# Programmazione per il Web
# Riassunto della lezione del 02/03/2015

Igor Melatti

## Verso le Pagine Web Dinamiche

- Slides 58–68: riassunto dal corso di Reti
  - in particolare, il fatto che sia senza stato costringe ad alcune acrobazie i programmatori Web
  - infatti, memorizzare uno stato è spesso indispensabile
    - ad esempio, per vedere il proprio conto corrente occorre prima essere autenticati: quindi ci sono almeno due stati in cui la pagina web del conto viene visualizzata, autenticato e non autenticato, dipendentemente da visite a pagine Web fatte precedentemente
  - nel seguito del corso verranno presentati diversi modi di gestire lo stato, aggirando il problema
- Slide 60: un possibile metodo (un po' macchinoso) per gestire uno stato anche se l'HTTP non lo supporta
  - il Web Server risponde a delle richieste fatte tramite il metodo "GET" da un form HTML
  - come si può vedere, con il metodo "GET" l'URL (qui mostrato senza l'indirizzo del server) contiene le informazioni che vengono mandate (le informazioni cominciano dopo il punto interrogativo)
  - comunque, il trucco funzionerebbe anche con il metodo "POST"
  - assunzione: sul Web server c'è una qualche applicazione che si chiama `miaservlet` e che è capace di accorgersi di com'è scritto l'URL in arrivo, e risponde con una pagina HTML di contenuto diverso in funzione di tale URL
  - ad esempio, tra la prima e la seconda freccia blu la differenza è il `Cognome=Rossi`
  - a seconda di quali informazioni sono già state inviate (quindi, a seconda dello "stato"), il form ha apparenze diverse
  - il funzionamento di `miaservlet` è il seguente:

1. si visita una pagina HTML che ha un form dove si inserisce solamente il nome
2. dopo aver inserito il nome e cliccato sul pulsante di sottomissione, si manda il nome a `miaservlet`, come mostrato nella prima freccia blu in alto nella slide
3. `miaservlet` risponde (prima freccia rossa) con una nuova pagina HTML, molto simile alla precedente, ma in cui il campo del nome è hidden e fissato a ciò che è stato immesso al passo precedente (nell'esempio, "Giovanni"); c'è inoltre un campo in più (l'unico visibile), per il cognome
4. dopo aver inserito il cognome e cliccato sul pulsante di sottomissione, si manda il cognome a `miaservlet`, come mostrato nella seconda freccia blu
   * c'è anche il nome, perché è quello che succede coi campi hidden: non vengono visualizzati dal browser, ma il loro contenuto (fissato) viene comunque mandato insieme ai campi visibili
5. `miaservlet` risponde (seconda freccia rossa) con una nuova pagina HTML, molto simile alla precedente, ma in cui i campi del nome e del cognome sono hidden e fissati a ciò che è stato immesso al passo precedente (nell'esempio, "Giovanni" e "Rossi"); c'è inoltre un campo in più (l'unico visibile), per l'indirizzo
6. e così via

– quindi, nonostante l'HTML non abbia uno stato, si è riuscito ad introdurne uno

– ovvero, cosa viene visualizzato in una particolare pagina HTML dipende da cosa si è visitato (o inviato) prima

– nota: `miaservlet` non può memorizzare le informazioni al suo interno man mano che gli arrivano!

– questo perché `miaservlet` deve poter essere invocata da svariati utenti contemporaneamente, anche provenienti dallo stesso host

– ultima nota: mettere il campo hidden permette di far sì che l'utente non possa più modificare quanto immesso in precedenza, e quindi che non possa modificare lo "stato" con cui è arrivato alla pagina

- Slide 64

  – esempio per server software: Apache, IIS

  – content-type: non è detto che sia un HTML, potrebbe essere un'immagine, o una pagina HTML compressa, o testo formattato (es.: JSON)

- Slide 68: la distinzione funzionale è ritornata in auge nelle applicazioni RESTful

- Slide 69: pagine "dinamiche" nel senso che il loro contenuto varia a seconda delle informazioni ulteriori che vengono inviate con le richieste GET o POST
  - o anche a seconda dello "stato" (da introdurre con metodi vari)
- Slide 70: le tecnologie lato server elencate qui fanno tutte riferimento al Java Enterprise Edition (tipicamente indicata come J2EE)
  - si tratta di una suite di programmazione della Oracle che permette di costruire applicazioni professionali per le imprese
  - Java Transaction Management, vedere qui: `http://en.wikipedia.org/wiki/Java_Transaction_API`
  - questo corso è principalmente basato su questa suite
  - Java Server Pages fa pure parte di questa suite, e permette di mischiare HTML statico e dinamico (in questo è simile al PHP e all'ASP)
- Slide 72: CGI sta per "Common Gateway Interface"; si tratta di eseguire un programma (C/C++ o Perl), il cui output è la pagina HTML richiesta (quindi occorre che l'output del programma rispetti la sintassi dell'HTML...)
- il programma CGI può accedere opportunamente a delle variabili che descrivono l'input (ad esempio, i campi di un form inviati come richiesta)
- Slide 73: sono comunque soluzioni molto usate, soprattutto nei casi in cui non si ha la necessità di usare le API per le imprese che sono offerte da J2EE
- Slide 74
  - se non viene usato un form, allora ogni richiesta di pagina HTML è un GET
  - pagina generata dinamicamente: nelle CGI la pagina HTML è l'output di un programma, qui (semplificando, lo si vedrà meglio in seguito) è l'output di una funzione di una classe Java
  - questo vuol dire scrivere una funzione piena di `println`, in modo che il risultato sia un documento HTML completo o comunque correttamente interpretabile da un browser (nel libro di testo questa soluzione è indicata come "pure Servlet")
  - J2EE offre un'alternativa nel caso in cui si debba modificare una piccola parte di una pagina HTML per lo più statica: Java Server Pages (JSP)
  - in modo simile al PHP e all'ASP, JSP permette di scrivere tranquillamente in HTML, e poi ci sono dei tag speciali al posto dei quali il server esegue un metodo di una servlet, il cui risultato viene messo al posto dei soli tag speciali (esempio a pag. 10 del libro di testo)

- Slide 75
  - servlet container = servlet engine, non è altro che l'ambiente Java che gestisce le varie servlet che vengono definite su un determinato server Web
  - le servlet sono organizzate con un preciso ciclo di vita (definito dal fatto che le servlet, come si vedrà, ereditano tutte da una ben precisa classe del J2EE)
  - il programmatore di servlet può modificare solo alcune parti ben definite di questo ciclo di vita (ridefinendo alcuni metodi di alcune classi)
  - questo ciclo di vita "esterno", sul quale il programmatore di servlet non ha controllo, è implementato in modo tale che, alla prima GET o POST di una servlet, viene caricato in memoria un nuovo processo persistente
  - creato tale processo, ogni invocazione della servlet (compresa la prima) sono solo thread di questo processo
  - quando un'invocazione della servlet viene servita completamente, "muore" solo il thread, ma non il processo
  - ad esempio, si può far sì che, se occorre comunicare con un database, si apra la connessione una volta sola e la si sfrutti per tutte le richieste successive
  - dato che i thread sono più leggeri dei processi (non hanno la parte di memoria, che è condivisa, ma solo lo stack e il program counter), questo risulta in una modalità di esecuzione più efficiente del CGI
  - c'è anche un retro della medaglia, non menzionato in queste slides e detto tra le righe del libro di testo: se ci sono tante servlet che vengono invocate, restano tutte in memoria (non muoiono mai) e la memoria stessa può finire!
  - questo problema può essere risolto in due modi:
    - l'amministratore del Web server si accorge del degrado di prestazione e uccide le servlet meno importanti (per lui...)
    - J2EE si accorge che una servlet è idle (ovvero, in memoria ma senza richieste) da troppo tempo, e allora la uccide
    - garbage collection di J2EE: vengono opportunamente selezionate alcune servlet, che vengono uccise
  - in tutti i casi di cui sopra, l'utente può definire un metodo da eseguire prima della rimozione di una servlet
- Slide 77: architettura J2EE per le servlet
  - si potrebbe anche ereditare direttamente `GenericServlet`, ma `HttpServlet` è già pronta per HTTP

- Slide 79: javax è il prefisso dei package di J2EE
- Slide 80:
  - in realtà, è sempre il servlet container a chiamare i metodi scritti qui (o direttamente, con `init` e `service`, o indirettamente con `destroy`, che viene invocato automaticamente quando il container distrugge una servlet)
  - una volta sola = non per ogni invocazione della servlet (dopo `destroy` ci sarà nuovamente la `init`)
  - `init` e `destroy` le scrive il programmatore web (ovviamente, la signature è fissata nell'interfaccia: si può scrivere solo il corpo delle funzioni)
  - `service` va scritta solo se si eredita da `GenericServlet`, altrimenti, se si eredita da `HttpServlet`, viene lasciata così com'è, e si scrive almeno uno dei due metodi chiamati da `service`: `doGet` e `doPost`
- Slide 83: in realtà il servlet engine non chiama la `doXxx`, bensì la `service`, che poi smista al corretto metodo (a meno che non venga ridefinita...)
- Slide 88
  - `getParameterNames` va bene anche per le GET, non solo per le POST
  - `getParameterValues`: ad esempio per i checkbox, o per i casi in cui più campi di un form vengono chiamati con lo stesso nome (vedere es. pagina 73 del libro)
  - `getCookies` e `getSession`: diversi modi di avere lo "stato" sulle pagine HTML; questo argomento verrà ripreso più avanti
- Slide 89: in realtà, ai metodi `doXxx` l'oggetto `HttpServletResponse` arriva tramite il metodo `service`, e non viceversa
- Slide 90
  - `getOutputStream`: ad esempio, la pagina HTML potrebbe essere compressa con gzip se è troppo grande; oppure potrebbe consistere in una singola immagine
  - non c'è la gestione delle sessioni, perché si fa tutta agendo sulla classe ritornata da `getSession`
  - `getWriter`: per mandare l'HTML "normale"
- Slide 93-94: in questo corso verrà usato TomCat (letteralm.: gatto maschio)
  - inizialmente era stato sviluppato all'interno del progetto Jakarta

  - Jakarta è stato sospeso nel 2011, e TomCat è ora distribuito da Apache, che era il promotore di Jakarta
  - esistono molte altre soluzioni per avere servlet e JSP, addirittura Apache ne fornisce due

- Slides 94-95: vale per Windows; per Linux Ubuntu dovrebbero bastare i seguenti comandi:

```
sudo aptitude install apache2
sudo aptitude install default-jdk
sudo aptitude install tomcat7
```

  - il primo installa il Web server "normale" (HTTP con pagine statiche)
  - il secondo installa Java con il compilatore (se non già presente)
  - il terzo aggiunge ad Apache la gestione di servlet e JSP
  - è possibile soffrire e compilare tutto da sorgenti, vedere ad esempio qui: `http://www.sitepoint.com/jsp-quick-start-guide-linux/`
  - alla fine, la pagina di testing è un po' diversa: `http://localhost:8080`
  - la directory dove mettere i files che definiscono una servlet è la seguente: `/var/lib/tomcat7/webapps/`
  - si potrebbe anche configurare per metterla altrove, ma è più comodo lasciarla lì e cambiargli l'owner:

```
cd /var/lib/tomcat7/
sudo chown -R vostronomeutente webapps
```

  - da qui si può creare una struttura di directory del tipo quella delineata nelle slides 101 e 108
  - la "directory radice" di cui si parla alla slide 101 è una sottodirectory di `webapps`, da creare
  - se ad esempio si crea una directory `vediamo` con dentro un file `esempio.html`, allora basterà aprire un browser su `localhost:8080/vediamo/esempio.html` per vedere il risultato
  - se il file si chiama `index.html`, basta andare su `localhost:8080/vediamo/`
  - se la directory si chiama `ROOT`, basta scrivere `localhost:8080/` (infatti, la pagina di default di TomCat è proprio lì...)
  - la directory `ROOT` esiste già