Programmazione per il Web

A.A. 2016/2017

Corso di Laurea in Informatica

Igor Melatti

Sapienza Università di Roma
Dipartimento di Informatica

SAPIENZA
UNIVERSITÀ DI ROMA

Informazioni Generali sul Corso

- Docente: Igor Melatti
- Dove potete trovare materiale: `http://twiki.di.uniroma1.it/twiki/view/PW/2016_2017`
  - ci saranno anche informazioni sugli esami ed in generale sull'intero corso
- Dove potete trovare materiale sull'anno scorso: `http://twiki.di.uniroma1.it/twiki/view/PW/2015_2016`
  - ci sono tutte le note su tutte le lezioni dell'anno scorso
  - quest'anno nuove slides, ma gli argomenti restano gli stessi
- Sono previste 25 lezioni (festività comprese), se non ne salta nessuna
  - ogni settimana 2 lezioni, una da 3 e una da 2 ore

Regole per gli Esami

- Ci sono 5 appelli d'esame nel corso dell'anno, più due "straordinari"
  - uno a giugno (subito dopo la fine del corso), uno fine giugno/inizio luglio e uno a settembre nel 2017, e due a gennaio/febbraio 2018
  - i due appelli straordinari (verso aprile il primo e verso novembre il secondo) sono riservati a studenti fuoricorso, ripetenti, part-time e lavoratori

Regole per gli Esami

- Per partecipare agli appelli straordinari, occorre seguire le istruzioni pubblicate qui: `http://www.studiareinformatica.uniroma1.it/appelli-d-esame`
- Verranno ammessi agli appelli straordinari tutti e soli gli studenti che la segreteria didattica comunicherà al docente
  - una volta ottenuto l'ok della segreteria, occorrerà iscriversi su Infostud
- Per gli appelli "normali", è necessaria e sufficiente la prenotazione su Infostud
  - le iscrizioni su Infostud si chiuderanno sempre 2 giorni prima del giorno dello scritto

Regole per gli Esami

- Ciascun appello d'esame è composto da uno scritto e da un orale
- È possibile partecipare a qualsiasi numero di esami (scritti e/o orali) nel corso dell'anno accademico
- Per superare l'esame scritto occorre aver preso almeno 18
  - fa fede il voto dell'*ultimo* esame scritto sostenuto
  - nota bene: questo significa che non superare un esame scritto invalida eventuali esami scritti superati in precedenza
- Il voto dell'esame scritto sarà al massimo 25
  - sono possibili eccezioni a discrezione del docente
- Chiunque abbia superato lo scritto può verbalizzare direttamente il voto dello scritto stesso, senza un esame orale
  - sono possibili eccezioni a discrezione del docente (in caso di dubbio di copiatura)

Regole per gli Esami

- Ogni studente può chiedere di alzare il voto dello scritto tramite lo svolgimento di un progetto
  - che andrà discusso in un esame orale
- Chi venga bocciato all'orale dovrà nuovamente superare un esame scritto
- L'orale può essere anche sostenuto (a discrezione dello studente) in un appello diverso da quello in cui ha superato lo scritto
  - il progetto andrà presentato nello stesso appello in cui si vuole fare l'orale
- Occorre sempre iscriversi su Infostud, sia per gli esami scritti che per quelli orali, con la seguente eccezione:
  - se si fa l'esame orale nello stesso appello in cui si fa lo scritto, basta iscriversi una volta sola (prima dello scritto)

Regole per gli Esami: lo Scritto

- Le domande verteranno sempre sull'intero programma del corso
- Si tratta di un compito a quiz da fare direttamente al computer, in laboratorio
- Nel caso in cui ci siano più iscritti all'esame che posti in laboratorio, l'esame si farà in più turni, comunicati sul sito del corso
- Ci saranno 40 domande da fare in 30 minuti
- Per ogni domanda, ci sono 4 opzioni, delle quali una sola è vera
- Tutti gli studenti hanno lo stesso compito, ma con le domande (e le opzioni) mischiate

Regole per gli Esami: lo Scritto

- Per ciascuno viene calcolato il punteggio come $2E - S$, con E numero di risposte esatte ed S numero di risposte sbagliate
  - quindi il punteggio va da -40 a 80
- Per assegnare un voto da insufficiente a 25 a ciascun punteggio, si valuteranno i punteggi di tutti i partecipanti all'esame, seguendo una distribuzione a campana di Gauss
  - conseguenza: se ci sono copiature, molti compiti avranno punteggi vicini, e quindi il voto si abbassa. Morale: non conviene copiare
  - non è necessario rispondere a tutte le domande
  - chi fa il compito migliore rispetto agli altri ha il punteggio più alto
- Per gli appelli successivi al primo, a questa valutazione partecipano anche i punteggi ottenuti da tutti gli studenti degli appelli precedenti
  - morale: conviene cercare di passare il compito il prima possibile

Regole per gli Esami: l'Orale

- Per accedere all'orale è necessario aver superato uno scritto (anche in appelli precedenti)
- L'orale consiste nella discussione di un progetto *individuale*
  - le specifiche di questo progetto verranno pubblicate sul sito del corso
- Verrà approntata una pagina di sottomissione dei progetti e verrà comunicata, per ogni sessione, una data di scadenza
- Potrà fare l'orale in una data sessione solo chi abbia consegnato in tempo il progetto

Regole per gli Esami: l'Orale

- Chi vuole fare l'orale, deve presentarsi con il proprio computer
- Su tale computer dovrà essere installato e funzionante il progetto consegnato
- L'orale consisterà nel dimostrare di conoscere il progetto consegnato (codice sorgente compreso)
- In caso di superamento dell'orale, al progetto verrà assegnato un voto da 0 a 12, da sommare al voto dello scritto
- In caso di orale non superato, sarà necessario rifare lo scritto
- Di norma, l'esame orale inizia subito dopo l'esame scritto, con la comunicazione da parte del docente di un calendario di massima
- In tale calendario di massima, a discrezione del docente, gli orali possono essere divisi su più giorni

Programmazione per il Web: Esempi

- `http://wwwusers.di.uniroma1.it/~melatti/`
  - pagina Web *statica*
- `http://smarthg.di.uniroma1.it/`
  - pagina Web *dinamica* lato client
  - aprendo la pagina, è visibile il codice Javascript
  - più che altro, per ottenere effetti "simpatici", tipo i menù
- `https://validator.w3.org/`
  - pagina Web *dinamica* lato server
  - in generale, ci potrebbe anche essere codice Javascript lato client, ma il codice lato server non viene mostrato
  - indistinguibile da una pagina statica o dinamica di solo lato client, a meno di non essere gli sviluppatori della pagina

Programmazione per il Web

- Scrivere una pagina statica non è vera "programmazione per il Web"
- Scrivere una pagina dinamica, sia lato server che lato client, sì
- In questo corso, ci si concentrerà sulla parte più critica, che è quella lato *server*
- Per meglio comprenderla, occorre un po' di ripasso di Reti degli Elaboratori, nonché qualche chiarimento sui sistemi informativi basati sul Web

WIS: Web-based Information System

- Come dice il nome stesso: sistema informativo basato sul Web
  - *sistema informativo*: contiene informazioni di interesse per alcuni utenti
  - *basato sul Web*: per accedere a tali informazioni, si usano i protocolli (applicativi) del WWW (World Wide Web)
  - lo strumento privilegiato (attenzione: non l'unico!) per accedere a tali informazioni è il *browser Web*
    - Firefox, Google Chrome, Opera, Safari, MS Edge, ...

WIS: Web-based Information System

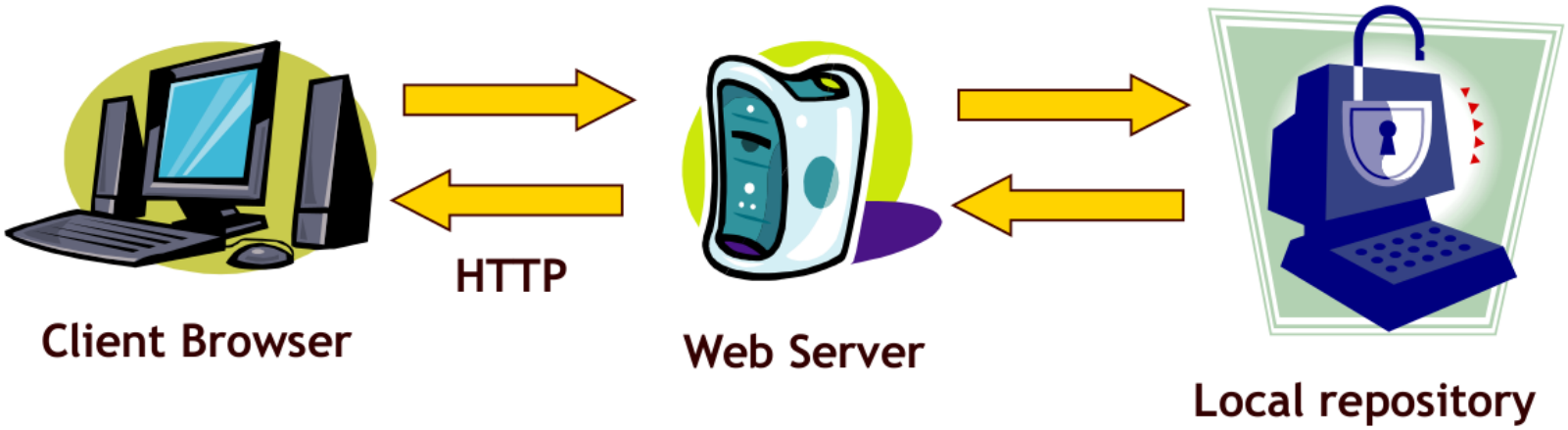
- Contiene normalmente un insieme di pagine Web, statiche o dinamiche (sia lato client che lato server)
- Nei casi più tipici, richiede l'interazione con un DBMS (DataBase Management System)
  - PostgreSQL, MySQL, DB2, MS SQLServer, ...
- L'interazione può essere non solo in lettura, ma anche in scrittura
  - sistemi transazionali
- Di tutto ciò, l'utente normale non è a conoscenza: è sufficiente il browser Web

Tipi di WIS

- Informativi: si limitano a fornire le informazioni quando vengono richieste
  - es.: Wikipedia (se non si è autori), tutte le pagine statiche
- Transazionali: permettono di modificare l'insieme delle informazioni
  - es.: Wikipedia (se si è autori), siti di banche, Infostud, ...
- Di comunicazione
  - es.: Messenger, Slack, ...

Architettura dei WIS

Il modello è l'architettura client-server



Architettura dei WIS: Local Repository

- Tipicamente, un DBMS relazionale
- Ma può essere qualsiasi fonte di informazioni
  - es.: file system, un DBMS non-relazionale
- Da notare che il Web browser, nella configurazione tipica, non ha modo di comunicare con il local repository
  - altrimenti, sarebbe facile attaccare il local repository stesso
  - e rubare o modificare informazioni importanti

Architettura dei WIS: Web Browser

- Software per visualizzare pagine Web
  - sfrutta il protocollo HTTP e il linguaggio di markup HTML

- Funzionamento tipico:
  - l'utente richiede un *URL* (Uniform Resource Location)
    - scrivendolo sulla barra degli indirizzi, o cliccando su un link
    - l'URL è tipicamente costruito come `[protocollo://]server[/risorsa_nel_server]`
  - tramite il protocollo `protocollo` (solitamente HTTP), viene inviata la richiesta al server per `risorsa_nel_server`
    - potrebbe essere necessario interrogare un DNS e svariate tabelle di instradamento

Architettura dei WIS: Web Browser

- Le risorse da visualizzare arrivano tipicamente come *pagine HTML*
  - Hyper-Text Markup Language

- Più in generale, si tratta di file il contenuto dei quali è identificato dal *MIME*
  - Multi-purpose Internet Mail Extensions
  - vedere `http://en.wikipedia.org/wiki/Internet_media_type`
  - è tutto ciò che si possa spedire con protocolli applicativi di rete, come HTTP, FTP, SMTP
  - ovvero: audio, video, immagini, più altre cose

Architettura dei WIS: Web Server

- Si tratta di un software che può essere installato su qualsiasi macchina sufficientemente potente
  - Esempi: Apache (multipiattaforma), IIS (Internet Information Service, solo sotto Microsoft), ...
- Minimalmente, consente di "pubblicare" pagine HTML statiche, mettendole in un'apposita directory del filesystem
  - nel caso di Apache: se installato sotto Linux, le pagine vanno tipicamente sotto `/var/www` (o, più recentemente, `/var/www/html`)
  - con Mac OS X, la directory dovrebbe essere `/Library/WebServer/Documents/`
  - con Windows, la directory dovrebbe essere `C:\Program Files\Apache Group\Apache2\htdocs`

Architettura dei WIS: Web Server

- Esempio sotto Linux: creando nel filesystem un file `/var/www/html/pagina.html`, è possible poi visualizzare questo file su un browser all'indirizzo `http://localhost/pagina.html`
- Esempio sotto Linux: creando nel filesystem un file `/var/www/html/dir1/dir2/page.html`, è possible poi visualizzare questo file su un browser all'indirizzo `http://localhost/dir1/dir2/page.html`
- I file `index.html` hanno un trattamento speciale:
  - un file `/var/www/html/dir1/dir2/index.html` può essere visualizzato in un browser sia all'indirizzo `http://localhost/dir1/dir2/` che all'indirizzo `http://localhost/dir1/dir2/index.html`

Architettura dei WIS: Web Server

- Da notare che è possibile aprire file HTML statici su un browser anche se non è installato un Web server: basta usare il procotollo FILE
  - basta usare il procotollo FILE
  - Ad esempio, un file `/home/utente/pagina.html` può essere aperto in un browser all'indirizzo `file:///home/utente/pagina.html`
- È possibile configurare Apache sotto Linux per farlo funzionare sulle home degli utenti
  - un file `/home/utente/public_html/dir1/index.html` potrà essere aperto su browser all'indirizzo `http://localhost/~utente/dir1`
  - con Mac OS X, la directory dovrebbe essere `Sites` anziché `public_html`

Architettura dei WIS: Web Server

- Se il computer su cui è installato Apache è raggiungibile dall'esterno all'indirizzo IP $x.y.w.z$, allora si può sostituire $x.y.w.z$ a `localhost` negli esempi di sopra
  - non ci devono essere firewall o altri programmi che bloccano la porta usata da Apache (di default, la numero 80)
- Meglio ancora, se è raggiungibile dall'esterno all'indirizzo mnemonico `macchina.dominio.it` si può sostituire `macchina.dominio.it` a `localhost` negli esempi di sopra
  - quindi, un file `/home/utente/public_html/dir1/index.html` potrà essere aperto su browser all'indirizzo `http://macchina.dominio.it/~utente/dir1`

Il Linguaggio HTML

- Definito formalmente dalle *recommendations* del W3C
  - World Wide Web Consortium, ente di standardizzazione fondato da Tim Berners–Lee nel 1994
  - `https://www.w3.org/TR/`
  - oltre che l'HTML, specifica anche i CSS e i vari protolli del WWW
  - compreso l'HTTP

Il Linguaggio XHTML

- Attualmente, esistono almeno 3 diversi standard:
  - HTML
    - detto anche HTML4, naturale evoluzione del primo HTML di Berners–Lee
    - molto "permissivo" sulla sintassi, permette ad esempio di non chiudere i tag in alcune situazioni
  - **XHTML** (dal 2000): eXtensible HTML
    - recepisce le regole sintattiche dell'XML: più severo nella sintassi
    - ma a parte questo, fa le stesse cose dell'HTML4
  - HTML5 (dal 2014)
    - ripristina la permessività di HTML4
    - ed aggiunge svariati nuovi tag, che non saranno oggetto di questo corso
    - per i browser che lo supportano, vedere `http://html5readiness.com/`

Il Linguaggio XHTML

- File di testo con codifica UTF-8 (ma si possono specificare altre codifiche)
  - se aperto con un editor di testo o con un browser, l'effetto sarà diverso
  - l'editor di testo mostra il *sorgente* della pagina, il browser *visualizza* (*render*) la pagina
  - tutti i browser permettono di aprire (a richiesta) anche il sorgente della pagina

Il Linguaggio XHTML

- Il concetto centrale è quello di *tag* (etichetta)
  - sintatticamente, un tag si apre con `<nometag>` e si chiude con `</nometag>`
  - tutti i tag aperti devono essere chiusi
  - tra un tag aperto ed uno chiuso può trovarsi qualsiasi testo, eventualmente contenente altri tag
  - se non c'è nessun testo, allora è ammessa la sintassi `<nometag/>`
- Un tag può contenere una lista di assegnamenti `nomeattributo="valore"`
  - tra `<nometag` e `>` o `/>`
  - anche `'` invece di `"`

Il Linguaggio XHTML

- Differenza fondamentale con XML (eXtensible Markup Language): l'insieme dei nomi di tag dell'XHTML è *fissato*
  - sono anche fissati gli attributi che ogni tag può avere, e come i tag possono essere annidati gli uni dentro gli altri
  - ad es., l'intero documento deve essere racchiuso in un tag `html`
- I browser sanno come interpretare ciascun tag dell'XHTML (e ogni suo attributo)
  - sanno anche che tutto ciò che è racchiuso tra `<!--` e `-->` è da considerarsi un commento, e non va visualizzato
  - uno stesso tag può essere interpretato in modo diverso a seconda di com'è annidato

Il Linguaggio XHTML

- Andare a capo nel sorgente di una pagina HTML non ha conseguenze sulla visualizzazione della pagina stessa
  - o meglio, l'andata a capo è considerata come uno spazio
  - ugualmente senza effetto sono gli spazi consecutivi o le tabulazioni
  - eccezioni: `<textarea>` e `<pre>` mantengono le spaziature così come scritte
- I tag non vengono visualizzati, ma servono a specificare modalità di visualizzazione
  - ad es.: `<b>` serve per la visualizzazione in grassetto

Tag notevoli: `<table>`

- Per ottenere pagine Web con un certo allineamento del testo (o delle immagini, o di altre cose embedded), si usa `<table border="0">`
  - le righe sono specificate dal tag `<tr>`, e contengono colonne specificate dal tag `<td>`
  - le colonne di righe diverse vengono allineate nella visualizzazione
  - attributo `rowspan`: una colonna che si espande su più righe
    - occorre poi scrivere altre righe, dentro altri tag `tr`, in cui si specificano solo le altre righe
  - attributo `colspan`: in una riga, una colonna che si espande anche sulle colonne seguenti
    - quindi occorre specificare meno colonne, $n - r$ se n è il numero totale di colonne della tabella ed r il valore di `colspan`
- `http://151.100.17.236/~melatti/h34_s11_d1_ha/`

Tag notevoli: `<form>`

- Per mandare informazioni aggiuntive alla richiesta di una pagina Web
  - fondamentale per le pagine Web dinamiche, e per la programmazione Web lato server
  - ad esempio, una tipica pagina di login viene realizzata con un `<form>`
- Contiene diversi tag `<input>`, di tipo diverso a seconda di quale input ci si attenda dall'utente
  - `text`: testo in una riga
  - `number`: un numero intero
  - `radio`: scelta multipla mutuamente esclusiva
  - `checkbox`: scelta multipla non mutuamente esclusiva
  - `submit`: pulsante per mandare le informazioni
  - `http://www.w3schools.com/tags/att_input_type.asp`
- Tutti i tag: `http://www.w3schools.com/tags/`

Caratteristiche del Protocollo HTTP

- Hyper-Text Transfer Protocol: protocollo per il trasferimento di iper-testi
  - le pagine Web sono iper-testi perché coniugano testo, immagini, video etc.
  - versione attuale: 1.1 (dal 1999)
- Livello applicativo della pila ISO/OSI
  - presuppone un protocollo di livello trasferimento orientato alla connessione: il TCP (Transfer Control Protocol)
- Meccanismo di richiesta/risposta, ovvero client/server
  - trasferimento solitamente mono-direzionale: il server manda una pagina Web al client
  - anche bi-direzionale se c'è un form: il client manda aggiunge informazioni alla richiesta spedita al server

Caratteristiche del Protocollo HTTP

- Basato sul meccanismo di *naming* degli URI
  - Uniform Resource Identifier
  - nel caso del WWW, gli URI diventano URL (Uniform Resource Location)
  - costituiti da un nome di un server e da un percorso all'interno di tale server
    - per localizzare il server servono sicuramente le tabelle di routing
    - se non si usa l'indirizzo IP (Internet Protocol, su cui si basa il TCP), allora occorre anche consultare un DNS (Domain Name System)
    - il percorso all'interno del server potrà essere fisico o virtuale, come si vedrà

Caratteristiche del Protocollo HTTP

- Le richieste del protocollo HTTP sono nativamente *senza stato*
  - ogni richiesta è a sé stante, ovvero indipendente da quali richieste siano fatte in precedenza
  - se si vuole introdurre uno stato, è necessario sfruttare opportuni meccanismi aggiuntivi rispetto all'HTML
    - li vedremo nelle lezioni che seguono: hidden inputs, sessioni, cookies
- Introdurre lo stato è spesso indispensabile: si pensi ad esempio ad un'applicazione Web per l'home banking
  - quando viene richiesta la pagina con l'estratto conto, è essenziale che essa dipenda da informazioni immesse al momento di fare login (username, password)
  - quindi, la risposta alla richiesta della pagina con l'estratto conto *deve dipendere* dalle richieste fatte in precedenza (quelle alla pagina di login)

Richiesta HTTP

È formata da:

- Un *verbo* (o *comando*): solitamente è sempre `GET` o `POST`, ma sono di uso comune anche `PUT`, `DELETE`, `OPTIONS`, `HEAD`, `TRACE`
  - `PUT`, `DELETE`, `OPTIONS` sono usati soprattutto nei RESTful services
- Un URL, senza l'indicazione del server
  - Un percorso, fisico o virtuale, all'interno del server
- La versione del protocollo in uso
- Esempio completo: `GET /main.html HTTP/1.1`
- Altre informazioni possono venire "appese" alla richiesta
  - dati inviati attraverso un form con `POST`, client usato...
  - esempio:

```
User-Agent:  Mozilla/4.0
Accept:  image/gif, image/jpeg, text/*, */*
```

Risposta HTTP

È formata da:

- un *header*
  - versione del protocollo
  - status code della risposta
    - 200: tutto a posto
    - 500: errore di risoluzione del nome
    - ...
  - software usato dal server (Apache, IIS, ...)
  - content-type della risposta
    - pagina HTML, pagina di testo, immagine, oggetto JSON...
- la risposta vera e propria, usualmente sotto forma di pagina HTML
  - ma in alcuni casi, come vedremo, c'è solo l'header
  - verrà interpretata dal browser usando content-type specificato nell'header

Verbi HTTP: `GET` e `POST`

- Originariamente, con `GET` si intendevano richieste read-only, con `POST` quelle read-write
  - lo spirito di questa distinzione è ritornato con le applicazioni RESTful
- Al giorno d'oggi, la distinzione tra i due verbi è netta, ma non ha nulla a che vedere con il loro significato nel linguaggio corrente
- Invece, la differenza sta nel *come* vengono inviate eventuali informazioni aggiuntive alla richiesta
  - con `GET`, vengono appese all'URL della richiesta, con un opportuno encoding (*query string*)
  - con `POST`, vengono appese all'interno della richiesta stessa, praticamente senza enconding
    - o meglio, con il solo encoding richiesto dai livelli inferiori del protocollo

Verbo `GET`: Conseguenze

- I parametri aggiuntivi della richiesta sono direttamente visibili nell'URL
  - non è quindi usabile per inviare password...
- La lunghezza delle informazioni inviabili è limitata
  - molti browser impongono un limite massimo alla lunghezza degli URL (come numero di caratteri)
  - solitamente, tale limite è 240 caratteri
  - non sarebbe quindi possibile inviare neanche un file di 1 KB (in generale, non si può fare l'*upload* di file)
- È possibile ripetere la richiesta semplicemente riusando l'URL
  - copia/incolla, bookmark, invio di link tramite email, reload della pagina
  - è per questo motivo che il `GET` viene usato...

Verbo POST: Conseguenze

- I parametri aggiuntivi della richiesta non sono direttamente visibili nell'URL
  - certo, si può sempre fare *sniffing*...
  - quindi, per inviare password, è meglio usare protocolli criptati (es. HTTPS)
- La lunghezza delle informazioni inviabili è illimitata
  - o meglio, limitata solo dalle impostazioni del server
  - in tale modo, si possono inviare anche file di decine di MB
- Non è possibile ripetere la richiesta semplicemente riusando l'URL
  - niente copia/incolla, bookmark, invio di link tramite email
  - invece, il reload della pagina è permesso perché i browser ricordano l'intera ultima richiesta (ma viene chiesta conferma esplicita)

Programmazione per il Web

- Generazione di pagine dinamiche lato client o lato server
  - "dinamiche" nel senso che il loro contenuto varia a seconda delle informazioni ulteriori che vengono inviate con le richieste GET o POST, o a seconda dello "stato" (da introdurre con metodi vari)
- Lato client: si fa affidamento sul client
  - essenzialmente, il browser non si limita a visualizzare la pagina, ma deve anche eseguire dei pezzi di codice
- Lato server: si fa affidamento sul server
  - il server invia al client una pagina che è già il risultato di una computazione, e non semplicemente la copia di un file nel suo filesystem
  - nei casi più complessi, la pagina inviata contiene *anche* del codice lato client
  - caso tipico: gestione di interfacce uomo macchina

Programmazione per il Web: Lato Client

- Ha diversi pro:
  - più facili da realizzare, non serve nessun software aggiuntivo
  - per alcune cose, è una scelta obbligata
    - ad esempio, se si vuol fare in modo che, al passaggio del mouse, un certo pulsante venga abilitato
- ... e diversi contro
  - bisogna fare affidamento sul browser: e se è obsoleto?
  - bisogna fare affidamento sulla macchina che esegue il browser: e se è poco performante?
  - ci sono molti browser disponibili, difficile riuscire a supportarli tutti
  - per alcune cose, non si può usare
    - interrogazione a database protetti
    - codice modificabile per attacchi
- Tecnologie più usate: JavaScript, Java Applet

Programmazione per il Web: Lato Server

- Ha diversi pro:
  - ideale per interrogare database protetti
    - solitamente, accessibili dalla stessa macchina che funge da Web server
  - impossibile modificare il codice per attacchi
    - a meno di non hackerare l'intero Web Server
- ... e alcuni contro
  - occorre avere del software apposito da installare sul Web Server
  - non si può usare per "effetti" da interfacce uomo-macchina (vedere sopra)
- Tecnologie più usate: CGI e FastCGI (oramai in disuso), PHP, JavaScript Node.js, Ruby on Rails, ASP.NET (solo Microsoft) e...

Java Enterprise Edition: J2EE

- Suite di programmazione della Oracle che permette di costruire applicazioni professionali per le imprese
  - definisce una serie di API Java che possono essere usate per lo sviluppo di applicazioni Web lato server
- Esistono implementazioni di queste API che sono disponibili gratuitamente
  - Apache *Tomcat* (letteral.: gatto maschio)
    - sviluppato inizialmente nel progetto Jakarta, che però è stato sospeso nel 2011
    - TomCat è ora distribuito da Apache, che era il promotore di Jakarta
  - Sun ONE Application Server, Microsofts Internet Information Server (IIS), Apache HTTP Server + modulo JServ, BEAs WebLogic Application Server, IBMs WebSphere Application Server, World Wide Web Consortiums Jigsaw Web Server

Installazione di TomCat

Vedere `https://www.ntu.edu.sg/home/ehchua/programming/howto/Tomcat_HowTo.html`

Pagina iniziale, per vedere se tutto è ok:
`http://localhost:8080`

Programmazione per il Web con J2EE

- Sono previste due diverse modalità:
  - Java Servlet
    - veri e propri programmi Java, basati sulle Java API di J2EE
    - il loro output è solitamente una pagina HTML da inviare come risposta
    - molto simile, concettualmente, a CGI e Fast-CGI
  - JSP (Java Server Pages)
    - pagine HTML con "innesti" di codice Java
    - la parte HTML viene inviata come risposta così com'è, il codice Java viene sostituito dalla sua elaborazione
    - molto simile, concettualmente, a PHP e ASP

Java Servlet

- Hanno un ben preciso modello di esecuzione, interallacciato con il Web server
  - nel seguito, ci riferiremo al Web server come *servlet container* o *servlet engine*
  - quando arriveremo a parlare della pagina JSP, useremo anche il termine di *JSP container*, che è in sostanza un ulteriore wrapper per il servlet container
- Informazioni di base:
  - sul Web server sono definite un certo insieme di *Web application*
    - ognuna è un'entità a sé stante, e fornisce un certo servizio Web
  - ogni Web application è costituita da un insieme di servlet (e pagine JSP)
    - ognuna è un'entità a sé stante, ma può collaborare con le altre per fornire il servizio Web

Web Application Manager

- Ogni Web application dev'essere opportunamente *avviata* (*start*) dal *Web application manager*
  - `http://localhost:8080/manager/html`
- Di default, quando viene avviato il Servlet container, tutte le Web application vengono avviate
- Il manager permette di *chiudere* (*stop*) e/o *riavviare* (*reload*) una qualsiasi Web Application

Java Servlet: Ciclo di Vita

- Per ogni servlet all'interno di una Web application avviata:
  - la prima richiesta alla servlet ha come effetto quello di creare un processo del sistema operativo (*inizializzazione*)
  - ogni richiesta (compresa la prima, ma *dopo* aver concluso l'inizializzazione) viene gestita creando un *thread* all'interno del processo di cui sopra (*servizio*)
    - nelle CGI, ogni richiesta era un processo; questa gestione è nettamente più leggera ed efficiente
  - il processo relativo alla servlet viene terminato quando il Web application manager chiude la relativa Web application (*deallocamento*)
    - anche quando l'intero Servlet container viene chiuso
    - include anche il caso del riavvio (reload = stop + start)
    - pertanto, un riavvio causa la deallocazione del processo e la ri-esecuzione dell'inizializzazione

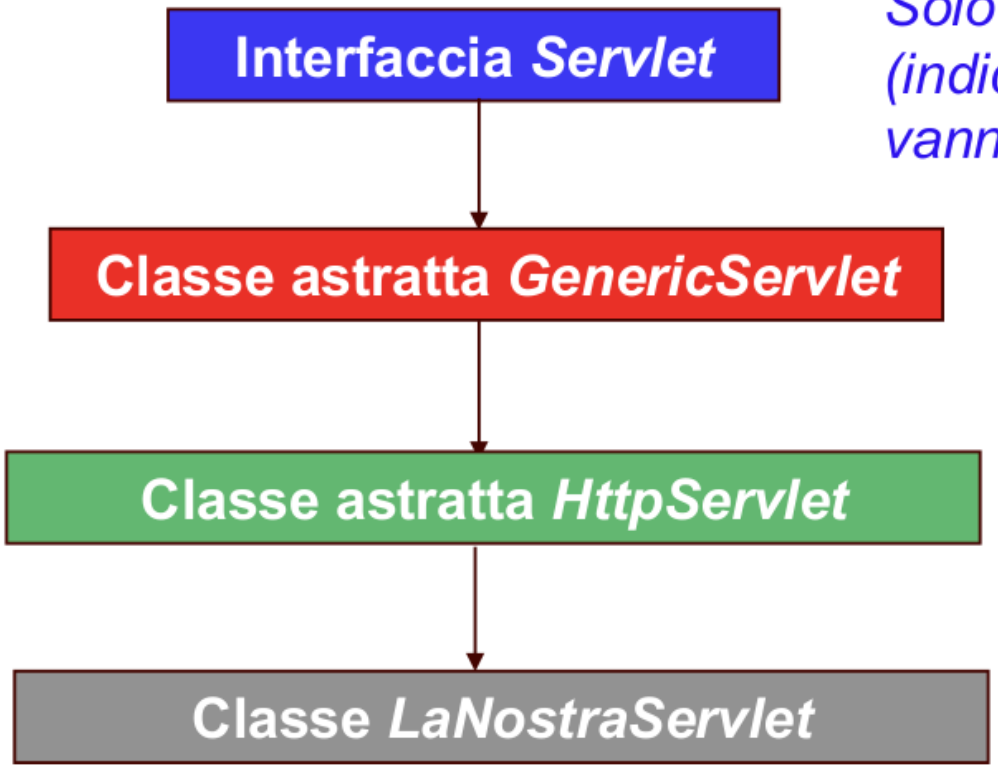
Java Servlet: Ciclo di Vita

- Per ogni servlet all'interno di una Web application avviata:
  - la prima richiesta alla servlet ha come effetto quello di creare un processo del sistema operativo (*inizializzazione*)
  - ogni richiesta (compresa la prima, ma *dopo* aver concluso l'inizializzazione) viene gestita creando un *thread* all'interno del processo di cui sopra (*servizio*)
    - nelle CGI, ogni richiesta era un processo; questa gestione è nettamente più leggera ed efficiente
  - il processo relativo alla servlet viene terminato quando il Web application manager chiude la relativa Web application (*deallocamento*)
    - anche quando l'intero Servlet container viene chiuso
    - include anche il caso del riavvio (reload = stop + start)
    - pertanto, un riavvio causa la deallocazione del processo e la ri-esecuzione dell'inizializzazione

Programmazione di una Java Servlet

- L'idea è che il programmatore di una Web Application non debba specificare l'intero ciclo di vita di ogni servlet
- Bensì, solo specificare delle istruzioni *aggiuntive* per ognuno dei passi del ciclo di vita
  - init - service - destroy
- Per rendere possibile tutto ciò, J2EE definisce delle classi e delle interfacce che definiscono le servlet
- Viene specificato che il servlet container chiamerà alcuni dei metodi di tali classi in determinati istanti del ciclo di vita
- Il programmatore può quindi limitarsi ad ereditare da una di queste classi e "riempire" il (corpo del) giusto metodo con il giusto codice
  - l'implementazione di default è tipicamente vuota o comunque banale, a seconda dei casi

Java Servlet: Java API



Solo interfaccia (indica quali metodi vanno implementati)

implements Servlet (indipendente dal protocollo in uso; non istanziabile)

extends GenericServlet (si usa solo con il protocollo HTTP; non istanziabile)

extends HttpServlet (istanziabile)

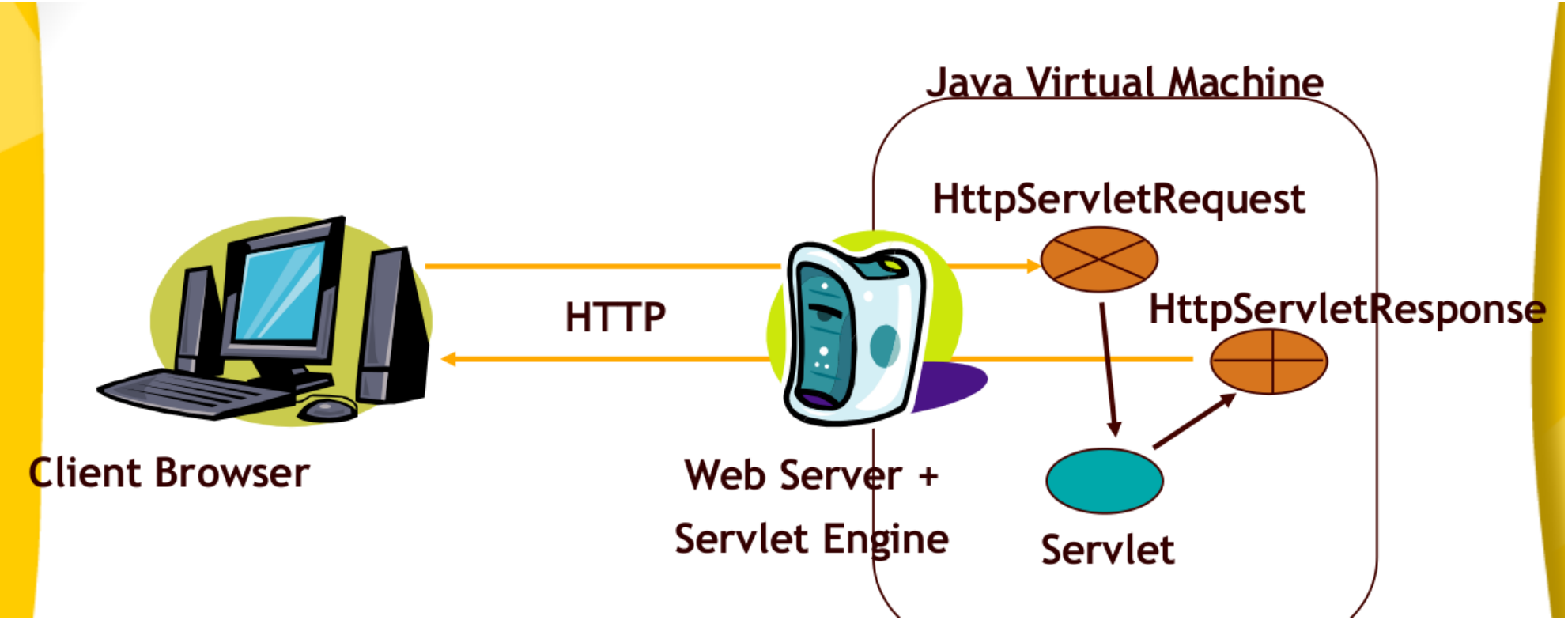
Java Servlet: Java API

- Tutte le interfacce e le classi di cui sopra si trovano nel package `javax.servlet`
  - `HttpServlet` si trova nel sottopackage `javax.servlet.http`
- Fino a `GenericServlet`, le servlet possono essere usate anche per altri protocolli di Internet diversi da HTTP
  - FTP, SMTP, ...
- `HttpServlet` invece è proprio per il protocollo HTTP
  - è l'unica sottoclasse di `GenericServlet` implementata in J2EE
  - nel seguito, useremo unicamente questa classe
  - i suoi metodi vengono opportunamente invocati dal servlet container come descritto sopra
  - ci si aspetta che il programmatore di servlet scriva il corpo di alcuni metodi di questa classe

La classe `GenericServlet`

- Per usare questa classe, il programmatore può ridefinire i corpi dei seguenti metodi:
  - `void init()`: chiamato al termine della fase di inizializzazione (prima di creare il thread per gestire la prima richiesta)
  - `void service(ServletRequest, ServletResponse)`: chiamato per gestire ogni singola richiesta
    - il codice viene eseguito come thread, quindi *in parallelo* ad altre richieste
    - il primo argomento è usato per l'input, e contiene le informazioni sulla richiesta
    - il secondo argomento è usato per l'output, e conterrà le informazioni sulla risposta
    - entrambi sono creati dal servlet container: in questo metodo sono già allocati
  - `void destroy()`: chiamato all'inizio della fase di deallocazione

Java Servlet: Java API



La classe GenericServlet

- https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/Servlet.html
- https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletConfig.html
- https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/GenericServlet.html
- https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletRequest.html
- https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletResponse.html

La classe `HttpServlet`

- Inizializzazione e deallocazione sono le stesse di `GenericServlet`
- Il metodo `service`, normalmente, non viene riscritto: chiama opportuni altri metodi
  - se il verbo usato per richiedere la pagina è X, il metodo chiamato è `do`X
  - possibili valori per X: `Get`, `Post`, `Put`, `Delete` (da ridefinire), `Head`, `Options`, `Trace` (ok l'implementazione di default)
  - questi sono i verbi della specifica HTTP 1.1
  - si possono implementare altri verbi, nel qual caso si possono aggiungere metodi e ridefinire opportunamente `doOptions`
  - notare che le interfacce usate per i 2 argomenti di queste funzioni sono `HttpServletRequest` e `HttpServletResponse`

Le Interfacce per Richiesta e Risposta

- https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletRequest.html
- https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletResponse.html
- https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServletRequest.html
- https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServletResponse.html

Il *Deployment Descriptor*

- Ogni servlet è definita da un file Java (possibilmente con la coadiuvazione di altri file Java)
- Il file Java non dice nulla su qual è l'URL completo per invocare una servlet
- Tale informazione è data dal *deployment descriptor*
- Si tratta di un file, che si deve trovare in una ben precisa locazione del file system

Struttura di una Web Application

- Ogni Web Application dev'essere una sottodirectory di una directory data
  - sotto Linux e per Tomcat 8, è `/var/lib/tomcat8/webapps`
  - il deployment descriptor si chiama sempre `web.xml`

```
melatti@melatti:/var/lib/tomcat7/webapps/teledidattica_lezione4$ tree
.
├── html_only
│   ├── servlet1.html
│   ├── servlet2.html
│   ├── servlet3.html
│   └── servlet4.html
├── index.html
├── META-INF
│   └── context.xml
└── WEB-INF
    ├── classes
    │   ├── Makefile
    │   ├── servlet1.class
    │   ├── servlet1.java
    │   ├── servlet2.class
    │   ├── servlet2.java
    │   ├── servlet3.class
    │   ├── servlet3.java
    │   ├── servlet4.class
    │   └── servlet4.java
    ├── lib
    └── web.xml
```

Sintassi di `web.xml`

- File XML che deve obbedire a delle regole precise per quanto riguarda i tag
- Tutto il documento è racchiuso dal tag `web-app`
- Per ogni risorsa (servlet, pagina JSP, pagina HTML) per la quale si voglia definire un URL, occorre definire una coppia di tag: `servlet` e `servlet-mapping`
  - la prima definisce il sorgente della pagina
    - classe Java per le servlet, sorgente HTML per le pagine HTML, sorgente JSP per le pagine JSP
  - la seconda definisce l'URL che dovrà essere usato per l'invocazione della risorsa
  - entrambi i tag sono figli di `web-app`

URL e `servlet-mapping`

- Deve cominciare con uno slash
  - niente nome del server
- Può contenere quanti slash si vuole: *directory virtuali*
  - non è richiesta alcuna corrispondenza con directory effettive del filesystem, né all'interno della Web application né altrove
- Può contenere wildcards, come `*`
  - unica wildcard per la specifica 3.0, ma le implementazioni potrebbero aggiungerne altre
  - vedere 12.2 della specifica per come si usa
- L'URL completo da usare per l'invocazione si ottiene aggiungendo il seguente prefisso all'URL definito nel `servlet-mapping`: `nome_server/nome_web_application`
- Per ogni tag `servlet`, si possono specificare più di un `servlet-mapping`

Il tag `servlet`

- Deve contenere i seguenti tag:
  - `servlet-name`, obbligatorio, per identificare univocamente la risorsa
  - `servlet-class`, quasi obbligatorio, per indicare il sorgente Java di una risorsa servlet
  - `jsp-file`, quasi obbligatorio, per indicare il sorgente JSP o HTML di una risorsa JSP o HTML
  - "quasi obbligatorio": esattamente uno tra `servlet-class` e `jsp-file` dev'essere specificato
  - `description`, facoltativo

Il tag `servlet-mapping`

- Deve contenere i seguenti tag:
  - `servlet-name`, obbligatorio, deve coincidere il `servlet-name` di un qualche tag `servlet`
  - `url-pattern`, obbligatorio almeno uno, per indicare l'URL della risorsa

URL Assoluti e Relativi

- Nei link all'interno delle pagine HTML (attributo `href` del tag `a`) si possono usare sia URL *assoluti* che *relativi*
  - attenzione: le cose saranno diverse quando parleremo delle redirezioni interne alle servlet
- URL assoluti: cominciano con il nome del server oppure con uno slash
  - se cominciano con uno slash, si sottintende che il server è lo stesso attuale
- URL relativi: non assoluti
  - non contengono mai il nome del server (è per forza quello attuale)

URL Assoluti e Relativi

- All'interno di uno stesso server, il significato è simile ai path nel file system di Linux:
  - per i path assoluti, la risorsa va cercata a partire dalla root di Tomcat (ad es.: `/var/lib/tomcat8/webapps`)
  - da lì, si considerano le directory virtuali dei `servlet-mapping` del `web.xml`, e vi si cerca la risorsa
  - se non la si trova, si considerano le directory "reali" del filesystem nella root di Tomcat
  - per i path relativi, vale la stessa cosa, ma il punto di partenza è la directory (reale o virtuale) nel quale si trova la risorsa attuale

Servlet e DBMS

- Uno dei compiti più diffusi per la programmazione Web lato server è quello di permettere l'accesso controllato a basi di dati
  - sottomissione e visualizzazione user-friendly di query complesse
  - sottomissione di query che modificano la base di dati
  - query transazionali

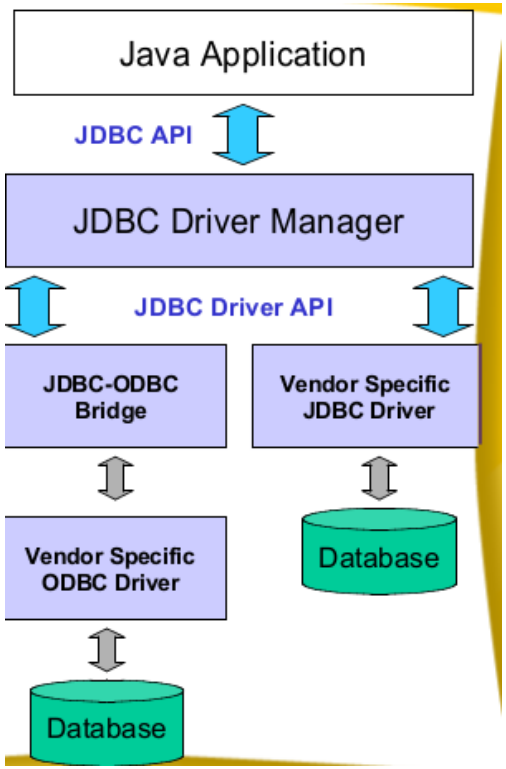
- Nel seguito, ci limitiamo a DBMS relazionali
  - PostgreSQL: `https://www.postgresql.org/`

JDBC: Java Database Connectivity

- Fa parte della Java SE (Standard Edition) della Oracle, e definisce delle API per l'accesso ai DBMS
  - qualsiasi DBMS che implementi ODBC è automaticamente supportato, in quanto JDBC può essere facilmente connesso con ODBC
    - la "O" sta per Open
  - praticamente tutti i più noti DBMS sono supportati: PostgreSQL, MySQL, DB2, SQLServer...
  - quindi, cambiare DBMS non implica cambiare il codice Java che ci si interfaccia
    - ovviamente, a parità di schema
- Le API definiscono delle interfacce e delle classi per connettersi ed interrogare un DBMS, e come "navigare" il risultato di una query

JDBC: Java Database Connectivity

- connessione JDBC-ODBC
- direttamente in Java



JDBC: I 6 Passi Fondamentali

1. Caricamento del driver JDBC
   - in realtà, il primo passo è scaricare ed installare un driver JDBC per il DBMS scelto
   - questo però va fatto una volta sola, mentre gli altri passi qui riportati vanno effettuati ogni volta che si vuole interagire con il DBMS
2. Connessione al DBMS
3. Creazione di un oggetto Java rappresentante la query da eseguire
4. Esecuzione della query
5. Navigazione del risultato della query (eventuale)
   - questi tre passi possono essere ripetuti più volte
6. Chiusura della connessione

Passo Zero: Scaricare ed Installare il Driver JDBC

- Per un dato DBMS, ce ne potrebbe essere più d'uno
- Per PostgreSQL: `https://jdbc.postgresql.org/`
- Per MySQL: `http://dev.mysql.com/downloads/connector/j/`
- Per SQLServer: `https://msdn.microsoft.com/it-it/library/mt484311(v=sql.110).aspx`
- Occorre metterlo nella cartella `WEB-INF/lib` della web application che lo deve usare
  - ovviamente, occorrerà settare opportunamente il `CLASSPATH` del compilatore

Primo Passo: Caricare il Driver JDBC

- Dipende dal driver, ma tutti usano il meccanismo della Java reflection
- Tale meccanismo prevede anche la "registrazione" del driver
  - ovvero, verrà poi usato quel driver nelle invocazioni delle API
  - per farlo, si userà la classe `DriverManager`

```
try {
  // per postgresql
  Class.forName("org.postgresql.Driver");
} catch (ClassNotFoundException cnfe) {
  System.out.println("Error loading driver: ",
                     cnfe);
}
```

Secondo Passo: Connessione al Driver JDBC

- Si chiama `DriverManager.getConnection`
  - metodo statico, come tutti quelli di `DriverManager`
- Il primo argomento contiene la stringa di connessione
  - separati da 2 punti: `jdbc`, il DBMS, `//`, l'host del DBMS, la sua porta e il nome di un database
  - gli ultimi due vanno separati da `/`
- Altri due argomenti sono username e password per lo specifico database

```
try {
  Connection conn = DriverManager.getConnection(
     "jdbc:postgresql://localhost:5432/example",
     "username", "password");
} catch (SQLException sqle) { ... }
```

Terzo Passo: Preparazione della Query

- È importante qui distinguere due tipi di query
  - query "semplici", che non fanno uso di costanti: interfaccia `Statement`
  - query "preparate", che fanno uso di costanti: interfaccia `PreparedStatement`
  - la seconda estende la prima
  - è possibile usarle in modo interscambiabile, ma è sconsigliabile usare `Statement` per query con costanti

```
Statement st = conn.createStatement();
st.setMaxRows(1000);
st.setQueryTimeout(60); //max 1 minuto
```

Terzo Passo: `PreparedStatement`

- Permette di ignorare i problemi di specifica delle costanti
  - ad esempio, stringhe tra apici, numeri no
- Per le stringhe, evita anche problemi di SQL-injection
- Ottimo anche per ripetere sempre la stessa query con diverse costanti

```
PreparedStatement st = conn.prepareStatement(
  "SELECT * FROM TABELLA WHERE ATTR_INT >= ? AND
   ATTR_STR LIKE ? AND ATTR_FLOAT <= ?");
st.setDouble(3, 4.0);
st.setString(2, "*qualcosa*");
st.setInt(1, 1);
```

Quarto Passo: Esecuzione della Query

- Metodi `executeQuery` e `executeUpdate`
  - prendono in input la query nell'interfaccia `Statement`, mentre sono senza argomenti nell'interfaccia `PreparedStatement`
- Per la precisione: la prima quando c'è un risultato, la seconda quando non c'è
  - le `SELECT` tornano sempre un risultato (magari vuoto)
  - le `INSERT` e le `UPDATE` ritornano un risultato *solo* quando c'è la clausola `RETURNING`
  - usare `executeQuery` in quest'ultimo caso solleva un'eccezione
    - ma la modifica viene eseguita
  - usare `executeUpdate` negli altri casi non permette di vedere il risultato
    - ritorna solamente un intero...

Quarto Passo: Query Transazionali

- Metodo `setAutoCommit` di `Connection`
  - prende in input un booleano
- Passandogli `True`, ogni query che debba modificare il database avrà effetto immediato
- Passandogli `False`, ogni query che debba modificare il database non avrà effetto permanente dopo la `executeUpdate`
  - per ottenere l'effetto permanente occorrerà chiamare `Connection.commit`
  - ovviamente, c'è anche la possibilità di chiamare `Connection.rollback`
  - l'effetto sarà quello di rendere permanenti o cancellare tutte le query eseguite dopo l'ultima commit o rollback
    - o dall'inizio, se questa è la prima `commit` o `rollback`

Quinto Passo: Navigazione del Risultato

- Il metodo `executeQuery` ritorna un oggetto dell'interfaccia `ResultSet`
  - usato anche dai metodi dell'interfaccia `DatabaseMetaData`
  - ottenibile da `Connection.getMetaData`
- Ha il concetto di "riga corrente" nell'output ritornato dalla query
  - opportuni metodi, come `first`, `last`, `next` o `previous`, possono cambiare la riga corrente
  - inizialmente, la riga corrente non è definita; occorre chiamare uno dei suddetti metodi per definirla
  - è come se fosse posizionata prima della prima riga, quindi chiamando subito `next` si va alla prima
  - quindi, fare un `while (result.next())` permette di scandire facilmente tutte le righe del risultato
  - opportuni metodi, come lo stesso `next` o `isFirst` o `isLast`, permettono di capire se è o no la prima o l'ultima

Quinto Passo: Navigazione del Risultato

- Data una riga corrente, si può ottenere il valore di una colonna
- Si usano una serie di metodi a seconda del tipo della colonna di cui prendere il valore
  - `getInt`, `getDouble`, `getString`...
  - ma c'è anche il "magico" `getObject`
- Tutti con 1 argomento, che può essere
  - la stringa usata nella `SELECT`
  - l'ordine usato nella `SELECT` (a partire da 1)
- Con il metodo `ResultSet.getMetaData` si possono ottenere informazioni sulla query appena eseguita

Sesto Passo: Chiusura della Connessione

- Semplicemente, `Connection.close()`
- Dopodiché, non sarà più possibile creare altri `Statement` o `PreparedStatement` sull'oggetto di tipo `Connection`
  - verrà sollevata un'eccezione
- Non si può più riaprire: occorre chiamare un'altra `DriverManager.getConnection`
- Data la complessità di apertura e chiusura della connessione, è meglio cercare di condensare tutte le query da fare tra un'unica coppia di apertura e chiusura connessione
  - piuttosto che aprirla prima e chiuderla dopo ogni operazione/query

Interazione con DBMS nelle Servlet

- Ci sono 4 opzioni (`https://www.safaribooksonline.com/library/view/java-programming-with/059600088X/ch04s02.html`):
  1. Una connessione per ogni transazione
  2. Una connessione per ogni servlet
     - solo se il driver garantisce che `Connection` è thread safe
     - ok per Oracle, non per PostgreSQL
     - vedere `http://stackoverflow.com/questions/1531073/is-java-sql-connection-thread-safe`
  3. Una connessione per ogni utente (client)
     - servono le sessioni, per ora posposto
  4. Gestione di un *pool* globale di connessioni

Una Connessione per ogni Transazione

- Tutti i 6 passi dentro la `doXxx`
  - o meglio, il primo (registrazione del driver) nella `init`
  - gli altri 5 nella `doXxx`
- L'oggetto di tipo `Connection` sarà una variabile locale alla `doXxx`
- Ogni utente ha la sua connessione
  - possibile esecuzione in parallelo tra diversi utenti
- Efficiente per l'esecuzione in parallelo, meno per dover stabilire ogni volta una connessione

Una Connessione per ogni Servlet

- Occorre un oggetto di tipo `Connection` come membro (privato) della classe
- L'apertura della connessione sarà nella `init`, la chiusura, se c'è, è nella `destroy`
  - chiusura non necessaria, in quanto viene comunque eseguita dal servlet container al momento della deallocazione della web application
- Tutti gli utenti della servlet useranno la stessa connessione
  - i metodi di `Connection` dovrebbero essere `synchronized`, quindi non ci sarà esecuzione in parallelo tra diversi utenti
  - se non lo sono, corse critiche sono possibili

Pool di Connessioni

- Permette di fare *load balancing*: non più di un certo numero di connessioni sono attive
- Chiaramente servono delle classi ausiliarie
  - gestiscono l'inizializzazione di driver e connessioni, nonché le richieste provenienti dalle servlet
- Una servlet che debba interagire con un DBMS userà una di queste classi per farsi ritornare una connessione attiva
- Ovviamente, ci vuole un pool separato per ogni specifico database del DBMS
  - anche username e password dovranno coincidere
- Possibile usare la stessa connessione tra diverse servlet

Lo *Stato* nella Programmazione Web

- Il protocollo HTTP è senza stato (*stateless*) per definizione
  - la risposta ad una richiesta non dipende da richieste fatte in precedenza
- Tuttavia, in moltissime applicazioni è essenziale introdurre uno stato
  - per visualizzare informazioni private mantenute su un server Web, è necessario fare prima un login
  - quindi, le informazioni da visualizzare dipendono dall'aver visitato in precedenza una pagina di login, e dall'avervi inserito gli input richiesti in modo corretto
- Vediamo nel seguito come si può introdurre (subdolamente) lo stato nell'HTTP

Gli *Hidden Inputs*

- Un primo modo, molto dozzinale ma in alcuni casi utile, è usare direttamente l'HTML
- All'interno di un form si possono inserire degli input testuali che non vengono visualizzati dal browser
  - è sufficiente usare `hidden` come valore dell'attributo `type` del tag `input`
- Questi input possono mantenere informazioni sullo stato
- Semplici da usare, ma non si possono usare per mantenere informazioni importanti (ad es., password)
  - basta aprire il sorgente della pagina per vedere tutto
- Inoltre, costringono ad avere sempre dei form sui quali cliccare

Variabili Locali ai Metodi

- Visto che le Servlet sono classi Java, perché non memorizzare le informazioni di stato su opportune variabili?
- Variabili locali alla `DoXXX`

```
void doGet(HttpServletRequest req,
           HttpServletResponse resp) {
  boolean isLoggedIn;
  if (isLoginReq(req) && loginOk(req))
    isLoggedIn = true;
  if (isPrivatePage(req) && isLoggedIn) ...
```

- Ad ogni richiesta HTTP vengono resettate, quindi non possono essere usate per mantenere informazioni persistenti

Attributi delle Classi Java

- Possono mantenere informazioni persistenti, quindi potrebbero andar bene
  - ricordiamo che l'istanza della classe permane dalla prima invocazione fino all'esplicita deallocazione effettuata dal Web manager
  - proprio quello che serve per mantenere informazioni di stato

```
public class Servlet {
  boolean isLoggedIn = false;
  void doGet(HttpServletRequest req,
             HttpServletResponse resp) {
    if (isLoginReq(req) && loginOk(req))
      isLoggedIn = true;
    if (isPrivatePage(req) && isLoggedIn) ...
```

Attributi di Classi Java

- Tuttavia, ogni invocazione della servlet è un *thread* dell'istanza della classe
  - quindi, dei semplici attributi non vanno bene, in quanto andrebbero *condivisi* tra le varie richieste
  - invece, ogni richiesta dovrebbe avere dei valori propri
  - ad esempio, se due utenti diversi effettuano un login (ciascuno con le proprie credenziali), è necessario poter mantenere tali informazioni *separatamente* per ciascun utente

- Si potrebbe quindi pensare ad un'hash table: la chiave è un identificativo dell'utente, il valore può essere un qualsiasi oggetto Java
  - ovviamente l'hash table sarebbe un membro della classe opportunamente inizializzato con la `init`

Attributi delle Classi Java

```
public class Servlet {
  HashMap<String, Boolean> logins =
    new HashMap<String, Boolean>; // should be in init
  void doGet(HttpServletRequest req,
             HttpServletResponse resp) {
    if (isLoginReq(req) && loginOk(req))
      logins.put(getId(req), true);
    if (isPrivatePage(req) && logins.get(getId(req))) ...
```

Attributi di Classi Java

- Il problema è proprio l'identificatore per ogni utente
  - si possono usare solo le informazioni presenti nella richiesta HTTP
  - quindi, solo l'indirizzo IP di provenienza
- Ai fini della programmazione Web, l'indirizzo IP di provenienza *non* è sufficiente ad identificare in modo univoco un utente
  - due utenti diversi potrebbero usare direttamente lo stesso computer
    - o con 2 diverse sessioni grafiche
    - o con 2 diverse sessioni testuali, e si usano client come `curl`
  - due utenti diversi potrebbero usare lo stesso *proxy*
    - è sufficiente impostare dei parametri del browser

Lo Stato sui Client

- L'unica cosa veramente univoca è lo stesso client
  - certo, ci potrebbero essere due utenti diversi che si alternano sullo stesso client, ma a quel punto sarebbero consci l'uno dell'altro...
- Quindi, per mantenere informazioni di stato, si memorizzano (e poi rileggono) informazioni sul client
  - il client scriverà dei file su disco, per poi rileggerli (ed eventualmente modificarli) quando necessario
- Tali informazioni sono del tipo chiave=valore
  - ciascun assegnamento di questo tipo è chiamato *cookie*

I Cookies

- Il loro funzionamento è specificato nell'RFC6265
  - essenzialmente, i cookies vanno inviati dal server insieme con la risposta HTTP
  - il client li deve salvare su disco, e poi reinviarli al server ad ogni richiesta HTTP successiva
  - tecnicamente, i cookies sono contenuti negli header della richiesta e della risposta
  - nel caso della risposta HTTP, il campo si chiamata *Set-Cookie* (va visto come una richiesta di salvataggio del cookie...)
  - nel caso della richiesta HTTP, il campo si chiama *Cookie*
  - hanno un tempo massimo di vita, trascorso il quale "scadono", e non vengono più inviati dal client

La Classe Cookie

```
void doGet(HttpServletRequest req,
           HttpServletResponse resp) {
  if (isLoginReq(req) && loginOk(req))
    resp.addCookie(new Cookie("login", "true"));
  if (isPrivatePage(req)) {
    Cookie c[] = req.getCookies();
    for (int i = 0; i < c.length; i++) {
      if (c[i].getName().equals("login") &&
          c[i].getValue().equals("true"))
        ...
    }
  }
```

Insiemi di Cookie

- Un cookie memorizza una sola accoppiata chiave=valore
  - sia la chiave che il valore devono essere stringhe
- In molte applicazioni è necessario ricordare svariate informazioni per ogni utente
  - *sessioni*
- Necessario organizzare meglio le informazioni: hash table

Cookie e Hash Table

- Si tratta di un'hash table i cui valori sono a loro volta hash tables
- La chiave è un identificativo per ogni utente
  - quindi il tipo sarebbe qualcosa come `HashMap<String, HashMap<String, String>>`
- La seconda hash table, ovvero quella che fa da valore per la prima, serve per mantenere le svariate informazioni necessarie per ogni utente
  - può essere visto come un insieme di cookie
  - ma essendo direttamente organizzato in hash table, è molto più facile trovare il valore di una data chiave

Creazione di una Sessione

```
String sessID = makeUniqueString();
Hashtable sessionInfo = new Hashtable();
Hashtable globalTable = findTableStoringSessions();
globalTable.put(sessID, sessionInfo);
Cookie sessionCookie = new Cookie("JSESSIONID", sessID);
sessionCookie.setPath("/web_application/");
response.addCookie(sessionCookie);
// add pairs to sessionInfo
```

Uso di una Sessione

```
Cookie c[] = request.getCookies();
String sessID;
for (int i = 0; i < c.length; i++)
  if (c[i].getName().equals("JSESSIONID")) {
    sessID = c[i].getValue();
    break;
  }
Hashtable globalTable = findTableStoringSessions();
Hashtable sessionInfo = globalTable.get(sessID);
// use pairs in sessionInfo
```

L'interfaccia `HttpSession`

- Semplifica i metodi da chiamare per effettuare le sequenze di operazioni mostrate sopra
  - soprattutto la fase di ricerca di un cookie
- Le operazione restano quelle, quindi per usare una sessione verrà memorizzato un singolo cookie sul client
  - quello che contiene l'ID dell'utente che usa quel client (chiave per l'hash table più esterna)
- L'oggetto di tipo `HttpSession` relativo all'attuale richiesta può essere ottenuto tramite il metodo `getSession` di `HttpServletRequest`
  - è possibile specificare se la sessione debba essere già stata creata, e non crearla in caso contrario: basta passare `false` a `getSession`
  - altrimenti, la sessione verrà creata se già non era presente: basta o non passare nulla o passare `true`

L'interfaccia `HttpSession`

- L'uso della risposta è un po' nascosto
  - non si può usare `getSession` dopo il commit della risposta
  - ovvero, dopo `out.flush` o `out.close`
- Più potente dei cookie
  - i cookie permettono solo associazioni stringa-stringa
  - `HttpSession.setAttribute` permette associazioni stringa-oggetto
- Funziona solo all'interno della stessa web application
  - i cookie possono funzionare per tutte le web application del servlet container

Creazione di una Sessione

```
String sessID = makeUniqueString();
Hashtable sessionInfo = new Hashtable();
Hashtable globalTable = findTableStoringSessions();
globalTable.put(sessID, sessionInfo);
Cookie sessionCookie = new Cookie("JSESSIONID", sessID);
sessionCookie.setPath("/");
response.addCookie(sessionCookie);
// add pairs to sessionInfo
```

Creazione di una Sessione

```
HttpSession session = request.getSession();
// add pairs to session using setAttribute
// remove pairs to session using removeAttribute
```

Uso di una Sessione

```
Cookie c[] = request.getCookies();
String sessID;
for (int i = 0; i < c.length; i++)
  if (c[i].getName().equals("JSESSIONID")) {
    sessID = c[i].getValue();
    break;
  }
Hashtable globalTable = findTableStoringSessions();
Hashtable sessionInfo = globalTable.get(sessID);
// use pairs in sessionInfo
```

Uso di una Sessione

```
HttpSession session = request.getSession();
// get pairs from session using getAttribute
```

Disabilitazione dei Cookie

- Tutti i client HTTP prevedono la possibilità di disabilitare i cookie
  - non vengono né salvati né (ri)mandati con la richiesta HTTP
  - questo per evitare problemi di privacy
- Conseguenza: web application basate su cookie e/o sessioni non funzionerebbero
  - ad es.: utenti loggati apparirebbero come non loggati
- Per i cookie non si può far nulla, ma per le sessioni c'è una via d'uscita: l'*URL rewriting*

URL Rewriting

- Si sfrutta una stringa "nascosta" per memorizzare il session identifier (`JSESSIONID`)
  - è l'unica cosa che servirebbe memorizzata sul client
  - o sono abilitati i cookies, e lo si memorizza
  - o non sono abilitati, e lo si nasconde nell'URL
- Per nascondere una stringa nell'URL, si usa un carattere riservato: il `;`
  - definito come "riservato" dall'RFC 1738
- L'URL rewriting consiste appunto nell'aggiungere il `JSESSIONID` all'URL, usando `;` come separatore
  - senza query-string, viene così: `URL;JSESSIONID=valore`
  - con query-string, viene così: `URL;JSESSIONID=valore?query-string`
  - ovviamente, il server ha accesso all'URL completo

URL Rewriting

- Metodo `HttpServletResponse.encodeURL(String url)`
  - ritorna `url` con l'aggiunta dell'id della sessione corrente come descritto in precedenza
- L'idea è quella di fare in modo che tutti i link della pagina vengano trasformati in questo modo
  - se l'utente ci clicca sopra, la sessione è preservata nonostante la disabilitazione dei cookie
- Ovviamente, se l'utente scrive lui stesso l'URL nella barra degli indirizzi (o usa il tasto back), la sessione viene persa
  - ma di solito si seguono i link...
  - analogamente, niente bookmarks

URL Rewriting

- Metodi `HttpServletRequest.isRequestedSessionIdFromCookie()` e `HttpServletRequest.isRequestedSessionIdFromURL()`
  - sono i due modi con cui si implementano le sessioni
- `encodeURL`, funzionamento dietro le quinte:
  - la prima volta che viene chiamata, si manda sia il cookie di sessione che l'URL riscritto
  - se alla seguente chiamata non viene trovato il cookie nella risposta, allora si prosegue con l'URL rewriting (cookie disabilitati)
  - altrimenti, si prosegue con il cookie (cookie abilitati)
  - così si ottimizza sulla lunghezza dei link generati

Cancellazione da Server di Cookie e Sessioni

- Per cancellare un cookie non ci sono metodi diretti
  - occorre trovare il cookie da cancellare, settare il suo tempo a 0 e poi rimandarlo al client
- Anche la sessione ha un suo tempo massimo
  - c'è però anche un metodo esplicito: `HttpSession.invalidate`

Interazione con DBMS: una Connessione per ogni Utente

- Occorre mettere un oggetto di tipo `Connection` dentro la sessione
- Quando la sessione viene creata, la connessione verrà stabilita
  - potrebbe tornare utile il metodo `HttpSession.isNew`
- Per fare query al DBMS, si prende la connessione con `HttpSession.getAttribute`
- Quando la sessione scade, o l'utente fa logout, la connessione viene chiusa
- In questo modo, ogni utente ha la sua connessione
  - parallelismo solo tra utenti diversi, ma ok
  - meno operazioni di instaurazione della connessione
  - possibile usare la stessa connessione tra diverse servlet

Configurazione di Servlet

- Si vogliono settare dei parametri di configurazione, senza dover aprire e modificare il codice delle servlet
  - ad es., la locazione di un file di input, o il nome del database da usare per la web application
- Si usa sempre il deployment descriptor `web.xml`
- Sono sempre coppie chiave-valore
  - si scrivono sul `web.xml`
  - si leggono dalle servlet con opportuni metodi
  - per cambiarle, è sufficiente modificare `web.xml` e poi fare restart della web application
    - senza dover modificare e ricompilare il codice delle servlet

Configurazione di Servlet

- Due tipi di configurazione: locali e contestuali
  - quella locale vale per una sola servlet (o anche pagina JSP)
  - quella contestuale vale per tutte le servlet (e le pagine JSP) di una stessa web application
- A livello del `web.xml`, le coppie chiave-valore vanno specificati con i tag `param-name` e `param-value`
  - configurazione locale: vanno racchiusi in un tag `init-param` all'interno del tag `servlet`
  - configurazione contestuale: vanno racchiusi in un tag `context-param` all'interno del tag `web-app`
    - così da non essere specifici di una servlet

Configurazione di Servlet

- Dal punto di vista delle servlet, per leggere le configurazioni si usano due diverse interfacce:
  - `ServletConfig` per le configurazioni locali
  - `ServletContext` per le configurazioni contestuali
- In entrambi i casi, ci sono dei metodi `getServletConfig` e `getServletContext`, disponibili direttamente sull'interfaccia `HttpServlet`
- In entrambi i casi, si usa il metodo `getInitParameter(String name)` per ottenere il valore di una specifica chiave
  - c'è anche il metodo `getInitParameters()` per scorrere tutti i parametri di configurazione

Redirezione tra Servlet

- È spesso necessario che una servlet ne possa "chiamare" un'altra
- Due tipi di "chiamata": rientrante (inclusione) e non rientrante (redirezione)
- Importante il passaggio di parametri
- Importante come specificare la destinazione
- Due modalità distinte
  - la prima permette solo la redirezione, e vale con qualsiasi URL
  - la seconda permette sia redirezione che inclusione, e vale solo all'interno della stessa web application

Solo Redirezione: `sendRedirect`

- Metodo di `HttpServletResponse`
- Un solo argomento: l'URL di destinazione
  - può essere un URL completo di server (che può essere esterno) ed eventualmente anche di protocollo
  - o può essere un URL relativo al server (deve iniziare con uno slash)
  - o può essere un URL relativo (non deve iniziare con uno slash)
- Non è possibile scrivere null'altro nella risposta HTTP, né prima né dopo una `sendRedirect`
- Passaggio di parametri: solo tramite query-string esplicita
  - solo accoppiate chiave-valore di tipo stringa

Redirezione ed Inclusione: `RequestDispatcher`

- Interfaccia `RequestDispatcher`, ottenibile con il metodo `getRequestDispatcher`
  - è un metodo sia di `HttpServletRequest` che di `ServletContext`, ma non è la stessa cosa
  - in entrambi i casi, c'è un solo argomento, l'URL di destinazione
    - può essere un URL *relativo alla web application* (deve iniziare con uno slash, ma la web application è sottintesa e *non va* specificata)
    - o può essere un URL relativo (*solo se* si è usata la `getRequestDispatcher` di `HttpServletRequest`)
- Metodi principali: `forward` (redirezione) e `include` (inclusione)
  - due argomenti, di tipo `HttpServletRequest` e `HttpServletResponse`, esattamente come la `doXXX`

Redirezione: Differenze

- Prima differenza: `sendRedirect` si può usare qualsiasi sia l'URL, `forward` solo all'interno della stessa web application

- Assumendo che l'URL di destinazione sia all'interno della stessa web application, cosa cambia tra l'usare `sendRedirect` e `forward`?
  1. l'URL va precisato in modo diverso
  2. la dinamica è differente
  3. il passaggio dei parametri è differente

Differenze nell'URL

- L'URL va passato direttamente come parametro alla `sendRedirect`, mentre nel caso del `forward` va passato alla `getRequestDispatcher`
- Se inizia con uno slash:
  - deve avere il context root nel caso del `sendRedirect` (URL relativo al server)
  - non deve avere il context root nel caso del `getRequestDispatcher` (URL relativo alla web application)
- URL completo possibile solo nella `sendRedirect`

Differenze nella Dinamica

- La chiamata alla `sendRedirect` ha come effetto i seguenti passi:
  1. viene mandata al client una risposta che contiene solamente l'header con l'URL cui effettuare il reindirizzamento
     - se l'URL passato alla `sendRedirect` non era completo, viene opportunamente completato prima di inserirlo nell'header
  2. il client invia una nuova richiesta HTTP con l'URL contenuto nell'header appena ricevuto
- La chiamata alla `forward` viene risolta all'interno del Web server, senza interventi del client
  - con la `sendRedirect`, la barra degli indirizzi del browser mostra il nuovo URL
  - con la `forward`, no
- È evidente che la `forward` è più efficiente

Differenze nel Passaggio dei Parametri

- Con la `sendRedirect` si possono usare:
  - la query-string (esplicita)
    - unico metodo se il server è diverso
  - i cookie
    - però sono persistenti, mentre i parametri non dovrebbero esserlo
    - stesso server, ma ok anche tra diverse web application
  - la sessione
    - però è persistente, mentre i parametri non dovrebbero esserlo
    - stesso server, stessa web application, diverse servlet
- Con la `forward`, si può usare anche il metodo `HttpServletRequest.setAttribute`
  - da abbinare ad un `HttpServletRequest.getAttribute`...
  - associazioni chiave-oggetto

Precisazioni e Limitazioni sulla SendRedirect

- L'istruzione `response.sendRedirect(url)` è equivalente alle seguenti 2 istruzioni:

```
response.setStatus(response.SC_FOUND);
response.setHeader("Location", url);
```

- In ciò, è molto simile alla redirezione tramite HTML:

```
<head>
<meta http-equiv="refresh" content="0; url=url"/>
</head>
```

- La costante `response.SC_FOUND` vale 302
  - è lo status 302 della risposta nel protocollo HTTP
  - lo stesso valore assume `response.MOVED_TEMPORARILY`, che però c'è solo per compatibilità all'indietro
  - volendo c'è anche `response.MOVED_PERMANENTLY` (status 301), ma la differenza è solo per i page rank

Status 303 vs Status 307

- Lo status 302 non chiariva bene alcuni aspetti della redirezione
  - se la richiesta originale era di GET, non ci sono problemi
  - ma se è un POST, un PUT o un DELETE, che possono (in principio) causare "modifiche", va permesso di farlo anche sul link di destinazione?
- Allora due nuovi status (HTTP/1.1): 303 e 307
  - 303: non importa il metodo della richiesta, la redirezione va fatta con GET
  - 307: il metodo della richiesta resta invariato
- Normalmente, per quasi tutti gli "agent" (client del protocollo HTTP), 302 = 303

Redirezione di Richieste di POST

- Con la `sendRedirect` non va bene, perché usa lo status 302 = 303
  - conseguenza: eventuali parametri vengono persi!
  - verrebbero persi anche con richieste di GET
  - occorre esplicitamente creare una nuova querystring che contenga i parametri della richiesta
- Fare `response.setStatus` prima di `response.sendRedirect` non aiuta
  - perché quest'ultima resetta nuovamente lo status a 302
- Quindi si risolve così:

```
response.setStatus(response.SC_TEMPORARY_REDIRECT);
response.setHeader("Location", url);
```

Indirizzi Relativi dopo la Redirezione

- Se la redirezione viene fatta con la `sendRedirect`, eventuali indirizzi relativi nella pagina di destinazione si comporteranno "bene"
  - l'esempio tipico è quello di immagini: `<img src="foo.jpg">`
  - l'immagine `foo.jpg` verrà cercata nella stessa directory della pagina di destinazione
    - directory reale o virtuale che sia
- Con il `RequestDispatcher` le cose si complicano: gli indirizzi relativi vengono intesi rispetto alla servlet *sorgente*
  - per evitare problemi, è consigliabile l'uso dei path relativi al server
  - ovvero, quelli che cominciano con uno slash, seguito dal context root

Autenticazione Programmata vs. Autenticazione Dichiarativa

- Le Web Application 3 e 4 presentavano esempi di *autenticazione programmata*
  - fatta per davvero, avrebbe dovuto interrogare un database
  - il programmatore di servlet deve scrivere esplicitamente il codice Java che gestisce le richieste di autenticazione
- Il framework J2EE mette a disposizione un'alternativa: l'*autenticazione dichiarativa*
  - sfruttando delle specifiche del protocollo HTTP
  - il programmatore di servlet si limita a dichiarare che tipo di autenticazione vuole
    - tra le 3 a disposizione
  - deve inoltre precisare chi ha accesso a quali risorse, seguendo una sintassi predefinita
  - niente codice Java, solo un certo numero di files XML
  - sta al servlet container fornire i metodi per realizzare il tutto

Roles, *Users* e *Principals*

- L'autenticazione dichiarativa è basata su ruoli, utenti e principali
- Utenti e principali sono sinonimi
  - si usa *user* quando si dichiarano i nomi nei file XML
  - si usa sia *user* che *principal* quando da una servlet si vuole conoscere il nome dell'utente loggato
  - questo perché in generale Java usa il termine *principal* come oggetto principale della sicurezza (`java.security.Principal`)
- Ciascun utente può ricoprire uno o più ruoli
- Le restrizioni di accesso sono definite sui ruoli e non sugli utenti

Definizioni di Ruoli ed Utenti

- Un esempio già visto: come accedere alla pagina di gestione delle web applications
  - lì ci sono dei ruoli già definiti durante l'installazione, come ad esempio `admin-gui`
  - in effetti, la pagina di gestione delle web application è gestita con l'autenticazione dichiarativa
- Pertanto gli utenti e i ruoli sono definiti nel file di configurazione `tomcat-users.xml`
- Tag `role` per i ruoli, tag `user` per gli utenti/principali
  - direttamente dentro il tag `tomcat-users`
  - attributi `rolename` e `username` per definire il nome
  - il tag `user` ha anche gli attributi `password` (in chiaro...) e `roles`
  - se l'utente ha più ruoli, vanno separati da virgole

Il Deployment Descriptor

- Utenti e ruoli sono quindi definiti per l'intero servlet container
  - se occorre modificarli, occorrerà poi fare il restart dell'intero servlet container
- Tutto il resto dell'autenticazione dichiarativa, invece, va definito separatamente per ogni web application
  - per farlo, si usa il deployment descriptor (`WEB-INF/web.xml`)
- Permette di dichiarare:
  1. il tipo di autenticazione (valido per l'intera web application)
  2. a quali risorse applicare l'autenticazione
     - servlet, pagine JSP, e perfino pagine HTML o files generici

Dichiarazione delle Risorse da Proteggere

- Si usa il seguente tag composto: `security-constraint+`
  - direttamente dentro `web-app`
  - al suo interno vanno annidati i tag composti `web-resource-collection+` e `auth-constraint?`
  - il primo dice quali sono le risorse da proteggere
  - il secondo dice quali ruoli possono accedere a tali risorse

- In `auth-constraint` va annidato il tag semplice `role-name+`
  - dev'essere uno dei ruoli definiti nel file di configurazione `tomcat-users.xml`

Dichiarazione delle Risorse da Proteggere

- In `web-resource-collection` vanno annidati i tag semplici `web-resource-name1` e `url-pattern+`
  - `web-resource-name` funge da semplice descrizione di questo insieme di risorse
  - `url-pattern` indica i path delle risorse
    - per le servlet dovrà coincidere con l'`url-pattern` del `servlet-mapping`
    - anche pagine JSP, pagine HTML e files generici, da indicare con il path relativo al contesto

Dichiarazione del Tipo di Autenticazione

- Si usa il seguente tag composto: `login-config`?
  - direttamente dentro `web-app` (fratello di `security-constraint`)
  - al suo interno va annidato il tag semplice `auth-method`?
    - 4 possibili valori: `BASIC`, `FORM`, `DIGEST` e `CLIENT-CERT`
    - l'ultimo richiede che il client abbia un certificato pubblico, ed è al di là degli scopi di questo corso
    - alcuni servlet container possono anche accettare il valore `NONE`, o altre modalità *vendor-specific*
  - più un altro tag che può essere il semplice `realm-name`? o il composto `form-login-config`?
  - il primo c'è nel caso `auth-method` sia `BASIC` o `DIGEST`
    - nel qual caso, `form-login-config`, se presente, viene ignorato
  - il secondo nel caso `auth-method` sia `FORM`
    - nel qual caso, `realm-name`, se presente, viene ignorato

Autenticazioni BASIC e DIGEST

- Praticamente uguali a livello di programmazione, cambia solo come vengono gestiti dal protocollo HTTP
- In entrambi i casi, la prima richiesta ha come risposta uno status 401 (*unauthorized*)
- Al che, il client fa apparire una schermata popup predefinita di login
  - all'interno di tale schermata di login, viene anche riportata la stringa all'interno del tag `realm-name` del `web.xml`
- Una volta riempita, viene mandata una nuova richiesta nel cui header ci sono username e password
  - in caso di errore con utente sconosciuto o password sbagliata, nuovamente risposta 401, anche all'infinito
  - in caso di utente conosciuto ma non abilitato, risposta 403 (*forbidden*) e schermata definitiva di errore

Autenticazioni BASIC e DIGEST

- Ovviamente, quanto detto sopra vale per i client browser grafici
  - c'è anche il discorso del caching, sul quale torneremo tra poco
- Per client come `curl`, occorre spedire manualmente la seconda richiesta
  - BASIC: `curl --user user:password sito` oppure `curl user:password@sito`
    - usando `user:password@sito` nella barra degli indirizzi di un client browser grafico, si può evitare la prima richiesta
  - DIGEST: `curl --digest -u user:password sito`
    - in un client browser grafico, non si può evitare la prima richiesta

Autenticazione `BASIC` nell'HTTP

- Username e password vengono mandate praticamente in chiaro
  - viene usata la codifica base64, che però è facilmente reversibile
  - data la stringa codificata, è facile ottenere la stringa originaria
  - con `curl --user user:password sito` oppure `curl user:password@sito`, c'è una sola interazione tra client e server
  - il client codifica con base64 e manda la richiesta, il server decodifica e controlla l'autenticazione prima di mandare la risposta

Autenticazione `DIGEST` nell'HTTP

- Viene usato l'hashing MD5
  - non è semplice ottenere la stringa originaria, data la stringa codificata
  - anche usando `curl`, ci sono 2 interazioni:
    - il server risponde con 401, ma nell'header della risposta c'è anche un *nonce* (valore random)
    - il client deve calcolare un valore unico usando username, password, metodo HTTP, URI (tutti noti a priori) più il nonce appena ricevuto
    - il tutto va codificato con l'MD5 (non reversibile) e messo nell'header della nuova richiesta
    - il server rifà lo stesso calcolo e controlla se le 2 hash sono coincidenti
    - se sì, invia la pagina richiesta, altrimenti nuovamente 401
    - ovviamente, se le hash sono ok ma l'utente non ha il giusto ruolo per accedere la risorsa, la risposta sarà 403

Autenticazione FORM

- A livello di programmazione è più flessibile
  - niente popup per il login o pagina di errore predefinita
  - è possibile scrivere delle pagine HTML, JSP o servlet a proprio piacimento per gestire login ed errori
- Il tag composto `form-login-config` contiene obbligatoriamente 2 tag semplici
  - `form-login-page`, con l'url relativo al contesto della pagina di login
    - viene richiesto che contenga un form con azione `j_security_check` e due input di testo con nomi `j_username` e `j_password`
    - non funziona se viene invocata esplicitamente!
  - `form-error-page`, con l'url relativo al contesto della pagina di errore (ci si arriverà a login sbagliato)

Autenticazione Dichiarativa e Sessioni

- BASIC e DIGEST sono legati a specifiche del protocollo HTTP, che è stateless
- Quindi, la richiesta di autenticazione va ripetuta per ogni richiesta, e non c'è nessun concetto di sessione
- Tuttavia, questo è strettamente vero solo per client come `curl`
- I browser grafici, invece, memorizzano il primo login che non ha risposta 401 e lo ripetono per le richieste successive
  - fino a che il browser non viene chiuso
  - ad esempio così fa Firefox
  - quindi viene memorizzato anche un login che ha risposta 403!
  - dà l'*illusione* di una sessione

Firefox e Autenticazione Programmata

- Per fare logout: immettere nella barra degli indirizzi `http://userinesistente:password@sito`
  - vale anche per il DIGEST!
  - c'è anche un metodo `HttpServletRequest.logout`, ma non serve a questo
- Se si usa `user:password@sito`, viene visualizzato il form di login anche se la risposta è 403
- Notare che è quindi impossibile autenticare 2 diversi tab dello stesso client con 2 utenti diversi
  - a meno che non corrispondano a 2 diverse modalità di autenticazione

Autenticazione Dichiarativa FORM e Sessioni

- FORM crea una vera e propria sessione, con tanto di cookie di nome `JSESSIONID`
- Quindi, la richiesta di autenticazione non va ripetuta per ogni richiesta
- Si può fare logout, basta usare `HttpSession.invalidate`
- Protocollo: 2 richieste
  - la prima manda in POST username e password (in chiaro...), e propone un `JSESSIONID`
    - la risposta è 303 ("See Other"), senza corpo
  - la seconda rimanda lo stesso `JSESSIONID`
    - la risposta contiene il cookie `JSESSIONID` con il valore definitivo

Le Pagine JSP

- Un po' pleonastico: JSP sta per "JavaServer Pages"
- Sono un modo un po' più veloce di scrivere pagine HTML dinamiche lato server
- Infatti, si saltano i seguenti passaggi, necessari per le servlet:
  - scrittura di una classe Java che estenda `HttpServlet`
  - compilazione della classe
  - copia del file `.class` dentro `WEB-INF/classes`
  - specifica delle informazioni in `WEB-INF/web.xml`
    - sopravvive solo una parte di quest'ultimo passo: la specifica di un `url-mapping`
    - ma è opzionale
    - gli altri passi non vanno mai fatti

Le Pagine JSP

- Inoltre, semplificano la scrittura della parte HTML "fissa"
  - semplicemente, la si scrive così com'è, come se si trattasse di una pagina HTML vera e propria
- La parte dinamica (variabile) della pagina HTML va scritta aprendo delle "parentesi" nella pagina HTML
  - usando una sintassi speciale, che ora vedremo
- Per la parte dinamica si possono anche usare i *tag standard* e i *tag personalizzati*
- Molto simile ad ASP e PHP, ma:
  - le "parentesi" sono diverse
  - il linguaggio da usare all'interno delle parentesi è Java
  - i tag personalizzati sono trattati in modo standardizzato

Le Pagine JSP: Sintassi

- JSP container: la parte di Tomcat che si occupa delle pagine JSP
  - per ora, rimandiamo la discussione sul suo funzionamento
- Invocato quando viene richiesto un file con estensione `.jsp`
  - si può trovare in qualunque posto dentro un context root
  - ma *non* dentro `WEB-INF` o una sua sottodirectory
  - neanche se è `web-inf`, per i sistemi operativi che lo permettono
- Per invocarli, si può fare come per le pagine HTML
  - o usando il path relativo del file system a partire dal context root
  - o ridefinendo l'URL nel `web.xml`

Le Pagine JSP: Sintassi

- Un file JSP è un file HTML *con estensione* `.jsp` in cui *possono* essere presenti i seguenti delimitatori aggiuntivi:
  - `<% ... %>`
    - *scripting element*: *scriptlet*
  - `<%= ... %>`
    - *scripting element*: *JSP expression*
  - `<%! ... %>`
    - *scripting element*: *JSP declaration*
  - `<%-- ... --%>`
    - commento JSP
  - `<%@ ... %>`
    - *direttiva*
  - tag standard (ci ritorneremo)
  - tag personalizzati (ci ritorneremo)
- Il testo HTML "semplice" è chiamato *template text*

Le Direttive: Sintassi

- Sintassi generica: `<%@direttiva {attributo="valore"}%>`
- Ci sono solo tre direttive: `page`, `include` e `taglib`
- Ognuna ha i suoi attributi, e dà un significato speciale ai corrispondenti valori
- Servono per lo più a dare indicazioni al JSP container su come eseguire il codice all'interno degli scripting elements
  - anche altri usi, come vedremo

Commenti: Sintassi

- Sintassi generica: `<%-- qualsiasi cosa --%>`
- Ovviamente, dentro un commento non ci possono essere i caratteri `--%>`
- Diversi dai commenti HTML:
  - i commenti HTML vengono comunque inviati al client nella risposta HTTP
  - è il browser che interpreta l'HTML a non mostrarli
  - se si apre il sorgente della pagina, li si può vedere
  - invece, i commenti JSP non vengono inviati nella risposta
  - sono visibili solo sul server al programmatore della pagina JSP
- Si possono annidare commenti HTML dentro commenti JSP e viceversa
- Si possono anche usare i commenti Java (`// commento`) dentro le scriptlet

Scripting Elements: Sintassi

- Sintassi generiche:
  - `<%= espressione Java %>`
    - qualsiasi cosa che potrebbe stare a destra di un assegnamento in Java, ma che sia di tipo `String`
  - `<%! dichiarazione membro o metodo Java %>`
    - qualsiasi cosa che potrebbe stare all'interno di una classe Java
  - `<% codice Java %>`
    - qualsiasi cosa che potrebbe stare nel corpo di un metodo Java
    - spesso chiamata *scriptlet*

Il JSP Container

- Per spiegare la semantica delle pagine JSP, è necessario spiegare come funziona il *JSP container*
- È un wrapper del servlet container, quindi è una parte del server Tomcat
- Per gestire una richiesta ad una pagina JSP, l'idea è di creare una servlet e poi usare il ciclo di vita delle servlet
  - ma in modo automatizzato: ci pensa il JSP container
  - inoltre, le servlet generate in questo modo sono un po' diverse dalle servlet "normali"
  - come per le servlet, a regime, ogni invocazione è un thread

Gestione delle Richieste a Pagine JSP

```
for each request of a JSP page p {
  if (∄servlet(p) ∨ time(servlet(p))<time(p)) {
    /* quindi o e' la prima invocazione o il
       sorgente della pagina JSP e' stato
       modificato; non serve ricaricare la
       web application! */
    create(servlet(p));
    if (creation fails) {
      send a standard error page as response;
      continue; }
    compile(servlet(p));
    if (compilation fails) {
      send a standard error page as response;
      continue; }
    init(servlet(p));
  }
  service(servlet(p)); }
```

Pagina JSP: Le Basi

- La cosa più semplice è scrivere tutto in template text
  - ma allora non è dinamica...
- Altrimenti, si mettono degli scripting elements
  - se sono espressioni JSP, il risultato della loro valutazione viene *sostituito* allo scripting element stesso
  - se sono scriptlet, il corrispondente codice viene eseguito
    - ...in un thread...
    - nella risposta, la scriptlet sparisce
    - al suo posto, ci saranno solo, eventualmente, le stringhe passate ad `out.print` o simili

Pagina JSP: Le Basi

- È sempre possibile mischiare template text e scripting elements
- Anche quando ci sono scriptlet che contengono blocchi
  - ad esempio:

```
<% if (una_qualche_condizione) { %>
template_text
<% } %>
```

- Pertanto, non è detto che tutto il template text contenuto in una pagina JSP venga effettivamente inviato nella risposta
  - o che venga inviato una volta sola: al posto dell'`if` potrebbe esserci un `while`...

Servlet da Pagine JSP

- Sotto Linux, si trovano in `/var/lib/tomcat8/work/Catalina/localhost/`*c*`/org/apache/jsp/`*p*`/`*j*`.java`, dove:
  - *c* è il context root
  - *j* è ricavato dal nome della pagina JSP
    - i punti diventano underscore (_)
    - più altre trasformazioni simili
  - *p* è l'eventuale path della pagina JSP dentro il context root

Servlet da Pagine JSP

- Aprendo una servlet generata da una pagina JSP, si notano alcune piccole differenze con le servlet "normali"
  - la classe di partenza non è `HttpServlet`, bensì `HttpJspBase`
    - dal package `org.apache.jasper.runtime`
  - 2 possibilità per inizializzazione e deallocazione:
    - `_jspInit` e `_jspDestroy`: create dal JSP container e non modificabili dall'utente
    - `jspInit` e `jspDestroy`: possono essere specificate dentro una pagina JSP, usando le dichiarazioni JSP
  - il metodo di servizio non è né `service` né `doXxx`, bensì `_jspService`
    - con i consueti 2 argomenti per richiesta e risposta

Servlet da Pagine JSP

- I contenuti degli scripting elements <%!%> sono ricopiati pari pari all'inizio della classe
- Per quanto riguarda gli altri scripting elements ed il template text, li si trova solo dentro `_jspService`
  - tutto il template text è messo, riga per riga, dentro delle chiamate `out.write`
    - opportunamente escappato...
  - il contenuto delle espressioni JSP <%=%> è passato direttamente a `out.print`
  - il contenuto delle scriptlet <%%> viene copiato così com'è
    - eventuali dichiarazioni di variabili contenute nelle scriptlet diventano variabili locali di `_jspService`
    - ogni thread ne avrà una copia tutta sua

Oggetti Per Scripting Elements

- Nelle espressioni JSP e nelle scriptlet si possono usare, in quanto già definiti, i seguenti oggetti:
  - `request` e `response`
    - argomenti di `_jspService`, con il consueto significato
  - tutte le variabili dichiarate all'inizio di `_jspService`, in particolare:
    - `out` (per inserire testo nella risposta)
    - `application` (per accedere ad oggetti condivisi con altre pagine JSP)
    - `session` (idem, ma divisi per utente)
    - `pageContext` e `config` (per accedere alle configurazioni)
    - `page` (poco utile)

Errori nelle Pagine JSP: Traduzione

- Il JSP container non riesce a generare la servlet
- Tipicamente, non sono stati ben chiusi i delimitatori per scripting elements/direttive/commenti
- Conseguenza: se un commento JSP non viene chiuso bene, niente risposta HTTP
- Non chiudere un commento HTML, invece, ha effetti tutt'al più sul rendering della pagina nel browser...

Errori nelle Pagine JSP: Compilazione

- La servlet generata dal JSP container contiene errori Java
- Gli stessi che si possono fare scrivendo una classe Java
- Più possibili errori "di confusione":
  - scrivere un'espressione dentro una scriptlet
  - o un comando dentro un'espressione JSP <%=%>
  - analoghi errori possono essere fatti con le dichiarazioni JSP <%!%>
- Viene visualizzata una risposta simile a quella del compilatore Java
- In caso di dubbi, vedere la servlet generata...

Errori nelle Pagine JSP

Simboli	**Permesso in template text (altrimenti, errore traduzione)**	**Permesso in costante stringa (altrimenti, errore compilazione)**
<%	Solo se chiuso da un corrispondente %>; altrimenti, va corretto in <\%	Sempre
%>	Sempre	Mai, va corretto in <\%
'	Sempre	Può essere corretto in \'
"	Sempre	Mai, va corretto in \"
\	Sempre	Mai, va corretto in \\

Errori nelle Pagine JSP: Esecuzione

- Eccezioni non catturate
- Come può capitare anche con le servlet
- È possibile definire una pagina da visualizzare ogni volta che c'è un'eccezione non catturata
- Altrimenti, c'è una pagina di default

Servlet vs. JSP

- JSP preferibili quando la maggior parte della pagina è fissa, e solo una parte è variabile
- Servlet da preferire nel caso contrario
- Ci sono cose che si possono fare solo con le servlet:
  - ad esempio, non è possibile richiedere una pagina JSP in `PUT` o `DELETE`
  - quindi le pagine JSP non possono essere usate per applicazioni RESTful
- Nelle servlet si può mettere esattamente ciò che serve
  - ad esempio, nelle pagine JSP c'è l'oggetto `application` anche se poi non lo si usa

La Direttiva page

- Come tutte le direttive, cambia il modo con cui la pagina JSP viene tradotta in servlet
- Ammette i seguenti attributi:
  - `import`, il cui valore viene usato per le istruzioni Java `import`
    - vengono messe all'inizio del file della servlet tradotta
    - si possono specificare più package, separandoli con la virgola
  - `errorPage`, il cui valore è la pagina da visualizzare quando c'è un errore
  - `isErrorPage`, il cui valore è un booleano
    - se `true`, ci sarà una variabile in più: `exception`
    - affinché abbia senso, dev'essere una pagina che si trova nell'`errorPage` di un'altra pagina
    - il default è `false`

La Direttiva page

- Altri attributi:
  - `isThreadSafe`, il cui valore è un booleano
    - se è `false`, allora al più un thread per volta potrà eseguire questa pagina
    - se arrivano più richieste contemporaneamente, verranno sequenzializzate, e mai eseguite in parallelo
    - ovviamente, il default è `true`
  - `session`, il cui valore è un booleano
    - se `true`, ci sarà una variabile in più: `session`
    - di default è `true`

La Direttiva `include`

- Un solo attributo interessante: `file`
  - dev'essere dentro l'attuale context root
  - se il suo path comincia con `/`, allora si intende essere relativo al context root
  - altrimenti, è relativo al punto dove si trova il file jsp
    - o relativo al suo path virtuale, se definito in `web.xml`
  - il file non può essere un cammino virtuale di `web.xml`
- Il JSP container, nel punto in cui incontra la direttiva:
  1. apre il file corrispondente
  2. lo traduce esattamente come sta facendo con quello attuale
     - nel farlo, potrebbe incontrare altre direttive include...
- Pertanto, la traduzione viene fatta *a tempo di compilazione*
  - vedremo che è possibile farlo anche a tempo di richiesta

I Tag Standard

- Detti anche tag predefiniti, o azioni standard
- Sono in numero molto limitato:
  - `jsp:include`
  - `jsp:forward`
  - `jsp:param`
    - da usare solo annidato in `jsp:include` o `jsp:forward`
  - `jsp:useBean`
  - `jsp:setProperty` e `jsp:getProperty`
    - da usare solo in combinazione con `jsp:useBean` o `jsp:forward`
  - `jsp:expression`, da usare come alternativa a <%=%>

Inclusione con `jsp:include`

- Inclusione *a tempo di richiesta*
- Contrapposto alla direttiva `<%@include%>`, che invece lavora a tempo di compilazione
- Il JSP container la traduce in una opportuna chiamata di funzione
  - ovviamente, dentro `_jspService`
- Svariate differenze
  - la risorsa da includere può essere:
    - nel caso dell'azione, qualsiasi risorsa raggiungibile tramite URL, purché all'interno dell'attuale context root
    - nel caso della direttiva, qualsiasi file nell'albero di directory radicato nell'attuale context root
    - ad esempio, nel caso della direttiva non si potrà mai includere (la risposta di) una servlet...

Inclusione con `jsp:include`

- Altre differenze
  - per l'azione, la risorsa da includere deve essere invocabile come URL
    - quindi, se ad esempio è una pagina JSP, deve essere sintatticamente corretta
    - ovvero, traducibile e compilabile dal JSP container
  - per la direttiva, non è necessario
    - ad esempio: può anche finire con un delimitatore <% aperto...
    - ... se poi viene opportunamente chiuso dopo l'invocazione della direttiva
    - altro esempio: una scriptlet potrebbe avere una parentesi graffa aperta e non chiusa, purché...

Inclusione con `jsp:include`

- E se viene modificata (sul server) la risorsa da includere?
  - per l'azione, non c'è problema:
    - ogni visita alla pagina implica una nuova richiesta alla risorsa da includere
    - quindi, verrà sempre ottenuta la risorsa modificata
  - per la direttiva, bisogna forzare la ricompilazione della pagina JSP che fa l'inclusione
    - fare il reload non basta
    - occorre un comando `touch`, ma se non si ha una shell simil-`bash`?
    - si può modificare fittiziamente il file
    - una soluzione portabile non esiste

Sintassi per Redirezione ed Inclusione

- Sono entrambi tag XML, quindi differentemente da quelli HTML sono case sensitive
  - occorre sempre chiuderli nel modo corretto, altrimenti ci sarà un errore di creazione della servlet
- Specificano la risorsa da includere/cui redirigere con un attributo `page`
  - o relativo semplice o relativo al context root
  - quindi, il context root non va mai specificato, e si intende essere quello attuale
- C'è anche un attributo booleano `flush`
  - c'era anche per la direttiva include
  - ma, sia per le azioni che per la direttiva, viene richiesto che sia `true`, che è anche il default
  - teoricamente, se `false` dovrebbe permettere il buffering

Sintassi per Redirezione ed Inclusione

- Possono essere o vuoti o annidare al loro interno un tag `<jsp:param>`
  - il tag `jsp:param` è sempre vuoto e ammette 2 attributi: `name` e `value`
  - se i loro valori sono *n* e *v*, vorrà dire che, nella richiesta che viene fatta, sarà presente l'associazione *n*=*v*
  - se la richiesta era di GET, allora è lo stesso che mettere la querystring ?*n*=*v*
  - quindi, la pagina di destinazione potrà chiamare `request.getParameter(`*n*`)` per ottenere *v*
  - è un modo per passare dei parametri

Redirezione ed Inclusione: e `RequestDispatcher`?

- Per farla breve: non funziona, o almeno non sempre
- La redirezione va bene
- L'inclusione no, perché le servlet "normali" e quelle delle pagine JSP lavorano su stream diversi
  - `JspWriter` contro `PrinterWriter`
  - l'uno preso dal `pageContext`, l'altro dalla `response`
- Il risultato finale è che l'inclusione ha lo stesso effetto della redirezione
- Come per `RequestDispatcher`, gli indirizzi relativi vanno intesi rispetto alla pagina JSP *sorgente*
  - per le redirezioni, la barra degli indirizzi non cambia

Altri Paralleli

- La variabile predefinita `session` è uguale a `request.getSession()`
  - ammesso che l'attributo `session` di `page` sia true...
- La variabile predefinita `config` è uguale a `getServletConfig()`
- La variabile predefinita `out` è *diversa* da `response.getWriter()`
- La variabile predefinita `application` è uguale a `getServletContext()`
- Manca, nelle servlet, il corrispettivo di `page`

Gli “Scope” delle Pagine JSP

- È possibile accedere a informazioni a diversi livelli di visibilità (*scope*)
  - sempre nella forma chiave=valore, con la chiave di tipo `String` e il valore `Object`
  - sempre chiamando opportunamente `getAttribute` o `setAttribute` su un opportuno oggetto
- Sono possibili anche per le servlet, sfruttando le equivalenze di cui sopra
  - tranne che per lo scope di pagina, che è completamente inutile per le servlet
  - e comunque non proprio indispensabile neanche per le pagine JSP
- Permettono lo scambio di informazioni (sotto forma di stringa) tra diverse pagine JSP

Lo Scope di Pagina

- `pageContext.getAttribute`, `pageContext.setAttribute`
- In pratica, comunicazione possibile solo all'interno della pagina stessa
- Unica eccezione: anche tra pagine diverse, a patto che una includa l'altra
  - ma solo con la direttiva include!
  - spiegare il perché...
- Evitabile in entrambi i casi
  - dichiarando, ad esempio, un oggetto di tipo mappa nella pagina
  - infatti, nelle servlet non c'è...

Lo Scope di Richiesta

- `request.getAttribute`, `request.setAttribute`
- Comunicazione possibile solo se una pagina fa una redirezione a/un'inclusione di un'altra
  - per quanto riguarda l'inclusione, solo con l'azione standard!
  - spiegare il perché...
- Da notare che la `setAttribute` è rimpiazzabile da `<jsp:param>`, come già visto
  - però poi il valore è una stringa
  - se si vogliono passare informazioni strutturate può essere complicato
- Funziona anche da/verso servlet

Lo Scope di Applicazione

- `application.getAttribute`, `application.setAttribute`
- Comunicazione possibile tra tutte le pagine di una stessa web application
  - pertanto, se una pagina JSP chiama `application.setAttribute(a, b)`...
  - ... una qualsiasi altra pagina JSP nella stessa web application, chiamando `application.getAttribute(a)`, vede ritornarsi *b*
  - lo stesso dicasi di una servlet che usi `getServletContext().getAttribute(a)`
  - o se è una servlet a chiamare `getServletContext().setAttribute(a, b)`
- L'informazione non "muore" mai, se non quando la web application viene ricaricata
  - a meno di non chiamare `application.removeAttribute(a)`...

Lo Scope di Sessione

- `session.getAttribute`, `session.setAttribute`
- Valgono le stesse cose dette per le sessioni delle servlet
- Quindi, è come lo scope application, solo che utenti diversi possono vedere abbinamenti chiave-valore diversi
  - invece, nello scope application, utenti diversi vedranno sempre gli stessi abbinamenti
- L'informazione non "muore" mai
  - neanche se l'applicazione viene ricaricata, o se Tomcat viene fatto ripartire!
    - `http://tomcat.apache.org/tomcat-5.5-doc/config/manager.html#Persistence_Across_Restarts`
  - a meno di non chiamare `session.removeAttribute(`*a*`)`, o addirittura `session.invalidate`
  - oppure, se è stato settato il timeout della sessione

Oggetti Impliciti e Scope

- È pertanto ovvio che:
  - `application` lavora a livello application
  - `session` lavora a livello sessione
  - `request` lavora a livello di richiesta
  - tutte le altre a livello di pagina: `out`, `response`, `exception` (se presente), `config`, `page`, `pageContext`

Pagine JSP e Java Beans

- Nelle pagine JSP si può interagire con particolari classi Java denominate Java Beans
  - si può fare anche con le servlet
  - ma con le pagine JSP ci sono scorciatoie sintattiche che val la pena considerare
- Lo strumento sintattico è dato dalle 3 azioni standard `jsp:useBean`, `jsp:setProperty` e `jsp:getProperty`

I Java Beans

- Sono classi Java all'interno delle quali vengono rispettate alcune norme sintattiche
- Prima di dare tali norme, una chiosa lessicale
  - normalmente, le variabili "globali" dichiarate all'interno di una classe Java vengono chiamate "attributi" o "membri" della classe
  - per i Java Bean, si preferisce il termine *proprietà* (*property*)
  - il motivo è che le proprietà possono essere sia membri della classe, che informazioni ricavabili da questi ultimi
  - ad esempio, se una classe `Utente` ha l'attributo `dataNascita`...
  - ... allora ha entrambe la proprietà `dataNascita` e `etaAnagrafica`

I Java Beans

- Regole imprescindibili affinché una classe Java sia un Java Bean:
  1. deve essere presente un costruttore con zero argomenti
     - ok anche se non c'è nessun costruttore: in questo caso, viene automaticamente definito un costruttore di default con 0 argomenti
  2. per ogni proprietà da "esporre" vanno definiti un getter ed un setter
     - con nome standard camel-like
     - `getEtaAnagrafica`, `setDataNascita`
     - `isXxx` per proprietà booleane, ad es. `isMaggiorenne`

I Java Beans

- Regola di "buona programmazione": non devono esistere altri modi di modificare le proprietà
  - a parte i setter detti sopra
- Quindi:
  - niente membri pubblici, altrimenti basta fare `oggetto.membroPubblico = valore;`
  - niente altri metodi, come `modificaMembroPubblico()`

I Java Beans e le Servlet

- È sufficiente usare il Java Bean quando necessario
  - se il file `class` non è nella stessa directory, o si usa il path completo o l'`import`
- Per mandare un Java Bean ad un'altra servlet, basta usare `setAttribute` nell'appropriato scope (sessione, richiesta, applicazione)
- L'altra servlet farà un corrispondente `getAttribute`

I Java Beans e le Pagine JSP

- Il codice va messo dentro `WEB-INF/classes`, e dev'essere dentro un package
  - quindi, occorre creare una directory dentro `WEB-INF/classes`, e metterci dentro il file `.class` del Java Bean
  - foss'anche l'unico file del package
- Per usare un Java Bean dentro una pagina, occorre usare l'azione standard `<jsp:useBean>`
  - dopodiché, per ottenere il valore delle proprietà si usa `<jsp:getProperty>`
  - per modificare il valore delle proprietà si usa `<jsp:setProperty>`
  - non è tuttavia obbligatorio, si può fare anche con le scriptlet

L'Azione Standard `jsp:useBean`

- Il tag `<jsp:useBean>` può essere sia vuoto che contenere degli "statement"
  - sulla semantica del caso non-vuoto ritorneremo tra poco
- Ammette i seguenti attributi:
  - `id`, il cui valore è una stringa
  - `class`, il cui valore è una stringa
  - `scope`, il cui valore è uno dei seguenti: `page`, `request`, `session`, `application`

L'Azione Standard `jsp:useBean`

- Il tag `<jsp:useBean>` può essere sia vuoto che contenere codice JSP
- Ammette i seguenti attributi:
  - `id`, il cui valore è una stringa
    - il nome dell'oggetto Java Bean
  - `class`, il cui valore è una stringa
    - la classe del Java Bean, completa del percorso del package
  - `scope`, il cui valore è uno dei seguenti: `page` (default), `request`, `session`, `application`

L'Azione Standard `jsp:useBean`

- `<jsp:useBean id=`*i* `scope=`*s* `class=`*c*`>`
- Cerca se esiste un bean con nome *i* nello scope *s*
- Se sì, ritorna quel bean, altrimenti ne crea uno nuovo di tipo *c*
  - nella versione non-vuota, esegue il codice JSP annidato all'interno del tag solo se ha creato il bean
  - se invece era già stato creato, non esegue il codice
- Nella servlet generata, viene dichiarata una variabile di nome *i* e tipo *c*
- Viene chiamata una funzione che cerca il bean di nome *i* nello scope *s*
- Se non viene trovato, la variabile *i* viene inizializzata con una `new` e l'oggetto ottenuto viene inserito nello scope *s*
  - ecco perché serve il costruttore di default...

L'Azione Standard `jsp:useBean`

- C'è un ulteriore attributo: `type`
  - se non viene dato, si assume uguale a *c* (valore dato a `class`)
- Può essere settato ad una superclasse *t* di *c*
- Nella servlet generata l'oggetto dichiarato è di tipo *t*, e viene invocato il costruttore di *c*
- Se viene specificato solo `type` e non `class`, allora non viene invocato alcun costruttore!
  - l'oggetto dovrà già esistere nello scope dato, altrimenti eccezione
  - almeno uno tra `type` e `class` ci deve essere, altrimenti errore di traduzione
- Conseguenza: gli si può anche passare un oggetto che non ha il costruttore di default

L'Azione Standard `jsp:getProperty`

- `<jsp:getProperty name=`*i* `property=`*p*`>`
- Chiama il getter `get`*p*`()` sul bean con id *i*
  - la prima lettera di *p* è trasformata in maiuscola
- Chiaramente, se il bean con id *i* non è stato creato/trovato o se il getter non esiste, errore di compilazione
- Nella servlet generata, chiama *i*`.get`*p*`();`

L'Azione Standard `jsp:setProperty`

- Viene in 3 "sapori":
  - `<jsp:setProperty name=`*i* `property=`*p* `value=`*v*`>`
    - *v* può essere il risultato di un'espressione JSP
  - `<jsp:setProperty name=`*i* `property=`*p* `param=`*r*`>`
  - `<jsp:setProperty name=`*i* `property="*">`
- Il primo chiama il setter `set`*p*`(`*v*`)` sul bean con id *i*
  - valgono considerazioni analoghe al getter
- Il secondo anche, ma al setter viene passato `request.getParameter(`*r*`)`
- Nel terzo, vengono chiamati *un insieme* di setter del bean
  - tutti quelli che corrispondono a proprietà nella richiesta
  - il valore per il setter set*p* sarà `request.getParameter(`*p*`)`

Pagine JSP e Java Beans

- L'uso tipico che se ne fa è quello di scambio di informazioni
- Soprattutto se strutturate:
  1. si definisce una classe Java Bean con tutte le informazioni
  2. la si instanzia e si settano le proprietà che interessano
  3. la si manda a chi di dovere
  4. chi la riceve può leggere le proprietà che servono
- Va bene per tutte le comunicazioni: JSP-JSP, servlet-JSP, JSP-servlet, servlet-servlet
  - ma ovviamente l'uso tipico è quello in cui una pagina JSP è la *destinazione* dello scambio di informazioni

Conversioni di Tipo per Proprietà

- Per i getter è facile: basta usare `toString`
  - ovviamente, se è un tipo base occorre usare la versione oggetto (ad es., non `int` ma `Integer`)
  - quello che serve nella pagina è sempre una stringa
- Per i setter, ci vuole una conversione apposita per ogni tipo
- Esempi:
  - per `boolean`, `Boolean.valueOf(paramString).booleanValue()`
  - `Double`, `Double.valueOf(paramString)`
  - vedere Table 13.1 nel libro di testo

Espressioni JSP in Attributi di Tag

- Le espressioni JSP si possono usare anche per dare un valore a degli attributi
  - ad es.: `<table border='<%= "1"%>'>` è equivalente a `<table border='1'>`
- Ma quando si parla di azioni standard, ci sono delle restrizioni
- Si possono usare solo con i seguenti attributi:
  - `name` di `jsp:setProperty`
  - `value` di `jsp:setProperty` e di `jsp:param`
  - `page` di `jsp:include` e di `jsp:forward`
- Quindi, cose come `<jsp:useBean id='<%= "nome"%>'/>` non sono ammesse

I Tag Personalizzati

- Esistono solo 6 tag predefiniti
  - per redirezione, inclusione e uso di Java Bean
- Lo standard JSP 1.1 permette di definirne di nuovi a piacere
- L'idea è la seguente:
  - si definisce un nuovo tag, precisandone nome (obbligatorio), attributi, contenuto
  - si definisce una classe Java i cui metodi verranno opportunamente chiamati quando il JSP container incontra il tag
  - il tag non comparirà nella risposta HTTP
  - può essere sostituito dall'output di questi metodi

I Tag Personalizzati

- Metodo potente, è un'esclusiva di J2EE
- Permette ad un programmatore di definire un modo standard per visualizzare delle informazioni
  - purché le si incapsuli dentro un tag personalizzato
- Il programmatore di pagine JSP può riusare tali tag quando necessario
  - i due programmatori qui menzionati possono essere diversi
- Per cambiare la visualizzazione, basta modificare la classe Java, e non serve toccare le pagine JSP

I Tag Personalizzati: Sintassi

- Definizione:
  - scrivere una classe Java che estenda una classe tra `TagSupport` o `BodyTagSupport`
    - come per i Java Bean, va messa in un package, anche se è una classe sola
  - scrivere uno o più file XML detto TLD, che specifichi le caratteristiche di un insieme di tag personalizzati
    - TLD: Tag Library Descriptor
    - per ogni tag: nome, attributi, se è vuoto o no, classe handler
- Uso:
  - occorre usare la direttiva `<%@taglib%>` per indicare il file TLD
  - infine, si può usare quante volte si vuole il tag, purché non si sia dentro uno scripting element

Definizione della Classe Handler

- Come detto, si può scegliere tra estendere `TagSupport` o `BodyTagSupport`
  - dipende anche dalla versione delle specifiche JSP: dalla 1.2 viene consentito di usare `TagSupport` più spesso
  - qui si farà riferimento a tale versione
  - le suddette classi implementano, rispettivamente, le interfacce `Tag` e `BodyTag`
    - la seconda estende la prima
- È necessario `BodyTagSupport` quando si vuole modificare il corpo del tag
  - o quando si vuole un maggiore "controllo"
- Altrimenti, si può usare `TagSupport`
  - quindi è il caso di tag vuoti, o tag che non devono modificare il proprio contenuto

Definizione della Classe Handler

- Come per le servlet, ci si aspetta di ridefinire il corpo di alcuni metodi
  - e di lasciarne inalterati altri
- E come per le servlet, c'è un *ciclo di vita*
  - invocato *per ogni istanza* del tag presente nella pagina JSP
  - per implementare dei tag personalizzati, occorre conoscerlo
  - le interfacce `Tag` e `BodyTag` hanno cicli di vita diversi
  - quello di `BodyTag`, ovviamente, estende quello di `Tag`
- Una volta instanziata, la classe handler resta in memoria
  - riusata per eventuali altre richieste alla stessa pagina
  - ma ogni pagina ha i suoi oggetti istanziati separatamente
    - vedere `_jspInit`
  - muore solo se si stoppa la web application
    - vedere `_jspDestroy`

Ciclo di Vita per l'Interfaccia `Tag`

- Se il tag è vuoto, allora il ciclo di vita è costituito da $5 + s$ metodi chiamati dal JSP container:
  1. `void setPageContext(PageContext)`
  2. `void setParent(Tag)`
  3. invocazione metodi setter per eventuali attributi del tag
  4. `int doStartTag() throws JspException`
  5. `int doEndTag() throws JspException`
  6. `void release()`
- Ci si aspetta che il programmatore di tag ridefinisca, al più, i metodi dal 3 in giù
- Inoltre, il programmatore di tag può chiamare `Tag getParent()`
  - serve per i tag che possono annidare al loro interno altri tag
  - non verranno considerati nell'attuale versione di questo corso

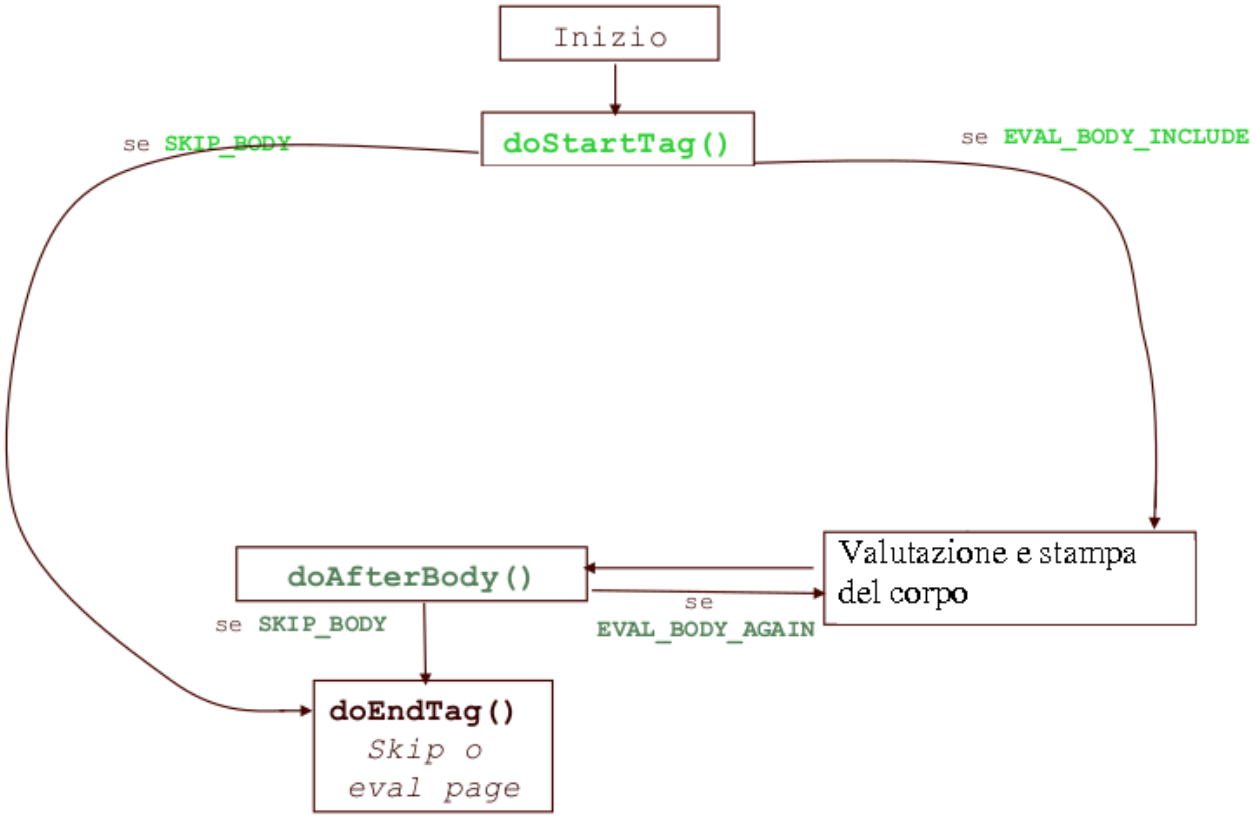
Ciclo di Vita per l'Interfaccia `Tag`

- I passi nel dettaglio:
  1. fa in modo che la classe handler abbia accesso a tutti gli oggetti impliciti della pagina
     - `request`, `response`, `out`...
     - ottenibili tramite `pageContext.getRequest()`, `pageContext.getResponse()`, `pageContext.getOut()`, ...
  2. se il tag è annidato in un altro tag t, allora setta t come il suo "parent"
     - altrimenti, è la pagina JSP stessa
     - è ciò che verrà ritornato in caso si invochi `getParent`
  3. azioni da fare per ogni attributo specificato nell'istanza attuale del tag
  4. azioni da fare non appena si trova l'apertura del tag
  5. azioni da fare dopo la chiusura del tag
  6. azioni da fare quando la pagina JSP viene distrutta

Ritorno per doStartTag e doEndTag

- Il valore ritornato da doStartTag e doEndTag viene usato da _jspService per decidere le azioni susseguenti
- L'interfaccia Tag definisce 4 costanti, da usare per la return di tali metodi
  - SKIP_BODY: non considerare il corpo, anche se c'è
  - EVAL_BODY_INCLUDE: copia il corpo così com'è
    - se il corpo contiene scripting elements o altri tag (personalizzati o no), valutali normalmente
  - SKIP_PAGE: termina qui la valutazione della pagina JSP
    - eventuali altri contenuti vengono ignorati nella risposta HTTP
  - EVAL_PAGE: continua con la normale valutazione della pagina JSP
- I primi 2 sono per doStartTag, gli altri per doEndTag

Ciclo di Vita Completo per l'Interfaccia `Tag`



Tag Library Descriptor

- È un file XML, con schema `http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd`
- Il tag che annida il tutto è `taglib`; al suo interno:
  - `tlib-version`: per tenere conto di diverse versioni della libreria
  - `short-name`: per tool che elencano questi file...
  - opzionali: `display-name`, `description`
  - zero o più tag `tag`, che danno le informazioni sui tag personalizzati
- Da mettere, nel filesystem, dentro il sottoalbero radicato nel context root

Tag Library Descriptor: Tag `tag`

- A parte i soliti opzionali `display-name` e `description`, contiene i seguenti tag:
  - `name`: verrà usato come nome del tag personalizzato che si sta creando
  - `tag-class`: classe Java dell'handler; occorre tutto il percorso del package, a partire da `WEB-INF/classes`
  - `body-content`: può prendere solo i seguenti valori:
    - `empty`: il tag dovrà essere vuoto
    - `JSP`: il tag può contenere qualsiasi codice JSP
    - `scriptless`: il tag può contenere template text, espressioni JSP e tag predefiniti
    - `tagdependent`: il contenuto del tag può essere interpretato dall'implementazione
    - ad es.: sarà una query SQL da passare ad un DBMS

Tag Library Descriptor: Tag `tag`

- Altri tag all'interno di `tag`:
  - `attribute` (zero o più): definisce gli attributi del tag, ed è a suo volta composto da:
    - `name`: il nome dell'attributo
    - `description`: al solito...
    - `required`: booleano, ad indicare se è obbligatorio oppure no
    - `rtexprvalue`: booleano, va messo a true se si vuol permettere la valutazione di espressioni JSP nel valore dell'attributo
  - l'unico obbligatorio è `name`
    - `required` e `rtexprvalue` sono falsi di default

Attributi Con Espressioni JSP

- Non tutti gli attributi dei tag possono avere, come valore, delle espressioni JSP
  - in tal caso, il valore viene calcolato a tempo di richiesta, e non a tempo di traduzione
- Lo sono tutti gli attributi di tag XHTML
- Più i seguenti:
  - `name` e `value` di `jsp:setProperty`
  - `value` di `jsp:param`
  - `page` di `jsp:include` e `jsp:forward`
- Più gli attributi dei tag personalizzati per i quali `rtexprvalue` è `true`

Gestione degli Attributi

- Il ciclo di vita del tag prevede l'invocazione di metodi setter prima del `doStartTag`
  - non tutti quelli definiti nel TLD; bensì solo quelli effettivamente settati per l'attuale istanza del tag personalizzato
- La convenzione è sempre quella camel-like
  - se il nome dell'attributo è `xxx`, allora occorre dichiarare il setter `void setXxx(String)`
- Tipicamente, il setter si limita ad assegnare valori ad uno o più membri della classe handler
- Così da poterli poi usare in `doStartTag` o `doEndTag`

La Direttiva `taglib`

- Ha 2 attributi
  - `uri`: il percorso del file TLD
    - può anche essere un URL su Internet
    - oppure un file nel context root attuale
  - `prefix`: da prefiggere al nome del tag
    - quindi, se il tag, nel file TLD, si chiama *n*...
    - ... e `prefix` vale *p*...
    - ... allora il nome completo del tag da usare nella pagina JSP sarà *p*:*n*

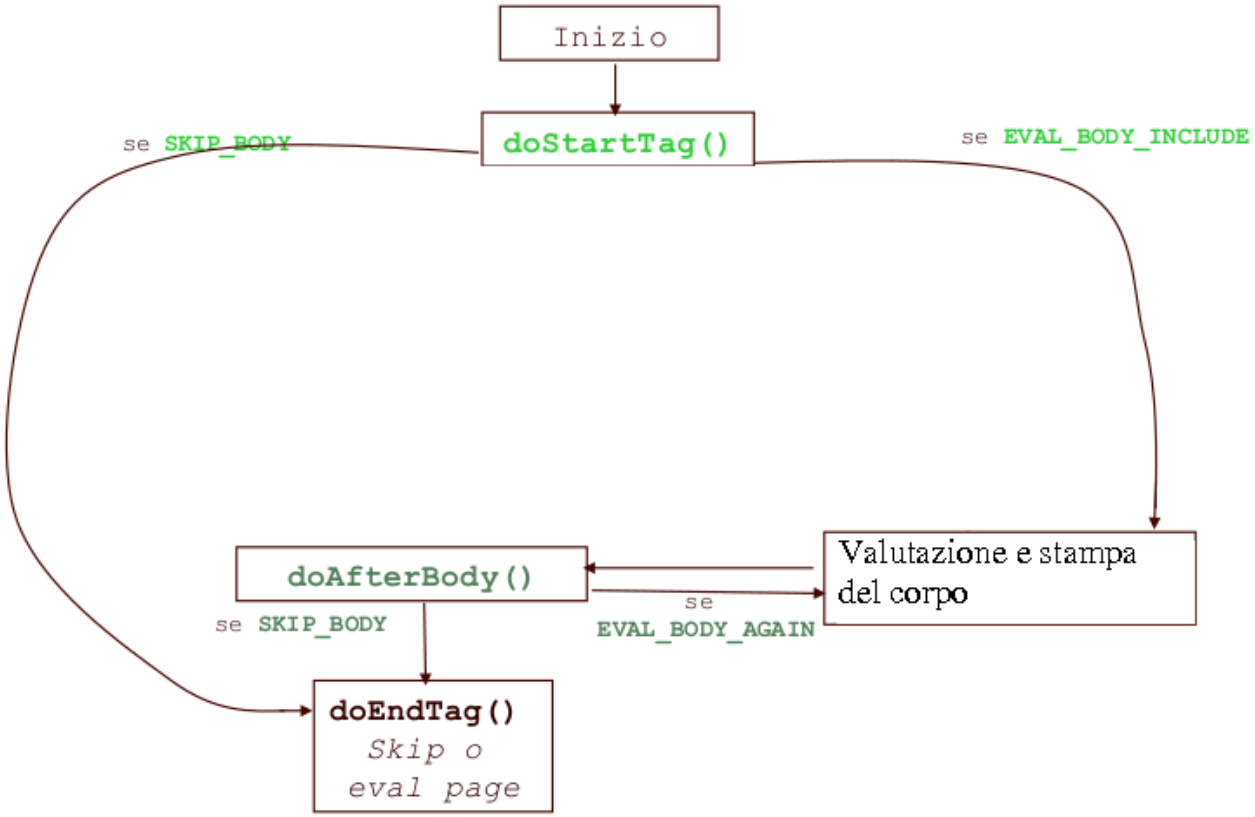
“Modificare” il Corpo dei Tag

- Manipolazioni del corpo del tag sono permesse anche dall’interfaccia `Tag`
  - viene permesso di ripetere la sua valutazione più volte
  - se il corpo non è template text, è possibile fare in modo che ciascuna valutazione sia diversa
- Tuttavia, non è possibile controllare completamente la scrittura sullo stream
  - ciascun `return EVAL_BODY_AGAIN` dà luogo ad una valutazione seguita da una scrittura
  - ad esempio, non si può decidere alla fine di tutte le ripetizioni se scrivere o no il contenuto del corpo

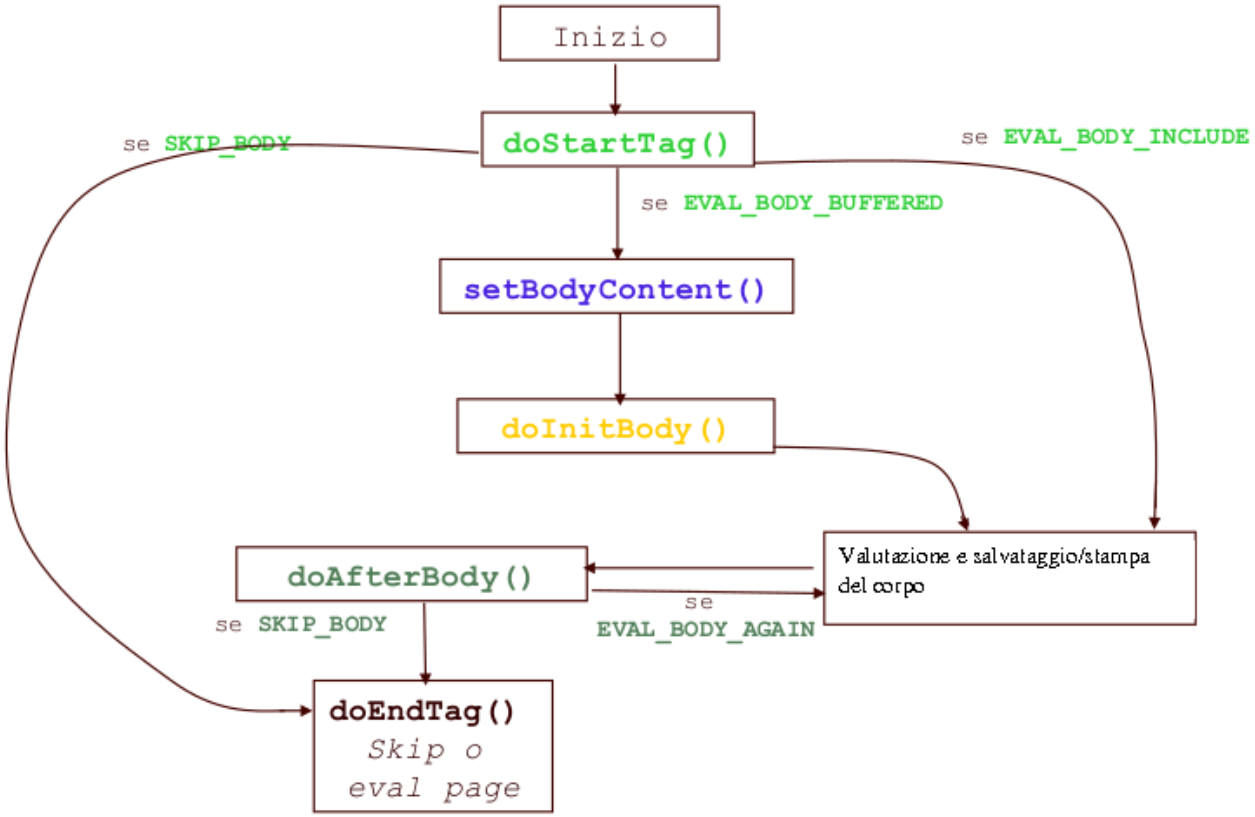
“Modificare” il Corpo dei Tag

- Basta capire bene il ciclo di vita dell’interfaccia `BodyTag`
- `BodyTag` estende `Tag`, quindi i metodi di cui sopra sono confermati
- Ma `doStartTag` ha un possibile valore di ritorno in più: `EVAL_BODY_BUFFERED`
- Nel qual caso, verranno invocati i seguenti metodi aggiuntivi:
  - `setBodyContent` (da non modificare)
  - `doInitBody`
- Infine, la valutazione del corpo non implica la sua scrittura nello stream: andrà fatto esplicitamente
  - con possibilità di scegliere lo stream

Tag: **Ciclo di Vita**



BodyTag: **Ciclo di Vita**



“Modificare” il Corpo dei Tag

- L’interfaccia `BodyTag` offre poi i seguenti altri metodi
  - funzionano solo se si sta usando `EVAL_BODY_BUFFERED`!
  - `JspWriter getPreviousOut()`: ritorna lo stream usato, per la risposta, da chi contiene il tag
    - può essere o il tag che annida quello corrente, oppure la pagina JSP stessa
    - da usare se si vuole scrivere qualcosa nella risposta
    - e si vuole scegliere l’opportuno livello di nesting dove scriverlo
  - non usare `pageContext.getOut()`: sono stream potenzialmente diversi
  - `BodyContent getBodyContent()`: mostra l’attuale contenuto
    - può essere scritto in vari modi sullo stream desiderato

L'Interfaccia `BodyContent`

- Vari modi per scrivere il suo contenuto nella risposta
  - `getBodyContent().writeOut(getPreviousOut())`
  -
    ```
    getBodyContent().writeOut(
      getBodyContent().getEnclosingWriter()
                                )
    ```
  - `getPreviousOut().write(getBodyContent().getString())`

L'Interfaccia `BodyContent`

- Per *modificare* il valore di un oggetto `BodyContent`, *non* bisogna usare `setBodyContent`
  - quello è un metodo chiamato dal JSP container
- Invece, si sfrutta la valutazione "iterativa" mostrata nella figura sul ciclo di vita
  - il contenuto di `BodyContent` viene aggiornato:
    - dopo `doInitBody`
    - ogni volta che `doAfterBody` ritorna `EVAL_BODY_AGAIN`
  - ogni aggiornamento consiste nel:
    - valutare il corpo del tag
    - concatenare il risultato all'attuale contenuto di `BodyContent`