

IL PROTOCOLLO HTTP

Il protocollo **HTTP** (**H**yper**T**ext **T**ransfer **P**rotocol) è un protocollo di livello applicativo utilizzato per trasferire le risorse web. È in grado di gestire sia le richieste inviate al server tramite URL sia le risposte inviate al client sotto forma di pagine.

Nei client e nei server mantengono, a livello di protocollo, uno stato (informazioni relative a messaggi precedentemente scambiati): è un protocollo *stateless*.

La terminologia

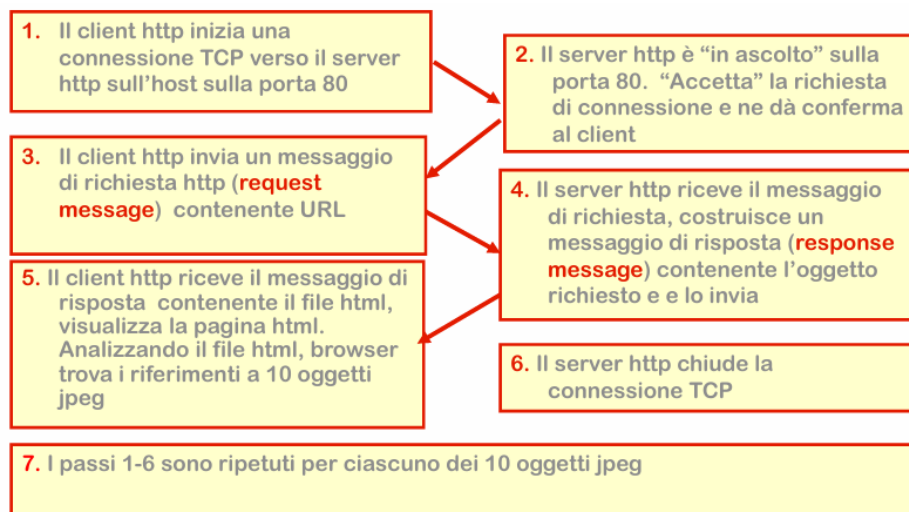
Ci sono diverse entità in gioco:

- **Client:** programma applicativo che stabilisce una connessione per inviare richieste
- **Server:** programma applicativo che accetta connessioni al fine di ricevere richieste e inviare le risposte con le risorse corrispondenti
- **Connessione:** collegamento (circuiti virtuale) stabilito a livello 4 (trasporto ISO/OSI) tra 2 applicazioni per comunicare
- **Messaggio:** è l'unità di base di comunicazione HTTP definita come una specifica sequenza di byte concettualmente atomica:
 - **Request:** messaggio HTTP di richiesta
 - **Response:** messaggio HTTP di risposta
- **Resource:** oggetto di tipo dato univocamente definito
- **URI:** Uniform Resource Identifier - identificatore unico di risorsa

Funzionamento di base

HTTP è un protocollo basato su TCP: sia client che server trasmettono le richieste e le risposte su uno stream TCP.

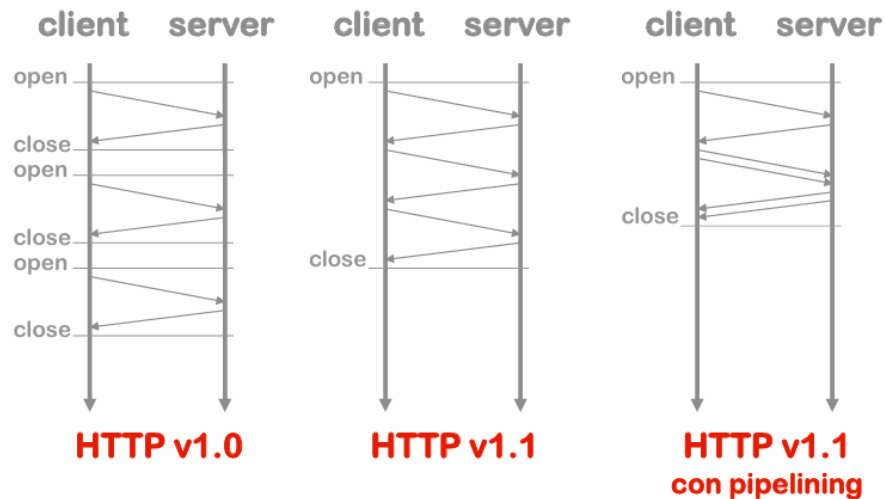
A titolo esemplificativo lo schema sottostante riporta il funzionamento di base richiedendo una pagina HTML e 10 immagini in formato jpeg:



Le diverse versioni

Esistono diverse versioni di HTTP ma quelle considerate in questo corso sono 2:

- HTTP v1.0
- HTTP v1.1 e v1.1 con pipelining



La differenza principale tra la 1.0 e 1.1 consiste nella possibilità del riutilizzo della stessa connessione HTTP per più richieste, difatti nella **versione 1.0** per **ogni richiesta** si apriva e chiudeva **una connessione**, mentre dall 1.1 in poi il server lascia aperta la connessione HTTP per eventuali nuove richieste e la chiude solo nel caso in cui nell'header del messaggio HTTP sia specificata la volontà del client di chiudere la connessione oppure dopo un time out.

Per migliorare le prestazioni di HTTP venne introdotto il pipelining che consiste nell'invio di molteplici richieste anche prima di aver ricevuto tutte le risposte. Il protocollo specifica che le risposte devono essere mandate **nello stesso ordine** delle richieste poichè non ci sono modi di ricondurre la risposta alla richiesta (non vi è, da protocollo, modo per associare richieste e risposte)

Esistono ulteriori versioni di HTTP:

- HTTP/2: ha l'obiettivo di migliorare le performance pur mantenendo la compatibilità con v1.1. È basato su SPDY, protocollo open networking promosso da Google. In particolare introduce:
 - request-response multiplexing
 - header compression
 - server push
- HTTP/3: vuole migliorare HTTP basandolo sul nuovo protocollo di trasporto QUIC (invece che TCP). Introduce:
 - stream multiplexing
 - controllo di flusso per stream
 - realizzazione di connessioni a bassissima latenza

I MESSAGGI HTTP

Un messaggio HTTP è composto da 2 strutture:

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

Esistono degli standard per ogni tipo di messaggio che non possono essere modificati.

I messaggi di response contengono i dati relativi ai messaggi di request. I dati sono codificati nel formato specificato nell'header, solitamente sono in formato MIME (Multipurpose Internet Mail Extensions)

Header HTTP Gli headers sono costituiti da coppie **nome:valore** che specificano le richieste del messaggio inviato/ricevuto. Ci sono headers di diverso tipo:

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

Il protocollo utilizza messaggi in formato ASCII. Un esempio abbastanza completo di **richiesta**:

```
GET /search?q=Introduction+to+XML+and+Web+Technologies HTTP/1.1
Host: www.google.com
User-Agent: Chrome/38.0 (X11; U; Linux i686; en-US; rv:1.7.2)
Gecko/20040803
Accept: text/xml,application/xml,application/xhtml+xml,
text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: da,en-us;q=0.8,en;q=0.5,sv;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.google.com/
```

I METODI DI RICHIESTA

GET

Serve per richiedere una risorsa ad un server, è il più utilizzato. È quello che viene attivato facendo click su un link ipertestuale di un documento HTML o specificando l'URL nel browser. Il passaggio dei parametri avviene tramite la parte < query > dell'URL. Una limitazione può essere che la lunghezza di un URL è limitata.

POST

Come GET serve per richiedere una risorsa, ma a differenza di quest'ultimo i parametri non sono passati all'interno dell'URL bensì nel body del messaggio di richiesta. Conseguentemente non ci sono limitazioni sulla lunghezza dei parametri di request. Viene spesso utilizzato per sottomettere dati di un form HTML ad una CGI. Tutto ciò non comporta la creazione di risorse sul server.

PUT e DELETE

Differentemente dal post creano (nel caso del PUT) e cancellano (nel caso del DELETE) risorse sul server all'URL specificato. Sono entrambi normalmente disabilitati sui server pubblici.

HEAD, OPTIONS e TRACE

L'HEAD è uguale al GET con la differenza che chiede al server di rispondere con solo headers relativi alla risposta ma senza body. Solitamente viene utilizzato per verificare la validità di un URL e se serve una autenticazione per quel URL.

OPTIONS server per richiedere informazioni sulle opzioni disponibili per la comunicazione

TRACE è usato per invocare il loop-back remoto a livello applicativo del messaggio di richiesta. Viene utilizzato in diagnostica e testing dei servizi Web.

IL FORMATO DELLA RISPOSTA

```
HTTP/1.1 200 OK
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>
```

Quanto sopra fa vedere un esempio di HTTP response, che in v1.0 corrisponde a chiudere la connessione, mentre in v1.1 il server mantiene aperta la connessione oppure la chiude se si aggiunge la clausola: **Connection: close**.

I CODICI DI STATO

Nella prima riga sono esposti protocollo e codice di stato. La prima cifra del codice di stato indica la classe di appartenenza ad una classe:

- **1xx** Informational: una risposta temporanea alla richiesta durante il suo svolgimento (sconsigliato da HTTP 1.0)
- **2xx** Successful: il server ha ricevuto, capito e accettato la richiesta
- **3xx** Redirection: il server ha ricevuto e capito la richiesta, ma sono necessarie altre azioni da parte del client per portare a termine la richiesta
- **4xx** Client Error: la richiesta del client non può essere soddisfatta per un errore da parte del client (richiesta non autorizzata o errore sintattico)
- **5xx** Server Error: La richiesta può essere corretta, ma il server non è in grado di soddisfare la richiesta per un problema interno

I COOKIE

Parallelamente alle sequenze request/response, il protocollo preve una struttura dati che si muove come un token da client a server e viceversa: **i cookie**. I cookie possono essere generati da entrambe le parti, e dopo la loro creazione vengono mandati ad ogni trasmissione. Scopo dei cookie è quello di supportare un mantenimento di stato in nel protocollo HTTP (stateless).

I cookie sono una collezione di stringhe:

- **Key:** identificatore univoco all'interno di un dominio:path
- **Value:** valore associato al cookie (max 255 caratteri)
- **Path:** posizione nell'albero del sito a cui è associato
- **Domain:** dominio dove è generato
- **Max-age:** numero di secondi di vita (opzionale)
- **Secure:** i cookie di questo tipo vengono trasferiti solo se si usa HTTPS (opzionale)
- **Version:** versione del protocollo di gestione dei cookie

AUTENTICAZIONE

Riconoscimento ip

È una soluzione poco utilizzata perchè presenta diversi svantaggi dovuti a NAT, DHCP e spoofing (tecniche per presentarsi con un IP falso)

HTTP basic

Consiste nella notifica, da parte del server, della necessità di autorizzazione per accedere ad una determinata risorsa, si aprirà un prompt sul client per la richiesta di username e password che saranno inviate (criptate) al server alla successiva request.

Form

Si utilizza un form HTML normalmente con method=POST e valgono le stesse considerazioni fatte per HTTP Basic

SICUREZZA

Per la trasmissione di informazioni si necessita, sempre più spesso, di sicurezza. Esistono 2 tipi di sicurezza per il canale di trasporto:

- **SSL:** Secure Sockets Layer
- **TLS:** Transport Layer Security
 - Sostituisce SSL
 - È alla base di HTTPS

Sostanzialmente viene posto un livello che si occupa della gestione di confidenzialità, autenticità ed integrità della comunicazione HTTP e TCP. Si accede tramite `https://...`

È tutto basato su crittografia a chiave pubblica e privata e tramite certificati per autenticare un server.

ARCHITETTURE DISTRIBUITE PER IL WEB

- **Proxy:** programma applicativo in grado di agire sia come Client che come Server al fine di effettuare richieste per conto di altri Clienti. Le Request vengono processate internamente oppure vengono ridirezionate al Server. Un proxy deve interpretare e, se necessario, riscrivere le Request prima di inoltrarle.
- **Gateway:** server che agisce da intermediario per altri Server. Al contrario dei proxy, il gateway riceve le request come se fosse il server originale e i Client non sono in grado di identificare che Response proviene da un gateway. Detto anche reverse proxy o server-side proxy.
- **Tunnel:** programma applicativo che agisce come “blind relay” tra due connessioni. Una volta attivo (in gergo “salito”) non partecipa alla comunicazione HTTP

CACHE

L'idea di base è memorizzare copie temporanee di documenti web al fine di ridurre l'utilizzo di banda e i tempi di attesa per il client. Una web cache memorizzano i documenti che la attraversano e, sotto specifiche condizioni, riutilizza questi documenti alle successive richieste del client.

USER AGENT CACHE

Lo user agent (browser) mantiene una cache delle pagine che ha visitato l'utente. Questo tipo di caching era importante in passato quando i client non avevano accesso a connessioni con banda larga. Tutt'ora è molto usato nei dispositivi mobili per permettere di lavorare anche con connettività intermittente e per ridurre i tempi di latenza.

PROXY CACHE

- **Forward Proxy Cache:** servono per ridurre le necessità di banda, il proxy intercetta il traffico e mette le pagine in cache in modo tale che alle successive richieste non si vada fino al server per scaricarle
- **Reverse (o server-side) Proxy Cache:** Gateway cache. Lavorano per conto del server per ridurre i carichi computazionali sulle macchine. I client non sono in grado di capire se le pagine arrivano dal server o dal gateway. Si utilizza ICP (Internet Caching Protocol) per il coordinamento fra le diverse cache, alla base per il Content Delivery Network.

HTTP definisce vari meccanismi che possono avere effetti collaterali per la gestione lazy dell'aggiornamento cache:

- **Freshness:** controllato dal server da Expires response header e lato cliente da direttiva CacheControl: max-age
- **Validation:** può essere usato per controllare se un elemento in cache è ancora corretto, ad es. nel caso in cui sia in cache da molto tempo (ad es. tramite 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

WEB DINAMICO

Il modello statico, quello visto fin'ora, con solo HTML e CSS è chiamato così perché pur essendo basato sul concetto di ipertesto distribuito l'insieme dei concetti è prefissato staticamente, le pagine sono preparate a priori e non esistono contenuti composti dinamicamente in base ad una qualsiasi interazione con l'utente. È sicuramente un modello semplice, potente, di facile implementazione ed efficiente, tuttavia presenta evidenti limitazioni.

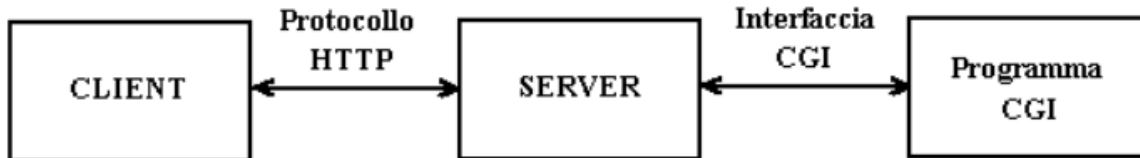
Prima di tutto proprio la limitata interazione con l'utente e quindi l'impossibilità di produrre dei risultati sulla base di input dell'utente. Ad esempio non è possibile implementare un qualsiasi tipo di ricerca all'interno del sito web visitato.

CGI

Una prima soluzione proposta per risolvere questo problema sono le **CGI** (Common Gateway Interface) presenti nel protocollo HTTP dalla versione 1.0. CGI è uno standard per le interfacce esterne con web server. I programmi scritti con questo tipo di standard prendono il nome di programmi CGI, possono essere scritti in un qualunque linguaggio di programmazione (C, PHP, Perl, Python ecc).

L'interazione avviene nel seguente modo:

1. il client invia una richiesta HTTP al server in cui chiede di eseguire un programma CGI con alcuni parametri in ingresso
2. Il server, attraverso l'interfaccia CGI, chiama il programma passandogli i parametri
3. Eseguite le operazioni, il programma rimanda al server i dati elaborati (pagina HTML) sempre tramite interfaccia CGI
4. Il server risponde al client con una HTTP response contenente i dati elaborati dal programma CGI



I programmi CGI ed il server comunicano tramite:

- Variabili d'ambiente
- Parametri sulla linea di comando: usati con il metodo GET, limitativo perchè la linea di comando ha una lunghezza massima, quindi la quantità di dati è limitata
- Standard input: usato con il metodo POST
- Standard output: per restituire l'HTML di risposta

Prima in chiamare il programma CGI il server imposta alcune env. var:

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

L'**output** è generato dopo l'elaborazione dei dati ed è comunicato al server tramite stdout, il server lo preleva e lo invia incapsulandolo in un messaggio HTTP.

Purtroppo questa soluzione presenta diversi problemi:

- Problemi di **prestazioni**: ogni volta che viene invocata una CGI si crea un processo (probabilmente pesante) che viene distrutto al termine dell'elaborazione
- **Robustezza**: le CGI, soprattutto se scritte in C, possono essere poco robuste in quanto è difficile capire che cosa potrebbe accadere in caso di errore bloccante
- Poco **riutilizzo** di codice già scritto: ogni programma CGI deve reimplementare una serie di parti comuni (accesso a DB, logica di interpretazione delle HTTP request, gestione di stato)
- **Sicurezza**: scarse garanzie di sicurezza

Per ovviare al problema del riutilizzo del codice si potrebbe creare un'unica CGI che implementa diverse funzioni, ma si finirebbe per avere un'applicazione monolitica di difficile scalabilità e aggiornamento, in più cosa potrebbe succedere se incappasse in errore? Nessuna funzionalità funzionerebbe più? Dipende dal linguaggio di programmazione, ma non si può correre questo rischio.

APPLICATION SERVER

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

- interfacciamento con il web server
- gestione del tempo di vita (attivazione on-demand delle funzioni)
- interfacciamento con il database
- gestione della sicurezza

La soluzione così pensata è modulare e le funzionalità ripetitive vengono portate a fattore comune. Un ambiente di questo tipo si chiama **application server**.

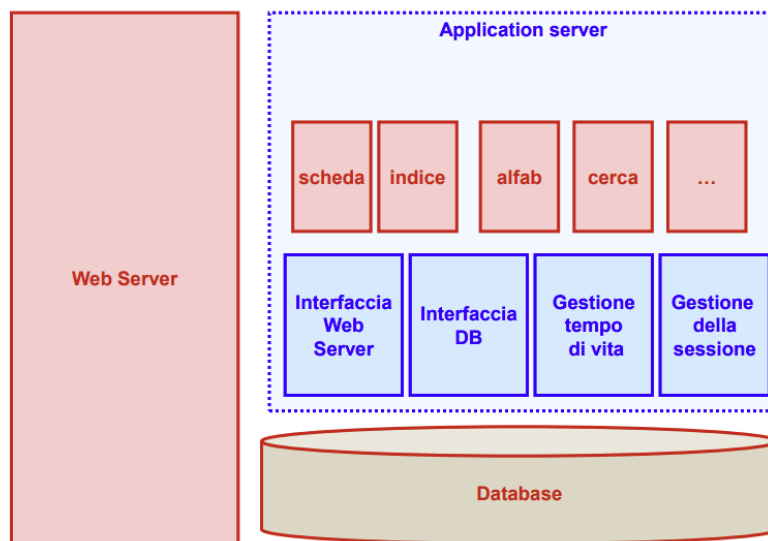


Figure 1: Esempio di architettura basata su application server

MODELLI A CONTENIMENTO

Molte funzionalità possono non essere controllate direttamente ma lasciate come responsabilità ad un'entità delegata supervisore (detta **contenitore**) che se ne occupa. Solitamente se ne occupa introducendo politiche di default, evitando che si verifichino errori e controllando gli eventi. I contenitori (detti anche container, engine o middleware) possono occuparsi di azioni automatiche sgravando l'utilizzatore che deve solo specificare la parte di alto livello, non ripetitiva e fortemente dipendente dalla logica applicativa. Si vedranno a titolo esemplificativo: CORBA (per gli aspetti C/S), Engine per framework a GUI e Container per servlet. Questi ultimi possono ospitare componenti più trasportabili, riutilizzabili e mobili. Il container fornisce molte delle funzioni per supportare l'interazione con l'utente:

- Supporto al ciclo di vita:
 - Attivazione/deattivazione del server
 - Mantenimento di stato
 - Persistenza trasparente e recupero delle informazioni

- Supporto al sistema di nomi (DNS):
 - Discovery del servizio
 - Federazione con altri container
- Supporto alla qualità del servizio:
 - Tolleranza ai guasti, selezione tra possibili deployment
 - Controllo della QoS richiesta e ottenuta
- Sicurezza

APPLICATION SERVER E TECNOLOGIE SERVER SIDE

Due tecnologie storicamente diffuse nell'ambito degli application server sono:

- .NET per componenti Microsoft ed evoluzioni
- Java J2EE per componenti EJB e Web Java-oriented

Altre tecnologie interessanti sono quelle basate su Perl, Python o JavaScript server side (node.js).

Altre soluzioni di struttura più semplice che però non sono application server ma spesso solo interpreti sono:

- PHP o Ruby
- Le vecchie tecnologie ISAPI e ASP di Microsoft

In queste però si parla di moduli di estensione del Web server.

LO STATO

L'interazione fra client e server può essere di 2 tipi:

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

INTERAZIONE STATELESS Un'interazione stateless è feasible senza generare grossi problemi solo se il protocollo applicativo è progettato con operazioni **idempotenti**. Le operazioni si dicono idempotenti se producono sempre lo stesso risultato indipendentemente dal numero di messaggi M ricevuti dal server stesso.

INTERAZIONE STATEFUL In generale tutte le volte che abbiamo bisogno di personalizzazione delle richieste Web possiamo beneficiare di interazioni stateful. Per esempio se è prevista un'autenticazione è necessario tener traccia fra una chiamata e l'altra del fatto che l'utente si è autenticato in modo tale da non dover rifare l'autenticazione ad ogni request, e da qui nasce il concetto di **stato**.

Nelle applicazioni Web è possibili classificare precisamente lo stato:

- **Stato di esecuzione** (insieme di dati parziali per una elaborazione): rappresenta un'avanzamento in stato di esecuzione, può essere mantenuto in memoria lato server come stato di uno o più oggetti. Per sua definizione è volatile.
- **Stato di sessione** (insieme dei dati che caratterizzano una interazione con uno specifico utente): la sessione viene gestita di solito in modo unificato attraverso l'uso di istanze di oggetti specifici (supporto ad oggetti di sessione).
- **Stato informativo persistente** (ad esempio gli ordini inseriti in un eCommerce): viene normalmente mantenuto in una struttura persistente tipo un database.

LA SESSIONE

La sessione rappresenta lo stato associato ad una sequenza di pagine visualizzate dall'utente. Contiene tutte le informazioni necessarie durante l'esecuzione: le informazioni di sistema (IP di provenienza, lista delle pagine visualizzate, ...) e quelle di natura applicativa (nome e cognome, username, quanti e quali prodotti messi nel carrello ...). Lo **scope** di sessione è dato da:

- Tempo di vita dell'interazione con l'utente (*lifespan*)
- Accessibilità: usualmente concessa alla richiesta corrente e alle future provenienti dallo stesso processo browser

La **conversazione** rappresenta una sequenza delle pagine di senso compiuto (ad esempio tutte le pagine necessarie per comprare un prodotto), è univocamente definita dall'insieme delle pagine che la compongono e dall'insieme delle interfacce di I/O per la comunicazione (flusso della conversazione).

Lo stato di sessione **deve** essere condiviso da client e server, inoltre è associato a uno o più conversazioni effettuate da ogni singolo utente, ne segue che ciascun user possiede il suo stato. Per gestire tutto ciò ci sono due tecniche: utilizzo dei **cookie** (storage lato client) e la gestione di uno stato sul server per ogni utente collegato (**session** e server side). Queste tecniche non sono necessariamente alternativa ma anche integrabili. La gestione della sessione è uno dei supporti orizzontali messi a disposizione di un application server.

ARCHITETTURA FREQUENTE DEI SISTEMI WEB

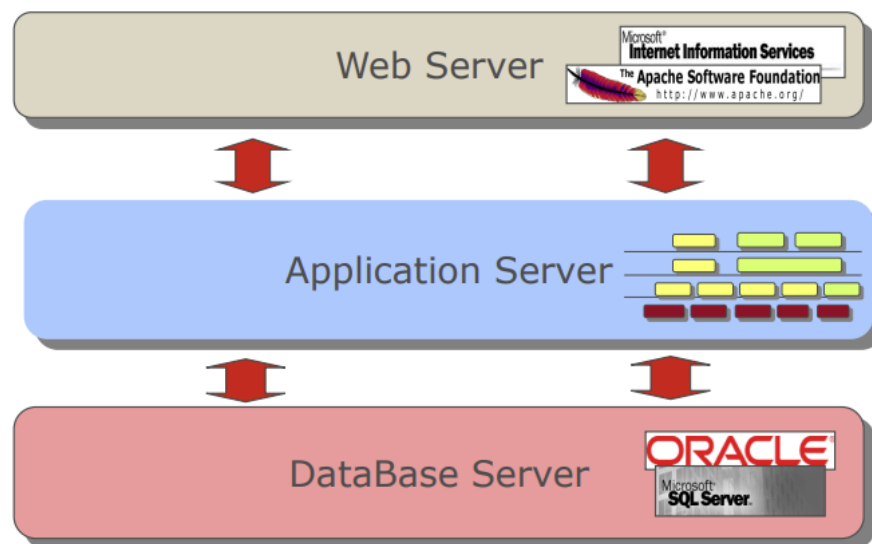


Figure 2: Esempio di architettura a 3 tier

La struttura rappresentata in figura rispecchia i 3 principali servizi che realizzano un sistema Web, possono risiedere tutti sullo stesso HW oppure essere su macchine separate (**distribuzione verticale**). La distribuzione non necessita di nessun accorgimento specifico e solitamente viene realizzata per migliorare le performance soprattutto quando si separa il livello applicativo dal livello del DB. Così com'è rappresentata non prevede replicazione quindi non è utile per risolvere i problemi di fault tolerance.

Per provvedere a questo problema si distribuisce orizzontalmente ogni livello: si replica il servizio su diverse macchine, purtroppo questo comporta importanti accorgimenti strettamente dipendenti dalla tecnologia in uso. La **distribuzione orizzontale** non è fatta solo per gestire la fault tolerance ma anche per distribuire e bilanciare il carico computazionale al fine di migliorare le performance.

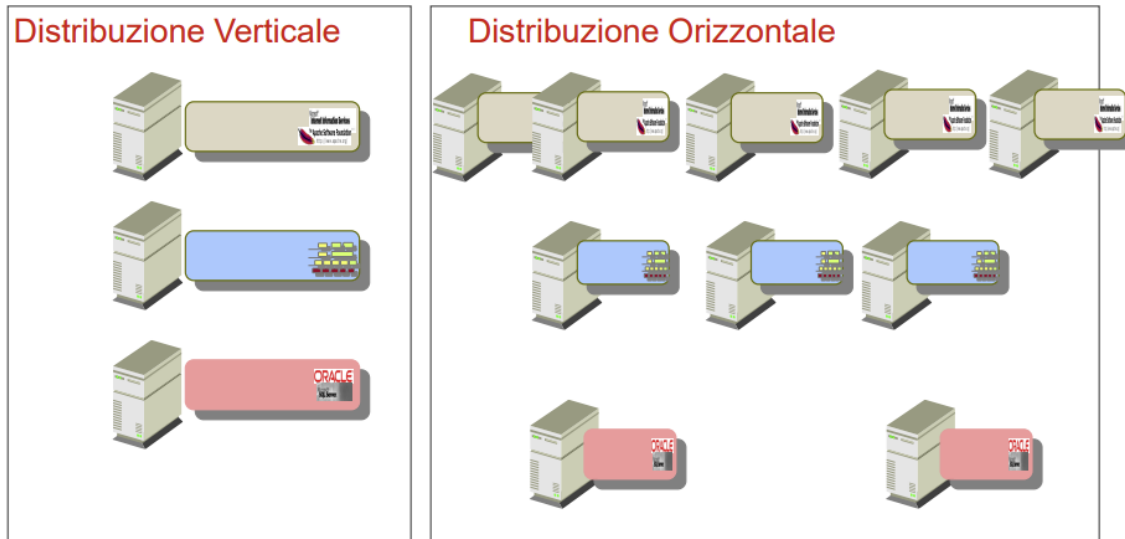


Figure 3: Distribuzione verticale ed orizzontale

Per la natura di HTTP il **Web server** è stateless e quindi molto facile da replicare. Il fatto che IP è embedded in URL può essere gestito sia tramite soluzione HW che SW. Si possono inoltre applicare politiche di load balancing con diverse euristiche usando i dispositivi appositi.

Per la replicazione del livello di **application server** si parla principalmente di replicazione dello stato di sessione. Può però accadere che un application server utilizzi oggetti o componenti con stato per motivi di performance o altre necessità specifiche, alcuni framework permettono la replicazione con tecniche di clustering ma altri framework non sono in grado di replicare orizzontalmente. Tuttavia se si mantiene lo stato di sessione concentrato all'interno della sessione e la sessione viene gestita internamente attraverso i cookie è possibile realizzare un framework applicativo interamente stateless lato server, ottenendo così una realizzazione più semplice e primitiva di configurazione completamente replicabile orizzontalmente.

La replicazione del **database** è molto delicata perchè deve mantenere il principio di atomicità delle transazioni. In più il database server è normalmente stateful, il che può creare non poche complicazioni nella replicazione. I database commerciali come Oracle e Microsoft SQL Server prevedono delle configurazioni di clustering in grado di gestire in modo trasparente un numero variabile di CPU e macchine distinte, anche se generalmente in numero basso (qualche unità).

SERVLET

Prima di dare la definizione e le modalità d'uso delle servlet è necessario introdurre un po' di definizioni:

- **Web application:** è un gruppo di risorse server-side che nel loro insieme creano un'applicazione interattiva fruibile via web
- **Risorse server side:** includono classi server-side, JSP, risorse statiche (doc. html, css, immagini, ...), applet e JS e/o altri componenti che diventeranno attivi client-side, e informazioni di config e deployment
- **Web container:** forniscono un ambiente di esecuzione per web application, in generale garantiscono servizi di base alle applicazioni sviluppate secondo paradigma a componenti

Una **servlet** è una classe (java) che fornisce un servizio comunicando con il client mediante protocolli di tipo request/response, tra questi il più diffuso è HTTP. Le servlet estendono le funzionalità di un web server generando contenuti dinamici e superando i classici limiti delle applicazioni CGI. Esse eseguono direttamente in un web container.

A livello pratico tutte le servlet sono sottoclassi della classe `HttpServlet` che implementa vari metodi che si andranno a ridefinire. Tutte le classi che ci interessano si trovano all'interno del package `javax.servlet.http.*`.

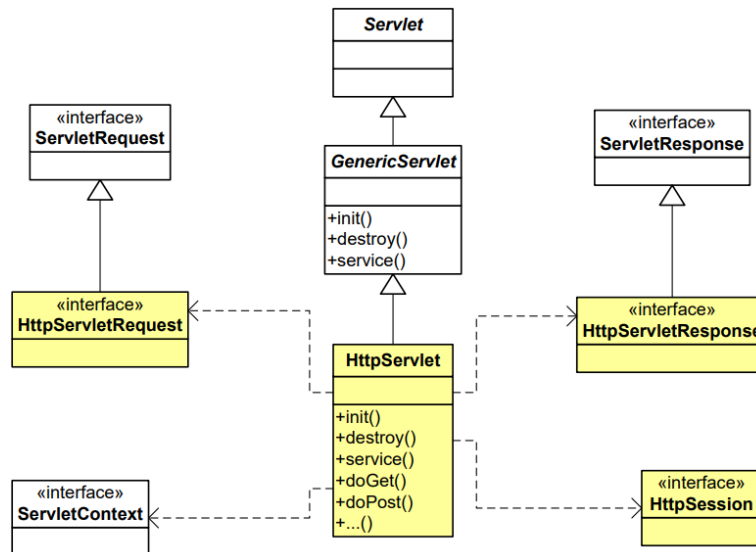


Figure 4: Gerarchia delle classi e interfacce per Servlet

Gli oggetti **Request** rappresentano la chiamata fatta dal server al client. Sono caratterizzati da chi ha fatto la request e quali parametri e headers sono stati passati nella request.

Gli oggetti **Response** rappresentano le informazioni restituite al client (in risposta alla request). In particolare contengono: status line (status code e status phrase), gli headers e il response body. Ha come interfaccia tipo l'interfaccia `HttpServletResponse` che espone metodi per:

- specificare lo status code
- indicare il content type (es text/html)
- ottenere un output stream
- indicare se l'output è bufferizzato
- gestire i cookie

Il ciclo di vita di una servlet

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

Se non esiste un'istanza della classe servlet nel container:

Carica la classe della servlet

Crea un'istanza della classe

Inizializza la servlet (invocando `init()`)

A regime:

Invoca la servlet (`doGet()` o `doPost()`) passando gli oggetti di tipo `HttpServletRequest` e `HttpServletResponse`

Servlet e multithreading In un modello normale c'è una sola istanza di servlet e un thread assegnato ad ogni request per la servlet anche se altre richieste sono già in esecuzione. Di conseguenza, avendo più thread che condividono una risorsa sola (servlet) si crea concorrenza:

- il metodo `init()` è invocato una sola volta quando il web container crea la servlet
- `service()` e `destroy()` possono essere chiamati solo dopo il completamento di `init()`
- il metodo `service()`, e quindi `doGet()` e `doPost()`, può essere invocato da numerosi client in modo concorrente ed è quindi necessario gestire le sezioni critiche (a carico del programmatore) tramite l'uso di blocchi `synchronized`, semafori e mutex

Si può anche pensare ad un modello single-threaded ma è deprecato e altamente inefficiente. Per implementare ciò una servlet dovrebbe implementare l'interfaccia `SingleThreadModel`

I metodi del controllo del ciclo di vita

`init()`

Viene chiamato una sola volta al caricamento della servlet. In questo metodo si può inizializzare l'istanza (ad es. si crea la connessione ad un DB).
È definito nella classe `GenericServlet`

`service()`

Viene chiamato ad ogni request. Chiama la `doGet()` o la `doPost()` a seconda della tipologia di request ricevuta.

È un metodo astratto definito nella classe `GenericServlet`

`HTTPServlet` fornisce un'implementazione di questo metodo che delega la richiesta a seconda del metodo:

- `doGet()`
- `doPost()`
- `doPut()`
- `doDelete()`

`destroy()`

Viene chiamato una sola volta quando la servlet deve essere disattivata, solitamente serve per rilasciare le risorse acquisite.

È definito nella classe `GenericServlet`

Gestione dello status code

È possibile definire lo status code della response tramite:

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

Mentre per gli errori:

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

Gestione degli headers Ci sono le funzioni per gestire gli headers della response:

```
public void setHeader(String headerName, String headerValue); //imposta un header arbitrario
public void setDateHeader(String name, long millisec); // imposta la data
public void setIntHeader(String name, int headerValue); // imposta un header con valore intero
```

Quelle elencate sopra sono generali ma esistono anche:

- `addHeader`, `addDateHeader`, `addIntHeader` aggiungono una nuova occorrenza dell'header dato
- `setContentType` configura il content-type. **Si usa sempre**
- `setContentLength` utile per la gestione di connessioni persistenti
- `addCookie` consente di gestire i cookie nella risposta
- `sendRedirect` imposta location header e cambia status code in modo da fare la redirectione

Gestione del contenuto Per gestire il contenuto della response si possono utilizzare due metodi di response:

```
public PrintWriter getWriter();
public ServletOutputStream getOutputStream();
```

Il primo è solitamente utile per restituire un testo nella risposta perchè mette a disposizione uno stream di caratteri.

Il secondo mette a disposizione uno stream di byte quindi è più utile per l'invio di un'immagine ad esempio.

L'oggetto request È un'istanza di una classe che implementa l'interfaccia `HttpServletRequest`, contiene i dati della request HTTP. Viene creato dal servlet container e passato come parametro ai metodi `doGet()` e `doPost()`. Fornisce metodi per accedere alle varie info (HTTP Request URL, HTTP Request headers, autenticazione, user info, cookie e session)

Ricorda!!

L'URL HTTP ha la seguente sintassi: `http://[host]:[port]/[request path]?[query string]`: il request path è composto dal contesto e dal nome della web application, la query string è composta da un insieme di parametri se la richiesta è in get (il metodo `getParameter(...)` di request ci permette di avvedere ai vari parametri)

Esistono tutti i vari metodi per accedere all'URL, agli headers e alle user info, autenticazione, cookie e sicurezza

Il Deployment

Un'applicazione web deve essere installata sul server prima di poter essere utilizzata: questa operazione prende il nome di **deployment**. Esso comprende:

- La definizione del runtime environment di una web app
- La mappatura delle URL sulle servlet
- La definizione delle impostazioni di default dell'applicazione come la welcome page o la error page
- La configurazione delle caratteristiche di sicurezza dell'applicazione

La modalità con cui avviene il deployment è rappresentata dagli Web Archives (.war), non sono altro che dei file .jar con una struttura particolare.

Per crearli si usa il comando `jar` ad esempio `jar -cvf newArchive.war myWebApp/*`



Figure 5: Struttura interna del .war

La struttura di dir delle Web app è basata su Servlet 2.4 specification.

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

Esempio di mappatura della servlet

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

E l'URL mappato su myServlet è `http://MyHost:8080/MyWebApplication/myURL`

Servlet configuration

Una servlet accede ai propri parametri di configurazione mediante l'interfaccia `ServletConfig`. Ci sono 2 modi per accedere a oggetti di questo tipo:

- tramite il parametro passato al metodo `init()`
- tramite il metodo `getServletConfig()` della servlet, invocabile in qualsiasi momento

L'interfaccia `ServletConfig` espone un metodo per ottenere il valore di un parametro in base al nome:

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

Esempio di parametro di configurazione:

```
<init-param>
  <param-name>parName</param-name>
  <param-value>parValue</param-value>
</init-param>
```

Servlet context

Ogni Web application esegue in un contesto: corrispondenza 1:1 tra una Web-app e suo contesto. L'interfaccia `ServletContext` è la vista della Web application (del suo contesto) da parte della servlet. Si può ottenere un'istanza di tipo `ServletContext` all'interno della servlet utilizzando il metodo `getServletContext()`. L'istanza consente di accedere ai parametri di inizializzazione, agli attributi di contesto e alle risorse statiche della web app (ad es le immagini) mediante il metodo `getResourceAsStream(String path)`. Molto importante è ricordare che il servlet context è condiviso fra tutti gli utenti, tutte le richieste e tutti i componenti server-side della stessa web app.

Parametri di inizializzazione del contesto sono definiti all'interno di elementi di tipo `context-param` nel `web.xml` e sono accessibili in lettura a tutte le servlet della web app.

Nel `web.xml`:

```
<web-app>
<context-param>
<param-name>feedback</param-name>
<param-value>feedback@deis.unibo.it</param-value>
</context-param> ...
</web-app>
```

Nelle servlet:

```
ServletContext ctx = getServletContext();
String feedback =
ctx.getInitParameter("feedback");
```

Attributi di contesto sono accessibili a tutte le servlet e sono sostanzialmente delle variabili globali, possono essere creati, scritti e letti dalle servlet e possono contenere oggetti complessi (serializzazione/deserializzazione).

<pre>//SCRITTURA ServletContext ctx = getServletContext(); User giorgio = new User("Giorgio Bianchi"); User paolo = new User("Paolo Rossi"); User claudio = new User("Claudio Neri"); ctx.setAttribute("utente1", giorgio); ctx.setAttribute("utente2", paolo); ctx.setAttribute("utente3", claudio);</pre>	<pre>//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); }</pre>
---	---

Gestione dello stato (di sessione)

Essendo HTTP un protocollo stateless non fornisce meccanismi nativi per la gestione dello stato, il problema sta nel fatto che la maggior parte delle web app attuali hanno bisogno di stato. Per questo si utilizzano i **cookie** (basso livello) oppure la **sessione** (alto livello). La sessione stessa è un'utile astrazione del concetto che rappresenta, per funzionare utilizza 2 meccanismi: i cookie e l'URL rewriting. Dei cookie si è già parlato in precedenza ma sostanzialmente sono un'unità di informazione che il web server deposita sul web browser client-side, tuttavia possono essere rifiutati dal browser (per esempio se sono disabilitati o se l'utente non ne consente l'uso) e spesso sono considerati un fattore di rischio.

La classe Cookie Modella il cookie HTTP, si recuperano i cookie della **request** utilizzando il metodo `getCookies()` e si aggiungono alla **response** usando `addCookie()`. Un cookie contiene un certo numero di informazioni, per esempio:

- coppia nome-valore
- il dominio internet che ne fa uso
- path dell'applicazione
- una expiration date in secondi (-1 se il cookie non verrà memorizzato)
- un valore boolean per definire il livello di sicurezza (usando `setSecure(true)` il client è forzato a inviare il cookie solo tramite protocollo HTTPS)

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

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

Uso della sessione in web container

La sessione web è un'entità gestita dal web container, è condivisa da tutte le richieste provenienti da uno stesso client (consente di mantenere uno stato). Può contenere dati di varia natura ed è identificata da un **session id**. Viene utilizzata dai componenti della web app proprio per mantenere uno stato, che HTTP standard non prevede, durante le molteplici interazioni con l'utente.

Accesso e gestione della sessione

L'accesso avviene mediante l'interfaccia `HttpSession`. Per ottenere l'accesso ad un oggetto di tipo `HttpSession` si utilizza il metodo dell'interfaccia `HttpServletRequest`:

```
public HttpSession getSession(boolean createNew);
//AD ESEMPIO
HttpSession session = request.getSession(true);
```

Se `createNew` è a valore `true` ritorna la sessione esistente, se non esiste ne crea una nuova.

Se invece `createNew` è `false` ritorna, se esiste, la sessione esistente, altrimenti `null`.

Negli attributi di sessione si possono memorizzare dati specifici dell'utente (coppia nome-valore), simili agli attributi di contesto anche se con **scope** totalmente diverso.

```
//Recupero di un oggetto dagli attributi
Cart sc = (Cart)session.getAttribute("shoppingCart");
sc.addItem(item);
session.setAttribute("shoppingCart", new Cart());
session.removeAttribute("shoppingCart");
Enumeration e = session.getAttributeNames();
while(e.hasMoreElements())
    out.println("Key: " + (String)e.nextElement());
```

Ci sono altre operazioni che si possono compiere su un oggetto sessione:

- `String getId()` Restituisce l'ID di sessione
- `boolean isNew()` dice se la sessione è nuova
- `void invalidate()` permetti di invalidare (distruggere) la sessione
- `long getCreationTime()` dice da quanto tempo è attiva la sessione in millisecondi
- `long getLastAccessTime()` dà informazioni su quando è stata utilizzata l'ultima volta

Un esempio di utilizzo di queste funzioni (e metodi) è il seguente, dove si prende l'ID di sessione, si controlla se è nuova, si invalida e si mostrano da quanto tempo è attiva e l'ultima volta che si è fatto un accesso alla sessione:

```
String sessionID = session.getId();
if(session.isNew())
    out.println("La sessione e' nuova");
session.invalidate();
out.println("Millisec:" + session.getCreationTime());
out.println(session.getLastAccessedTime());
```

Il **session ID** è usato per identificare le richieste provenienti dallo stesso utente e mapparle sulla corrispondente sessione. Ci sono due tecniche per trasmettere questo dato:

- Tramite cookie che però non in tutti i browser sono attivi
- Si include nell'URL: si parla di **URL rewriting**. È buona prassi codificare gli URL generate dalle servlet usando il metodo `encodeURL()` della `HttpServletResponse` (si usa sia per i tag a `href="..."` che per i form).

Scope differenziati

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

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

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

- `void setAttribute(String name, Object o)`
- `Object getAttribute(String name)` per assegnarlo ad una variabile con chiarezza serve il cast
- `void removeAttribute(String name)`
- `Enumeration getAttributeNames()`

Inclusione di risorse web

Includere risorse web può essere utile quando si vogliono aggiungere contenuti creati da un'altra risorsa. Si possono includere 2 tipi di risorse:

- risorse statiche, ad esempio un'altra pagina html
- risorse dinamiche: la servlet inoltra una request ad un componente web che la elabora e restituisce il risultato che viene poi incluso nella pagina prodotta dalla servlet

La risorsa inclusa può lavorare con il responsive body.

Per **includere** una risorsa si ricorre ad un oggetto di tipo `RequestDispatcher` che può essere richiesto al contesto indicando la risorsa da includere. Per effettuare l'effettiva inclusione si utilizza il metodo `include(...)` e come parametri devono essere passati la request e la response che vengono così condivisi con la risorsa inclusa. Se necessario l'URL originale può essere salvato come un attributo di request. Un esempio:

```
// come parametro si passa l'URL della risorsa da includere
RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher("/inServlet");
dispatcher.include(request, response);
```

Oltre all'inclusione si può fare anche l'**inoltro** ad un'altra risorsa. Solitamente si usa in situazioni in cui la servlet si occupa di parte dell'elaborazione della richiesta e delega a qualcun altro la gestione della risposta: è importante ricordare che in questo caso la response è di competenza esclusiva della risorsa che riceve l'inoltro, nel caso nella servlet che inoltra si faccia un accesso a `ServletOutputStream` o a `PrintWriter` si ottiene una `IllegalStateException`. Anche in questo caso si ottiene un oggetto di tipo `RequestDispatcher` da request passando come parametro il nome della risorsa, e per effettuare l'inoltro si utilizza il metodo `forward(...)` passando anche qui request e response. Come per l'inclusione si può, se necessario, salvare l'URL originale come attributo di request. Un esempio:

```
// come parametro si passa l'URL della risorsa a cui si inoltra
RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher("/inServlet");
dispatcher.forward(request, response);
```

Attenzione: completamente diversa è la ridirezione del browser. Difatti è anche possibile inviare al browser una risposta che lo forza ad accedere ad un'altra pagina (ridirezione). Come visto in precedenza, si utilizza uno dei codici di stato HTTP dedicati a questo tipo di informazioni, in particolare si usa il **301 Moved permanently**: URL non valida, il server indica la nuova posizione. Per ottenere questo risultato si può agire direttamente sull'oggetto response in due possibili modi:

- invocando il metodo: `public void sendRedirect(String url)`
- Lavorando a livello più basso tramite header:
`response.setStatus(response.SC_MOVED_PERMANENTLY);`
`response.setHeader("Location", "http://...");`

JSP

Le JSP, Java Server Pages, sono uno dei due componenti di base della tecnologia J2EE, relativamente alla parte web hanno template per la generazione di contenuto dinamico ed estendono HTML con codice Java custom.

Quando viene effettuata una richiesta a una JSP (in sintesi):

1. la parte in HTML vanilla 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 finale (parte statica + parte generata dinamicamente) viene restituita al client

Per certi versi le JSP sono assimilabili a linguaggi di scripting come PHP o Perl, ma in realtà vengono trasformate in servlet dal container.

Le richieste JSP sono gestite da una particolare servlet (ad es. su Tomcat si chiama JspServlet) che effettua le seguenti operazioni:

- come detto in precedenza, la JSP viene tradotta in servlet
- compilazione della servlet risultante in un .class
- esecuzione

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

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

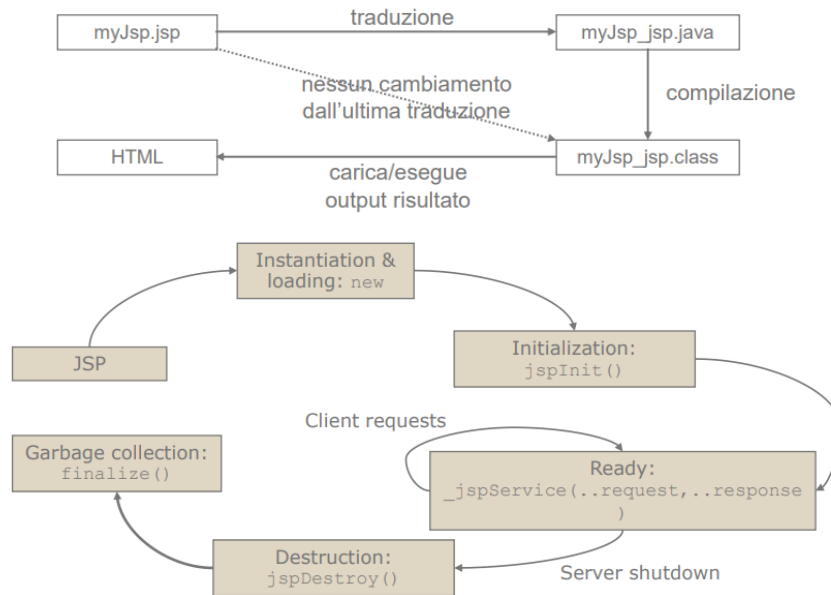


Figure 6: Sopra la creazione di una JSP e sotto il ciclo di vita

Ma quindi perchè usare JSP invece delle servlet?

Nelle servlet la generazione del documento HTML è completamente implementata in Java, infatti il processo di generazione è time-consuming, ripetitivo e soggetto a errori, in più l'aggiornamento delle pagine è scomodo. Le JSP invece nascono per facilitare la progettazione grafica e l'aggiornamento delle pagine: si può separare facilmente il lavoro fra grafici e programmatori, gli web designer possono produrre pagine senza dover conoscere i dettagli della logica server-side e, infine, la generazione del codice dinamico è implementata in Java.

E allora quale devo usare?

La risposta giusta a questa domanda è: dipende. Le servlet forniscono il completo controllo della web app agli sviluppatori, inoltre se si vogliono fornire contenuti differenziati a seconda di diversi parametri (ad es. identità dell'utente, condizioni date dalla business logic ecc.). JSP invece si usa quando si devono fare pagine dinamiche HTML o XML e di uso frequente in quanto sono molto semplici da implementare tramite questo linguaggio di scripting. Tuttavia JSP, come tutti i linguaggi di scripting che generano codice, rende più problematico il testing e il controllo della correttezza.

COME FUNZIONANO LE JSP

Prima di cominciare con la trattazione sulla sintassi e gli esempi è utile fare un paio di considerazioni sul flusso. Ricordando come funziona HTTP e la struttura delle pagine HTML: il client si aspetta di ricevere prima tutto il response header e poi solo successivamente il body, quindi JSP prima deve effettuare le modifiche all'header prima di cominciare con il body.

Una volta che il web server comincia a restituire la response non può più interrompere il processo, altrimenti il browser visualizzerebbe solo la frazione parziale che ha ricevuto, ne consegue che una volta che JSP ha cominciato a produrre output non si può più effettuare la forward ad un'altra JSP (proprio come le servlet).

Ogni volta che arriva una request il server compone dinamicamente il contenuto della pagina. Ogni volta che incontra un tag JSP valuta l'espressione Java contenuta all'interno e inserisce il risultato nell'output. Come è facile immaginare questo meccanismo permette di creare dinamicamente le pagine.

I tag

I tag sono le parti variabili della pagina, valutate ad hoc per ogni request. Esistono 2 tipi di tag:

- Scripting-oriented tags,
sono definiti da delimitatori entro cui è presente lo script (self-contained):
 - `<%! %>` Dichiarazione
 - `<%= %>` Espressione
 - `<% %>` Scriptlet
 - `<%@ %>` Direttiva
- XML-oriented tags,
seguono la sintassi XML, esistono tags equivalenti a quelli elencati sopra (ma si usano prevalentemente gli scripting-oriented tags):
 - `<jsp:declaration>declaration</jsp:declaration>`
 - `<jsp:expression>expression</jsp: expression>`
 - `<jsp:scriptlet>java_code</jsp:scriptlet>`
 - `<jsp:directive.dir_type dir_attribute />`

Ecco un esempio di Hello World con possibile nome del visitatore della pagina:

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

Se chiamassi questa pagina con URL del tipo `http://hostname:port/webAppName/helloworld.jsp` il risultato sarebbe:

```
<html>
  <body>
    Hello, World!
  </body>
</html>
```

Se invece l'URL fosse `http://hostname:port/webAppName/helloworld.jsp?name=Pier` il risultato sarebbe:

```
<html>
  <body>
    Hello, Pier!
  </body>
</html>
```

Le Dichiarazioni: come definito in precedenza si usa il tag `<%! >` per dichiarare variabili e metodi. Le variabili e i metodi dichiarati possono essere referenziati in ogni punto del codice JSP. I metodi diventano metodi della servlet quando la JSP viene tradotta.

Ad es.:

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

Le espressioni: si usano i delimitatori `<%= e %>` per valutare le espressioni java. Il risultato viene convertito in stringa ed inserito al posto del tag nella pagina.

Ad es.:

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

Diventa una cosa del tipo:

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

Le Scriptlet: si usa il tag `<% %>` per aggiungere un frammento di codice Java eseguibile dalla JSP. Solitamente permette di inserire logiche di controllo di flusso nella produzione della pagina. L'insieme di tutti gli scriptlet in una JSP devono definire un blocco logico completo di codice Java.
Ad es.:

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

Le direttive: sono comandi JSP valutati a tempo di compilazione. Le più importati e di cui si farà una trattazione più dettagliata in seguito sono: `page`, `include` e `taglib`. È importante segnalare che non producono alcun output visibile.
Ad es.:

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

La direttiva `page`

Permette di importare di importare package, dichiarare pagine di errore, definire modello di esecuzione JSP relativamente alla concorrenza, ecc.
La sintassi:

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

- `language="java"` definisce il linguaggio di scripting nelle parti dinamiche, l'unico valore ammesso è Java (allo stato attuale)
- `import="{package.class|package.*},..."` lista di package da importare. Gli import più comuni sono impliciti (`java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*`, `javax.servlet.http.*`)
- `session="true|false"` indica se la pagina fa uso o no della sessione (se è a `false` non si può usare l'oggetto session)
- `buffer="none|8kb|sizekb"` dimensione in KB del buffer di uscita
- `autoFlush="true|false"` dice se il buffer viene svuotato automaticamente quando è pieno. Se è `false` e il buffer è pieno viene lanciata un'eccezione.

- `isThreadSafe="true|false"` indica se il codice della pagina è thread-safe. Se vale `false` le chiamate alla JSP vengono serializzate
- `info="text"` testo di commento (può essere letto con `Servlet.getServletInfo()`)
- `error-page="relativeURL"` indirizzo della pagina a cui vengono inviate le eccezioni
- `isErrorPage="true|false"` indica se la JSP corrente è una pagina di errore (si può utilizzare l'oggetto eccezione solo se l'attributo è `true`)
- `contentType="mimeType [;charset=charSet]" | "text/html; charset=ISO-8859-1"` indica il tipo MIME e il codice di caratteri usato nella risposta

I valori in grassetto indicano i valori di default.

La direttiva *include*

Serve ad includere il contenuto del file specificato. È possibile nidificare un numero qualsiasi di inclusioni, l'inclusione viene fatta a tempo di compilazione (eventuali modifiche al file incluso non determinano una ricompilazione della pagina corrente).

Ad es.:

```
<%@ include file="/shared/copyright.html"%>
```

La direttiva *taglib*

Le JSP permettono di definire dei tag custom oltre a quelli predefiniti. Una taglib è una collezione di questi tag non standard realizzata mediante classe Java.

La sintassi è: `<%@ taglib uri="tagLibraryURI" prefix="tagPrefix"%>`

BUILT-IN OBJECTS

Le specifiche JSP forniscono 9 oggetti built-in utilizzabili senza dover creare istanze, rappresentano utili riferimenti ai corrispondenti oggetti Java.

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

L'oggetto **page** rappresenta l'istanza corrente della servlet, ha come tipo l'interfaccia `HTTPJspPage` che discende da `JSP page`, la quale a sua volta estende `Servlet`, quindi può essere utilizzato per accedere a tutti i metodi definiti nelle servlet. Ad es.:

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

Risulta:

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

L'oggetto config contiene la configurazione della servlet (parametri di inizializzazione), è un oggetto poco usato in quanto nelle JSP sono poco usati i parametri di inizializzazione. I metodi referenziabili tramite l'oggetto `config` sono:

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

L'oggetto request È l'oggetto request che viene passato al metodo `service()` della servlet. Rappresenta la richiesta alla pagina JSP, consente l'accesso a tutte le informazioni relative alla richiesta HTTP. Ad es.:

```
<% String xStr = request.getParameter("num");
    try{
        long x = Long.parseLong(xStr); %>
        Fattoriale: <%= x %>! = <%= fact(x) %>
    <%}
    catch (NumberFormatException e) { %>
        Il parametro <b>num</b>non contiene un valore intero.
    <%} %>
```

Alcuni metodi di questo oggetto sono:

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

I metodi sono gli stessi identici dell'oggetto request delle Servlet

L'oggetto response è l'oggetto legato all'I/O della pagina JSP. Rappresenta la risposta che viene restituita al client. Consente di inserire nella risposta diverse informazioni, per esempio: content type ed encoding, eventuali header di risposta, URL Rewriting e i cookie. I metodi associati a questo oggetto sono gli stessi legati all'oggetto response passato alla Servlet nel metodo `service()`:

- `public void setHeader(String headerName, String headerValue)` imposta header
- `public void setDateHeader(String name, long millisecs)` imposta data
- `addHeader, addDateHeader, addIntHeader` aggiungono nuova occorrenza di un dato header
- `setContentType` determina content-type
- `addCookie` consente di gestire i cookie nella risposta
- `public PrintWriter getWriter()` restituisce uno stream di caratteri (un'istanza di `PrintWriter`)
- `public ServletOutputStream getOutputStream()` restituisce uno stream di byte (un'istanza di `ServletOutputStream`)

Ad es.:

```
<%
    response.setDateHeader("Expires", 0);
    response.setHeader("Pragma", "no-cache");
    if (request.getProtocol().equals("HTTP/1.1")){
        response.setHeader("Cache-Control", "no-cache");
    }
%>
```

L'oggetto out È legato all'I/O della pagina JSP: è uno stream di caratteri e rappresenta lo stream di output della pagina. I metodi associati a quest'oggetto sono:

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

Esempio di uso dell'oggetto out:

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

L'oggetto session Fornisce informazioni sul contesto di esecuzione della JSP in termini di sessione utente. Ricorda che l'attributo `session` della direttiva `page` deve essere `true` affinché JSP partecipi alla sessione. I metodi associati a quest'oggetto sono:

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

Un esempio dell'uso dell'oggetto session:

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

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 app a cui JSP appartiene. Consente di interagire con l'ambiente di esecuzione:

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

L'oggetto pageContext Fornisce informazione sul contesto di esecuzione della pagina JSP, rappresenta l'insieme degli oggetti built-in di una JSP: consente l'accesso a tutti gli oggetti impliciti e ai loro attributi, inoltre consente il trasferimento del controllo ad altre pagine.

L'oggetto exception È l'oggetto connesso alla gestione degli errori, rappresenta l'eccezione che non viene gestita in un blocco catch. Non è automaticamente disponibile in tutte le pagine ma solo nelle Error Page (quelle dichiarate con l'attributo `errorPage` settato a true).

Ad esempio:

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

AZIONI

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

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

Ad esempio:

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

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

La sintassi è: `<jsp:forward page="localURL" />`

L'attributo **page** è anche generabile dinamicamente: `<jsp:forward page='<%= "message"+statusCode+".html"%>' />`

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

Attenzione: forward è possibile solo se non è stato prodotto alcun output.

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

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

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

Trasferisce temporaneamente controllo ad un'altra pagina.

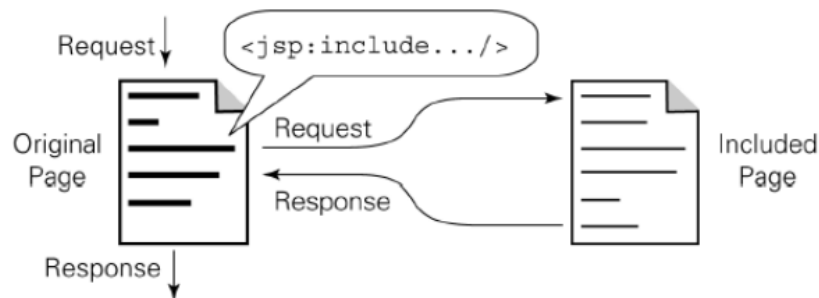
L'attributo **page** definisce l'URL della pagina da includere.

L'attributo **flush** stabilisce se sul buffer della pagina corrente debba essere eseguito flush prima di effettuare l'inclusione.

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

La sintassi è: `<jsp:include page="localURL" flush="true" />` Come per la forward si possono aggiungere dei parametri all'oggetto request tramite `<jsp:param>`. Ad esempio:

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



JAVABEAN

Scriptlet ed espressioni consentono uno sviluppo centrato sulla pagina, il che è uno svantaggio quando si vuole dividere la logica applicativa e la presentazione dei contenuti in quanto questo tipo di modello non lo consente e purtroppo le applicazioni complesse necessitano di maggiore modularità ed estensibilità tramite un'architettura a più livelli.

A tal fine le JSP consentono anche lo sviluppo basato su un modello a componenti.

Il modello a componenti di base Java è basato sui **JavaBeans**.

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

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

Le Proprietà

Le proprietà sono elementi dello stato del componente che vengono esposti in modo protetto. In Java sono solo una convenzione: sono coppie di metodi di accesso che seguono regole di denominazione.

La proprietà (per esempio) **prop** è definita da due metodi **getProp()** e **setProp()**, logicamente il tipo del parametro passato a **setProp()** e il valore di ritorno di **getProp()** devono essere uguali. Se si definisse solo il metodo get si avrebbe una proprietà in sola lettura.

Per le proprietà di tipo **boolean** la regola è leggermente diversa: il metodo in scrittura rimane uguale, mentre il metodo in lettura invece che essere **getProp()** è **isProp()**.

Esiste, inoltre, la possibilità di definire proprietà indicizzate per rappresentare collezioni di valori. In questi casi sia get che set prevedono il parametro che fa da indice. Ad esempio **String getItem(int index)** e **String setItem(int index, String value)** definiscono la proprietà indicizzata **String item[]**.

Componenti e container

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

I container non conoscono a priori i componenti che devono gestire quindi interagiscono con loro mediante meccanismi di tipo dinamico (spesso reflection).

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

L'obbligo del costruttore di default ha proprio lo scopo di consentire la creazione dinamica delle istanze.

Un esempio di JavaBean che espone l'ora corrente in ore e minuti (in sola lettura):

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

JSP e JavaBean

Le JSP prevedono una serie di tag per agganciare un bean e utilizzare le sue proprietà all'interno della pagina.

I tag sono di 3 tipi:

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

Esempio dell'uso di un bean:

```
<jsp:useBean id="user" class="RegisteredUser" scope="session"/>
<jsp:useBean id="news" class="NewsReports" scope="request">
    <jsp:setProperty name="news" property="category" value="fin."/>
    <jsp:setProperty name="news" property="maxItems" value="5"/>
</jsp:useBean>
<html>
    <body>
        <p>Bentornato
        <jsp:getProperty name="user" property="fullName"/>,
        la tua ultima visita è stata il
        <jsp:getProperty name="user" property="lastVisitDate"/>.
        </p>
        <p>
        Ci sono <jsp:getProperty name="news" property="newItems"/>
        nuove notizie da leggere.</p>
    </body>
</html>
```

Tempo di vita di un bean

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

Tag `jsp:useBean` Inizializza e crea il riferimento al bean. Gli attributi principali sono: l'*id* che rappresenta il nome con cui l'istanza del bean verrà indicata nel resto della pagina, la *class* è la classe Java che definisce il bean e lo *scope* che definisce l'ambito di accessibilità dell'oggetto (di default è **page**). Sintassi:

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

Tag `jsp:getProperty` Consente l'accesso alle proprietà del bean. Produce come output il valore della proprietà del bean. Non ha mai body e ha solo 2 attributi: *name* che rappresenta il nome del bean a cui fa riferimento e *property* che è il nome della proprietà di cui si vuole leggere il valore. Sintassi:

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

Tag `jsp:setProperty` Consente di modificare il valore delle proprietà del bean. Gli attributi di questo tag sono il *name* del bean, la *property* da modificare e il *value* da impostare. Sintassi:

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

Proprietà indicizzate I tag per JavaBean non supportano proprietà indicizzate però un bean è un normalissimo oggetto Java ed è quindi possibile accedere a metodi e variabili. Ad es:

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

AJAX

L'utilizzo di DHTML (JS/Eventi + DOM + CSS) delinea un nuovo modello per applicazioni web, ci si avvicina a un modello a eventi simile a quello delle app tradizionali.

A livello concettuale abbiamo, però, due livelli di eventi:

- **Eventi locali** che portano ad una modifica diretta DOM da parte di Javascript e quindi a cambiamento locale della pagina
- **Eventi remoti** ottenuti tramite ricaricamento della pagina che viene modificata lato server in base ai parametri passati in GET o POST

Il ricaricamento della pagina per rispondere ad un'interazione con l'utente pre il nome di **postback**.

Quando si lavora con applicazioni desktop siamo abituati a un livello elevato di interattività e le applicazioni sono in grado di reagire in modo rapido ed intuitivo ai comandi. Le applicazioni web tradizionali, invece, espongono un modello di interazione rigido: questo tipo di modello prende il nome di “click, wait and refresh” proprio perchè è necessario il refresh della pagina da parte del server per la gestione di un qualunque evento. È ancora un modello **sincrono**: l'utente effettua una richiesta e deve attendere la risposta dal server.

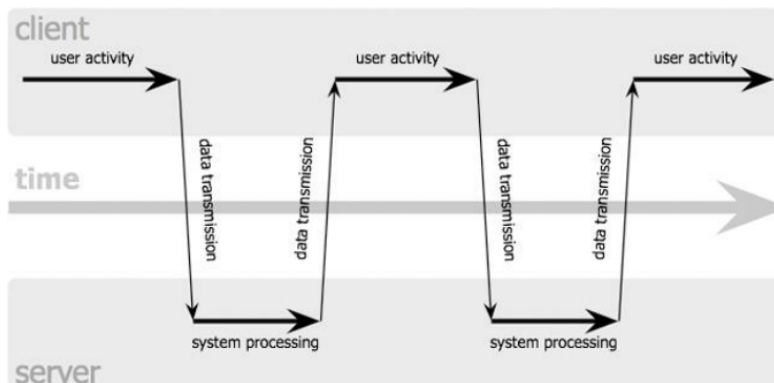


Figure 7: Modello di interazione classico

AJAX E ASINCRONICITÀ

AJAX non è un acronimo ma spesso viene interpretato come Asynchronous Javascript And Xml. Non è una tecnologia per se ma è basato su tecnologie standard e combinate insieme per realizzare un modello di interazione più ricco.

AJAX punta a supportare applicazioni user friendly con elevata interattività (RIA, Rich Interface Application), infatti l'idea di base di AJAX è quella di consentire agli script JS di interagire direttamente con il server.

L'elemento centrale è l'utilizzo di un oggetto JS: `XMLHttpRequest`, il quale consente di ottenere dati dal server senza ricaricare l'intera pagina, difatti realizza una comunicazione **asincrona** fra client e server (il client non interrompe l'interazione con l'utente anche quando è in attesa di risposte dal server).

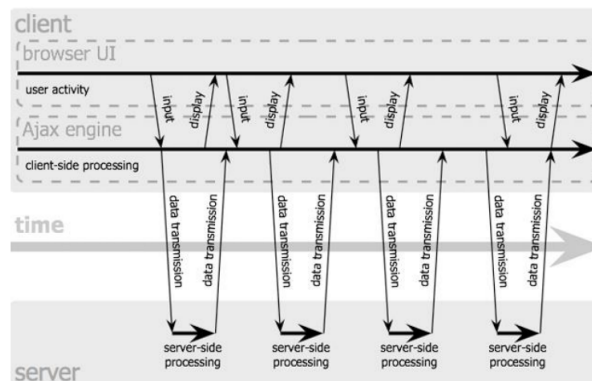


Figure 8: Modello di interazione tramite AJAX

Una tipica sequenza **AJAX** consta di:

1. Si verifica un evento determinato dall'interazione fra utente e pagina web
2. L'evento comporta l'esecuzione di una funzione JS in cui:
 - Si istanzia un oggetto della classe `XMLHttpRequest`
 - Lo si configura: si associa una funzione di callback, ...
 - Si effettua la chiamata asincrona al server
3. Il server elabora la richiesta e risponde al client
4. Il browser invoca la funzione di callback che:
 - Elaboro il risultato
 - Aggiorna il DOM della pagina per mostrare i risultati dell'elaborazione

FOCUS SU XMLHTTPREQUEST

È l'oggetto `XMLHttpRequest` che effettua la richiesta di una risorsa via HTTP al server web. È importante sottolineare che NON sostituisce URI corrente con quello della richiesta e NON provoca cambio di pagina, in più può inviare parametri sotto forma di variabili (come nei form).

Può effettuare richieste sia di tipo POST che di tipo GET, esse possono essere:

- Sincrone: bloccano il flusso di esecuzione del codice JS
- Asincrone: NON interrompono il flusso di esecuzione del codice JS né le operazioni dell'utente sulla pagina (quindi hanno un thread dedicato)

I browser recenti supportano XMLHttpRequest come oggetto nativo quindi per crearlo basta scrivere:

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

Invece la gestione della compatibilità con browser molto vecchi complica le cose: difatti alcune versioni di IE lo supportano come oggetto ActiveX (e non nativo), in più solo dalla versione 4 e, per altro, in modi differenti. Per essere compatibili con ogni versione di browser ancora installata bisogna procedere nel seguente modo:

```
function myGetXmlHttpRequest(){
    var xhr = false;
    var activeXopt = new Array("Microsoft.XmlHttp", "MSXML4.XmlHttp",
    "MSXML3.XmlHttp", "MSXML2.XmlHttp", "MSXML.XmlHttp" );
    // prima come oggetto nativo
    try
        xhr = new XMLHttpRequest();
    catch (e) { }
    // poi come oggetto activeX dal più al meno recente
    if (! xhr){
        var created = false;
        for (var i = 0; i < activeXopt.length && !created; i++){
            try {
                xhr = new ActiveXObject( activeXopt[i] );
                created = true;
            }
            catch (e) { }
        }
    }
    return xhr;
}
```

Attenzione: per motivi di sicurezza XMLHttpRequest può essere utilizzata solo verso il dominio da cui proviene la risorsa che la utilizza.

La lista dei metodi disponibili dipende da browser a browser ma in generale si usano quelli presenti in Safari (sottoinsieme più limitato ma comune a tutti i browser che supportano AJAX):

- open()
- setRequestHeader()
- send()
- getResponseHeader()
- getAllResponseHeaders()
- abort()

Metodo open() Ha lo scopo di inizializzare la richiesta da formulare al server. Lo standard W3C prevede 5 parametri di cui 3 opzionali: open (method, uri [,async] [,user] [,password]).

L'uso più comune è: open (method, uri, true) dove method è una stringa e assume i valori "get" o "post", uri è una stringa che identifica la risorsa da ottenere ed, infine, async è un booleano che deve essere impostato a true per indicare al metodo che la richiesta da fare è di tipo asincrono.

Metodo `setRequestHeader()` Consente di impostare gli header HTTP della richiesta da inviare:

- viene invocato più volte (una per ogni header da impostare)
- per una richiesta GET gli header sono opzionali
- sono, invece, necessari per impostare la codifica utilizzata nelle richieste post
- è comunque importante impostare l'header `connection` di solito al valore `close`

Sintassi: `setRequestHeader(nomeheader, valore)`

Metodo `send()` Consente di inviare la richiesta al server. Non è bloccate in attesa della risposta se il parametro `async` di `open` è stato impostato a `true`. Prende come parametro una stringa che costituisce il body.\ Sintassi: `send(body)`

Un esempio di richiesta di tipo GET potrebbe essere:

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

Mentre una di tipo POST:

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

Stato e risultati della richiesta vengono memorizzati dall'interprete JS all'interno dell'oggetto `XmlHttpRequest` durante la sua esecuzione. Le proprietà comuni a vari browser sono:

- `readyState`
- `onreadystatechange`
- `status`
- `statusText`
- `responseText`
- `responseXML`

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

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

Attenzione, però, che questo ordine non è sempre identico e sfruttabile allo stesso modo su tutti i browser. L'unico stato supportato da tutti i browser è il 4.

Proprietà `onreadystatechange` Come si è detto l'esecuzione del codice non si blocca sulla `send` in attesa di risultati, quindi per gestire la risposta si adotta un approccio ad eventi. Occorre registrare la funzione di callback che viene richiamata in modo asincrono ad ogni cambio di stato della proprietà `readyState`. Bisogna fare attenzione ad evitare comportamenti imprevedibili, per fare ciò basta fare l'assegnamento prima della `send()`. La sintassi è:

```
//METODI EQUIVALENTI:
/*1*/
xhr.onreadystatechange = nomefunzione
/*2*/
xhr.onreadystatechange = function(){istruzioni}
/*3*/
xhr.onreadystatechange = () => {istruzioni}
```

Proprietà `status` Contiene un valore intero corrispondente al codice HTTP dell'esito della richiesta:

- 200 in caso di successo (l'unico in base al quale i dati ricevuti in risposta possono essere ritenuti corretti e significativi)
- Possibili altri valori (in particolare d'errore: 403, 404, 500, ...)

Proprietà `statusText` Contiene invece una descrizione testuale del codice HTTP restituito dal server. Per esempio:

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

Proprietà `responseText` Stringa che contiene i dati restituiti dal server nel body della risposta HTTP, è disponibile **solo** a interazione ultimata, quindi quando `readyState === 4`.

Proprietà `responseXML` Come la precedente contiene i dati restituiti dal server nel body della risposta ma convertito, se possibile, in documento XML. Questo è utile perchè consente la facile navigazione via JS. Nel caso il body della HTTP response non sia convertibile in un documento XML (e quindi non sia un documento ben formato) questa proprietà vale `null`.

Metodi `getResponseHeader()` e `getAllResponseHeaders()` Consentono di leggere gli header HTTP che descrivono la risposta del server:

- Sono utilizzabili solo nella funzione di callback
- Possono essere invocati sicuramente in modo safe solo a richiesta conclusa (`readyState==4`)
- In alcuni browser possono essere invocati anche in fase di ricezione della risposta (`readyState==3`)

La sintassi è: `getAllResponseHeaders()` e `getResponseHeader(header_name)`.

Ruolo della funzione di callback Usa `readyState` per leggere lo stato di avanzamento della richiesta, per questo è chiamata ad ogni variazione di tale proprietà. Inoltre, usa `status` per verificare l'esito della richiesta. Ha accesso agli headers tramite i due metodi descritti sopra (`getAllResponseHeaders()` e `getResponseHeader()`). Se `readyState === 4` \implies può leggere il contenuto della risposta con `responseText` e/o `responseXML`.

VANTAGGI E SVANTAGGI DI AJAX

Sicuramente si guadagna in espressività ma si perde la linearità dell'interazione.

Mentre l'utente è all'interno della stessa pagina le **richieste** sul server possono essere **numerose** ed **indipendenti**.

Il **tempo di attesa** passa in secondo piano o non è avvertito affatto.

Infine possono presentarsi criticità (meglio descritte in seguito) sia per l'utente che per lo sviluppatore.

Criticità nell'interazione con l'utente

Le richieste AJAX permettono all'utente di continuare a interagire con la pagina ma non necessariamente lo informano di che cosa stia succedendo e possono durare troppo. L'effetto finale è un possibile disorientamento dell'utente. Solitamente si agisce di conseguenza su due fronti per limitare i comportamenti impropri a livello utente:

- Si rende visibile in qualche modo l'andamento della chiamata tramite barre di scorrimento o info utente
- Si interrompono le richieste che non terminano in tempo utile per sovraccarichi del server o momentanei problemi di rete (timeout)

Per interrompere le operazioni di invio e ricezione esiste il metodo `abort()`: non ha bisogno di parametri e termina immediatamente la trasmissione dei dati.

Attenzione: non ha senso chiamarlo all'interno della funzione di callback in quanto se `readyState` non cambia il metodo non viene richiamato, e, se la risposta si fa attendere, `readyState` non cambia.

Solitamente si crea un'altra funzione da far richiamare in modo asincrono al sistema mediante il metodo:

```
setTimeout(funzioneAsincronaPerAbortire, timeOut);
```

Al suo interno si valuta se continuare l'attesa o abortire l'operazione.

Criticità per il programmatore

Alcuni aspetti critici per il programmatore sono:

- L'accrescimento della complessità delle web app
- La ripartizione della logica di presentazione fra client-side e server-side
- Le applicazioni AJAX pongono problemi di debug, test e mantenimento

Aspetti secondari ma non meno importanti sono:

- Il test di codice JavaScript è complesso
- Il codice JavaScript ha problemi di modularità
- I toolkit AJAX sono molteplici e solo recentemente hanno raggiunto una discreta maturità
- Mancanza di standardizzazione di `XMLHttpRequest` e assenza di supporto nei vecchi browser

GESTIONE DELLA RISPOSTA

Spesso i dati scambiati fra client e server sono codificati in XML: AJAX come abbiamo visto è in grado di ricevere documenti XML. In particolare è possibile elaborare i documenti XML ricevuti utilizzando API W3C DOM. Il modo con cui si opera su dati in formato XML è analogo a quello visto per ambienti Java: si usa un parser e si accede agli elementi di nostro interesse e per visualizzare i contenuti ricevuti si modifica il DOM della pagina HTML.

La domanda che a questo punto sorge spontanea è: ma XML è la scelta giusta?

Si può dimostrare che lo scambio di dati in formato XML fra client e server porta alla generazione ed utilizzo di quantità di byte piuttosto elevate e non ottimizzate, in più non è semplice da leggere e da mantenere ed è oneroso in termini di elaborazione (JS è interpretato).

E allora esiste un'alternativa più efficiente e semplice da manipolare tramite AJAX?

La risposta è sì. Il formato alternativo prende il nome di JSON.

JSON

JSON (JavaScript Object Notation) è un formato per lo scambio dei dati considerato molto più comodo di XML:

- Leggero in termini di quantità di dati scambiati
- Molto semplice ed efficiente da elaborare da parte del supporto runtime al linguaggio di programmazione (in particolare JS)
- Ragionevolmente semplice da leggere per l'operatore umano

È largamente supportato dai maggiori linguaggi di programmazione e si basa sulla notazione usata per le costanti oggetti (object literal) e le costati array (array literal) in JS.

Oggetti e costanti oggetto In è possibile creare un oggetto in base a una costante oggetto:

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

Che è del tutto equivalente a:

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

Array e costanti array In modo analogo è possibile creare un array utilizzando una costante di tipo array:

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

Del tutto uguale a:

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

Si possono anche creare oggetti che contengono array:

```
var Beatles =
{
  "Paese" : "Inghilterra",
  "AnnoFormazione" : 1959,
  "TipoMusica" : "Rock",
  "Membri" : ["Paul", "John", "George", "Ringo"]
}
```

Infine si possono creare 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"]
}
]
```

La sintassi JSON si basa su quella della costanti oggetto e array JS: un oggetto JSON non è altro che una stringa equivalente a una costante oggetto di JS.

Per la conversione da stringa ad oggetto JS mette a disposizione la funzione `eval()` che invoca l'interprete per la traduzione della stringa passata come parametro: la sintassi JSON è un sottoinsieme di JS, quindi con `eval()` possiamo trasformare una stringa JSON in un oggetto, tuttavia la sintassi della stringa passata deve essere: '(espressione)' quindi bisogna racchiudere la stringa JSON fra parentesi tonde. Ad esempio:

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

L'uso di `eval()` presenta diversi rischi: la stringa passata come parametro potrebbe contenere codice malevolo, infatti di solito si preferisce utilizzare parser che traducono solo oggetti JSON e non espressioni JS di qualunque tipo. Alcuni dei più diffusi sono:

- Google GSON
- jabsorb

Ad esempio il parser jabsorb espone l'oggetto JSON con 2 metodi:

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

JSON e AJAX Si riporta un esempio di un'interazione client-server in cui il client vuole trasferire un oggetto JSON.

Sul lato client:

- Si crea un oggetto JavaScript e si riempiono le sue proprietà con le informazioni necessarie
- Si usa `JSON.stringify()` per convertire l'oggetto in stringa JSON
- Si usa la funzione `encodeURIComponent()` per convertire la stringa in un formato utilizzabile in una richiesta HTTP (vedi esercitazione su AJAX)
- Si manda la stringa al server mediante `XMLHttpRequest` (stringa viene passata come variabile con GET o POST)

Sul lato server:

- Si decodifica la stringa JSON e la si trasforma in oggetto Java utilizzando un apposito parser
- Si elabora l'oggetto
- Si crea un nuovo oggetto Java che contiene dati della risposta
- Si trasforma l'oggetto Java in stringa JSON usando il parser suddetto
- Si trasmette la stringa JSON al client nel corpo della risposta HTTP: `response.out.write(strJSON);`

Sul lato client al momento della ricezione:

- Si converte la stringa JSON in un oggetto Javascript usando `JSON.parse()`
- Si usa liberamente l'oggetto per gli scopi desiderati

Parsing JSON in Java: GSON Gson è una libreria java per il parsing/deparsing di oggetti JSON. È stata realizzata da Google per l'esclusivo sviluppo di prodotti interni, ora Gson è libreria open source. È una libreria molto potente e largamente utilizzata sia in ambito accademico che industriale. Alcuni dei suoi punti di forza sono:

- Fornisce dei metodi semplici e facili da usare per “conversione” Java-JSON e viceversa
- Genera output JSON compatti e leggibili
- Consente rappresentazioni custom per gli oggetti
- Consente la conversione da/a JSON di oggetti Java immutabili pre-esistenti (non occorre modificare sorgente)
- Supporta oggetti di complessità arbitraria

Esempio di utilizzo di GSON (in versioni recenti, a partire da 2.8):

Inizializzazione dell'oggetto Gson:

```
Gson g = new Gson();
```

Serializzazione di un oggetto:

```
Person santa = new Person("Santa", "Claus", 1000);  
g.toJson(santa);
```

Deserializzazione di un oggetto:

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

UN PICCOLO RIASSUNTO

- AJAX consente gestione asincrona
- AJAX aggiunge un nuovo elemento al modello a eventi
- L'uso di XMLHttpRequest rappresenta una modalità alternativa per gestire gli eventi remoti
- Si può adottare la tecnologia in modo più “radicale” e utilizzare solo AJAX eliminando i caricamenti di pagina (Single Page Applications)
- Nei casi più comuni però AJAX e la modalità di navigazione tradizionale convivono

Si ha quindi:

- Una modalità per gestire gli eventi a livello locale
- Due modalità per gestire gli eventi remoti (postback e XMLHttpRequest)