modo asincrono non-bloccante tramite callback. Utilizzando un thread singolo con un event loop, Node.js supporta decine di migliaia di connessioni
concorrenti, senza costo di context switching.

Quando utilizzare Node.js?

Node permette l'uso di una collezione di moduli che realizzano varie funzionalità core: per file system I/O, per networking, per funzioni crittografiche, per gestione stream di dati, ecc…
Possiede insiemi di moduli (framework) per velocizzare lo sviluppo di applicazioni Web, come Express.js.
Non è solo server-side (come già visto in React): sono disponibili diversi strumenti per sviluppo frontend.

Uniformità JavaScript client-server

Non c'è bisogno di cambiare linguaggio, si utilizza JavaScript sia lato client che lato server, ma è importante ricordare:

- client-side usa ampiamente DOM, tipicamente NO accesso a file o persistent storage
- server-side lavora prevalentemente con file e/o persistent storage, NO DOM

Moduli Node.js

Il core di Node consiste di circa una ventina di moduli, alcuni di più basso livello come per
la gestione di eventi e stream, altri di più alto livello come HTTP. 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 → package manager che semplifica sharing e riuso di codice JavaScript in forma di moduli, esegue tramite linea di comando ed è incluso in Node

Listener/Emitter pattern

Listener come funzione da chiamare quando un evento associato viene lanciato; Emitter come segnale che un evento è accaduto.

L'emissione di un evento causa l'invocazione di tutte le funzioni listener.

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

// in seguito ad emit, i listener sono invocati in modo sincrono-bloccante e ell'ordine in cui sono stati registrati
```

Stream

Node contiene moduli che producono/consumano flussi di dati (stream). Può essere modo utile per strutturare server. Si possono costruire stream anche dinamicamente e aggiungere moduli sul flusso. Esempi di stream sono: Readable stream, Writable stream, Duplex stream, Transform stream, ecc…

```
// Leggere file usando Stream:
var readableStreamEvent = fs.createReadStream("bigFile");

readableStreamEvent.on('data', function (chunkBuffer) {
console.log('got chunk of', chunkBuffer.length, 'bytes');
});
// Lanciato dopo che sono stati letti tutti i datachunk
readableStreamEvent.on('end', function() {
console.log('got all the data');
});
readableStreamEvent.on('error', function (err) {
console.error('got error', err);
});

// Scrivere file usando Stream:
var writableStreamEvent = fs.createWriteStream('outputFile’);
writableStreamEvent.on('finish', function () {
console.log('file has been written!');
});
writableStreamEvent.write('Hello world!\n');
writableStreamEvent.end();
```

TCP Networking

Esiste un modulo di rete Node, chiamato net, che fa da wrapper per le chiamate di rete del sistema operativo.

```
// include funzionalità di alto livello, come
var net = require('net');
net.createServer(processTCPconnection).listen(4000);

// che crea una socket, fa binding sulla porta 4000 e si mette in stato di listen per connessioni
// per ogni connessione TCP, invoca la funzione processTCPconnection
```

Express.js

Ispirato e basato sul precedente Sinatra, è il framework più utilizzato oggi per lo sviluppo di applicazioni Web su Node; ha come principali caratteristiche:

- focus su high performance
- diverse opzioni e motori di templating
- eseguibili per rapida generazione di applicazioni

```
var express = require('express’);
var app = express();
app.get('/',function(req,res) {res.send('Hello World!’); });
var server = app.listen(3000,function() {
    var host = server.address().address;
```

```
    var port = server.address().port;
    console.log('Listening at http://%s:%s',host,port);
});
```

Riassunto di Node

- Utilizzo dello stesso JavaScript engine sia lato browser che servitore
- Ovviamente senza bisogno di DOM lato server
- Gestione eventi su una coda degli eventi
- Ogni operazione esegue come una chiamata dall'event loop
- Uso di interfaccia ad eventi per ogni operazione di SO
- Wrapping di tutte le chiamate bloccanti di SO (I/O su file e socket/network)
- Uso di sistema di moduli (import/export)
- Moduli specializzati per il supporto a data management efficiente