Universita' degli Studi di Bologna
Facolta' di Ingegneria

Anno Accademico 2008-2009

Laboratorio di Tecnologie Web

Pagine JSP

Pattern DAO

http://www-lia.deis.unibo.it/Courses/TecnologieWeb0809

Concetti fondamentali

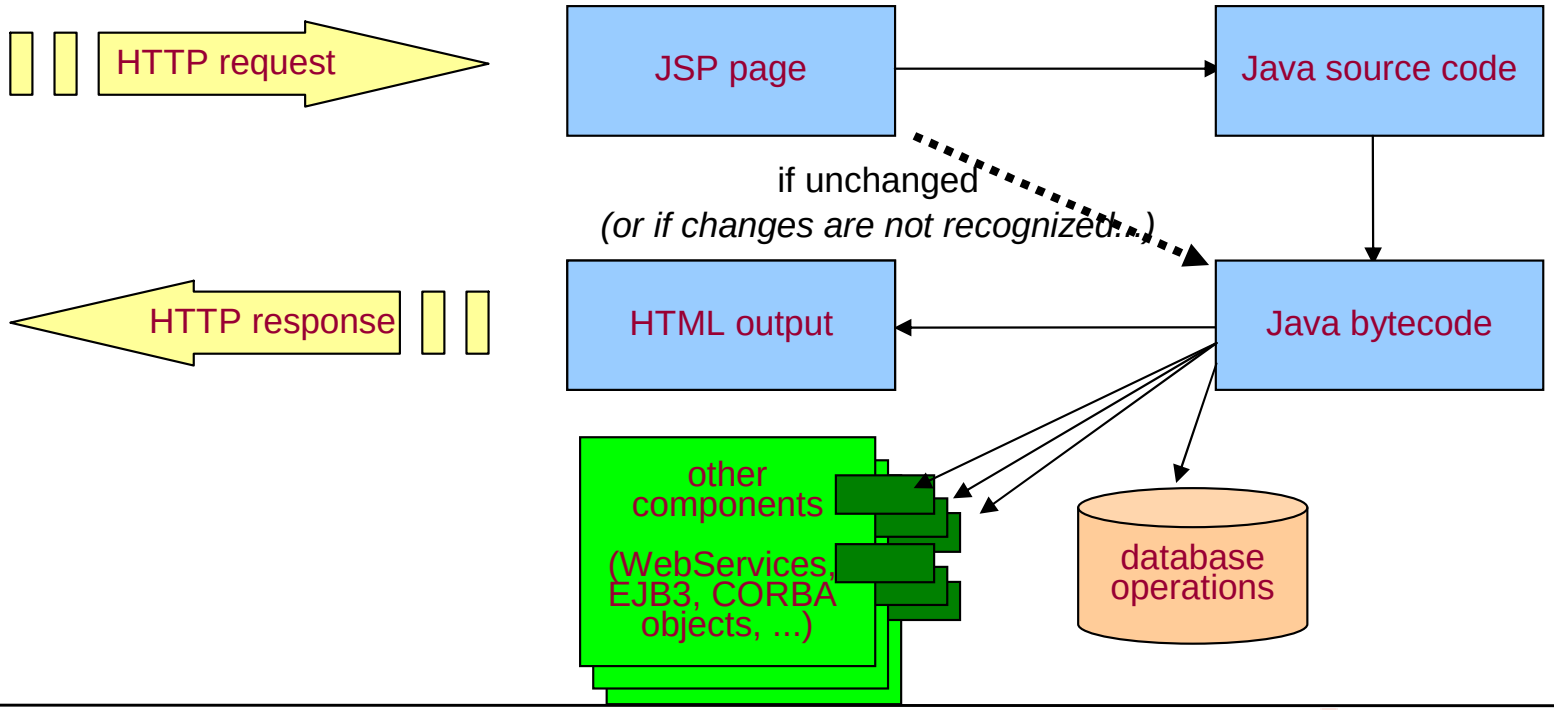
- Pagine “HTML” che includono codice Java
  - elaborate lato server, ma **non interpretate** bensì:
    - **tradotte in Servlet** al volo alla prima richiesta
      - Tomcat salva su file i *.java* che genera automaticamente!
      - Si può curiosare:

```
$TOMCAT_HOME/work/Catalina/localhost/$WEBAPP_NAME/org/apache/jsp/$PATH_TO_PAGE/$PAGE_NAME.java
```

    - **compilate**
      - Tomcat salva su file anche i *.class*
    - **eseguite** come normali Servlet a seguito di ogni richiesta per ottenere:
      - **logica di business** lato server
        - operazioni su sessione, parametri della richiesta, cookie, …
        - modifica di un database, …
        - ecc…
      - **generazione dinamica dell’HTML** restituito al browser
  - l’HTML che giunge sul browser in risposta alle richieste per tali pagine non coincide più (ovviamente) con il loro codice sorgente
    - confrontate i file in Eclipse con il “Visualizza sorgente pagina di firefox”

Funzionamento lato server

- Le pagine JSP vengono gestite su Tomcat da una particolare Servlet
  - `JspServlet`
    - automaticamente “associata” a tutte le richieste per risorse di tipo *.jsp*
    - traduce il loro sorgente testuale (ibrido: HTML + Java) in codice sorgente di una classe Java che estende, attraverso la classe `HttpJspBase`, la classe `HttpServlet`



4 costrutti

- **Direttive**: <%@ … %>
  - definizione di proprietà della pagina, import di classi a cui si fa riferimento, inclusione di librerie di tag aggiuntive, inclusione di altri documenti, ecc…
  - **valutate a tempo di compilazione**
- **Dichiarazioni**: <%! … %>
  - definizione di variabili e metodi utilizzabili nel resto della pagina
  - l'equivalente di variabili membro e metodi non statici per una Servlet
- **Scriptlet**: <% . . . %>
  - codice valutato via via che il server genera la risposta per il client
  - l'equivalente del codice nei metodi *doGet()* e *doPost()* di una Servlet
    - …confrontate il sorgente di una *.jsp* con i *.java* generati da Tomcat!
  - **devono complessivamente costituire del codice Java ben formato**
    - è possibile aprire un blocco ("{") e chiuderlo ("}") in scriptlet diversi, divisi da codice HTML"…
    - …o anche scrivere il codice di un *if* in uno scriptlet, chiudere lo scriptlet, scrivere del codice HTML, quindi aprire un nuovo scriptlet e scrivere in esso il codice dell'*else*
  - **anche il codice HTML che li intervalla deve risultare, alla fine, ben formato**
    - …se ad esempio si apre un tag <*p*> prima dello scriptlet con un *if* e lo si conclude all'interno sia dell' *if* che dell' *else*, ci saranno due </*p*> nel sorgente, ma solo uno arriverà nell'HTML finale restituito al browser!
- **Espressioni** <%= … %>
  - espressioni Java il cui risultato è direttamente inserito nel codice HTML circostante
  - **non istruzioni: non c'è il *";"* alla fine !**

E' il motivo per cui **non si possono dichiarare metodi dentro uno scriptlet**! Sarebbe come definire un metodo all'interno di un altro metodo in una classe Java!

4 costrutti (versione con delimitatori XML)

- Modalità alternativa di dichiarare gli stessi costrutti appena visti
  - seguendo la sintassi XML
  - scrivendo pagine JSP i cui sorgenti sono documenti XML ben formati

- **Direttive**:

```
<jsp:directive.type attribute />
```

- **Dichiarazioni**:

```
<jsp:declaration> methods & members definitions </jsp:declaration>
```

- **Scriptlet**:

```
<jsp:scriptlet> java code </jsp:scriptlet>
```

- **Espressioni**:

```
<jsp:expression> java expression </jsp:expression>
```

6 azioni

- Comandi JSP **valutati a tempo di esecuzione** della richiesta

```
<jsp:nomeComando attributiComando … />
```

- Lista:
  - **useBean**: istanzia un oggetto conforme alle convenzioni JavaBean e lo rende disponibile al codice che segue tramite un preciso identificativo e un preciso scope di validità
  - **getProperty**: ritorna in forma di oggetto la property indicata
  - **setProperty**: imposta il valore della propery indicata
  - **include**: include a request time (non a compile time, come le direttive) il contenuto di un file nel sorgente della JSP valutato dal server
  - **forward**: cede la gestione della richiesta a un'altra risorsa
  - **plugin**: genera il contenuto necessario per scaricare un plug-in Java

8 oggetti built-in

- Sono risorse…
  - …rese automaticamente disponibili dal servlet container
  - …accessibili **semplicemente per nome** all'interno del codice della pagina
  - …equivalenti agli oggetti omologhi di norma accessibili all'interno di una Servlet

- Lista:
  - page (la pagina e le sue proprietà)
  - config (dati di configurazione)
  - out (il print writer su cui scrivere l'HTML della response)
  - request (la richiesta HTTP ricevuta e i suoi attributi, header, cookie, parameteri, ecc…)
  - response (la risposta HTTP e le sue proprietà)
  - application (dati condivisi da tutte le pagine della web application)
  - session (dati specifici della sessione utente corrente)
  - exception (eventuali eccezioni lanciate dal server; utile per pagine di errore)
  - pageContext (dati di contesto per l'esecuzione della pagina)

Approfondimento - Tag library (taglib)

- Funzionalità aggiuntive…
  - …rese disponibili sotto forma di librerie di tag
  - …utilizzabili all'interno della pagina a seguito della direttiva

```
<%@ taglib uri="mnemonico dichiarato dalla tag library"
    prefix="prefisso usato nella pagine per marcare i tag di tale libreria" %>
```

- Esempi: richiamare attraverso la dichiarazione di tag di una tag library…
  - …frammenti di HTML piu' complessi
  - …funzionalità fornite da librerie Java (es: JSTL)
    - ***non ci sarà all'esame***
    - se comunque volete curiosare, Tomcat mostra alcuni esempi
    - è interessante, davvero! ☺
  - …funzionalità fornite da componenti Java personalizzati

Expression language (EL)

- Valutazione di espressioni racchiuse all'interno dei caratteri `${ ... }`

- Accesso efficace alle proprietà dei JavaBean dichiarati
  - accesso ai campi dei bean semplicemente per nome (in maniera analoga a come si accede alle proprietà degli oggetti Javascript)
  - risoluzione trasparente dei corrispondenti metodi getter/setter

```
${ miobean.miocampo }
```

- Risultato sostituito…
  - …direttamente nell'HTML (scrittura del contenuto finale della pagina)
  - …come valore degli attributi dei tag di una taglib (valutazione da parte del processore JSP)

```
<c:if test="${ miobean.miocampo > 0}">
    ...
</c:if>
```

Esempi di pagine JSP

- Tomcat fornisce out-of-the-box alcuni esempi di come utilizzare le API JSP
  - utili soprattutto come **cheat sheet**
    - dispersive per imparare
    - valide come riferimento veloce per sapere come fare operazioni precise…
  - visionabili a partire da:

    http://localhost:8080/jsp-examples

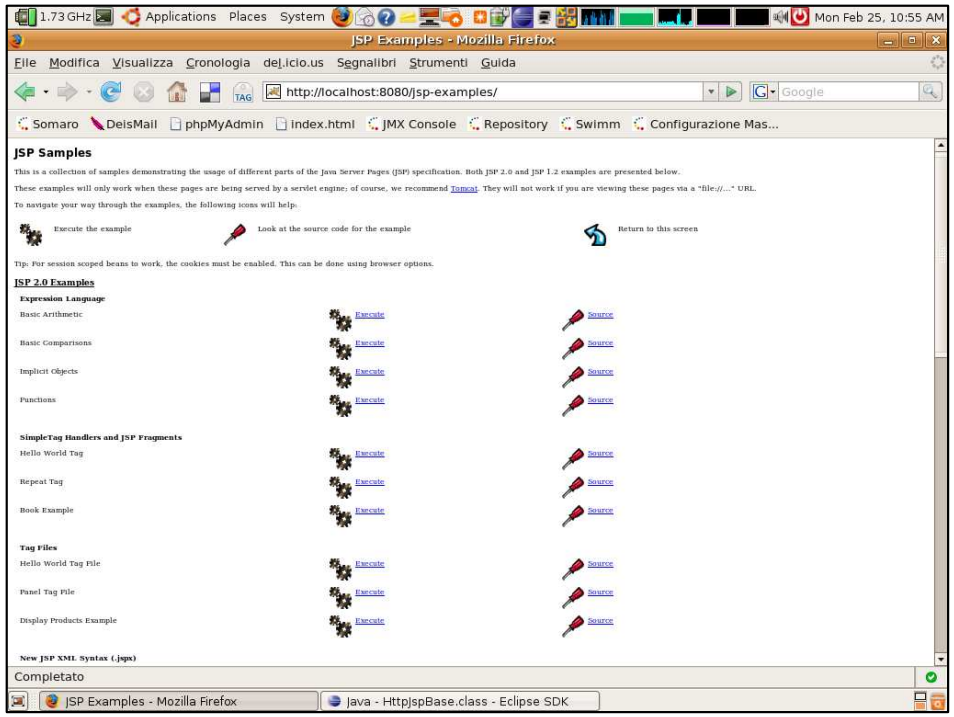
    codice sorgente completo

    disponibile su *file system*,

    nella corrispondente directory

    di deployment

    `$TOMCAT_HOME/webapps/jsp-examples/...`



Progetto di esempio

- Scaricare (dal sito del corso)…
  - …importare (come *'existing project'* in Eclipse)…
    - …configurare (modificando *environment.properties*)…
      - …deployare (lanciando il target di ANT *'deploy.as.XXX'*)…
        - …il progetto di esempio *TemplateJSPandDAO.zip*
- Funziona tutto?

index.jsp

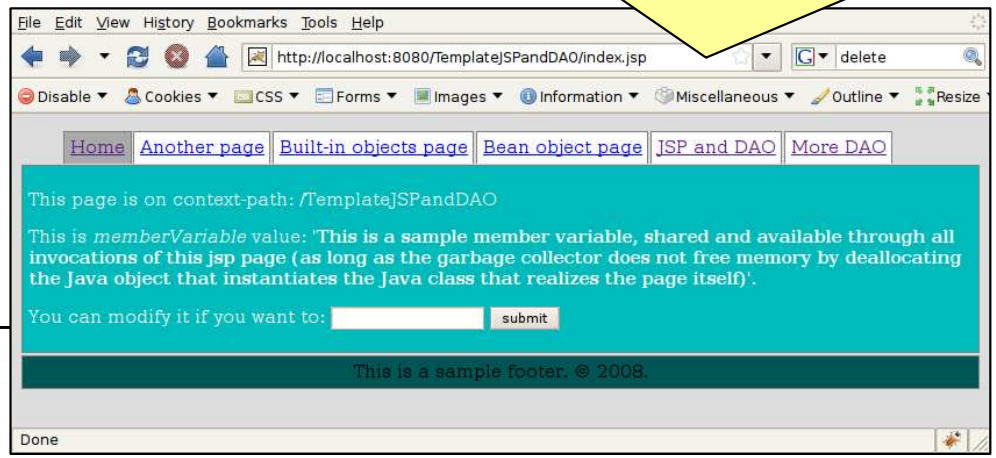
- **direttive:**
  - inclusione nella pagina del contenuto di altri documenti testuali:
    - menu
    - footer
  - affidamento della gestione di eventuali eccezioni a una pagina specifica
- **dichiarazioni**
  - metodi di utilità
  - variabili della pagina
- **scriptlet**
  - generazione di contenuto via invocazione dei metodi di cui sopra

- …frizzi e lazzi:
  - al caricamento dell pagina ricerco la linguetta del menu' che la contraddistingue e le cambio il colore di sfondo, in modo da evidenziare la pagina corrente

• settate il valore nella form
• navigate altrove e tornate qui
• aprite la pagina con un altro browser

IL VALORE NON SI RESETTA !!

• è definito all'interno di una dichiarazione (come una variabile membro della Servlet che corrisponde a questa pagina JSP)
• se non interviene il garbage collector per liberare memoria sul server, l'oggetto Java di tipo JSP/Servlet istanziato per servire la prima richiesta servirà anche le successive (non vengono istanziate sue repliche)
• se invece serve liberare memoria, il server rimuove le istanze di oggetti non utilizzate (ergo **non è questo il sistema per condividere informazioni tra più utenti**! Dichiarate piuttosto dei *jsp:bean* con scope di applicazione)



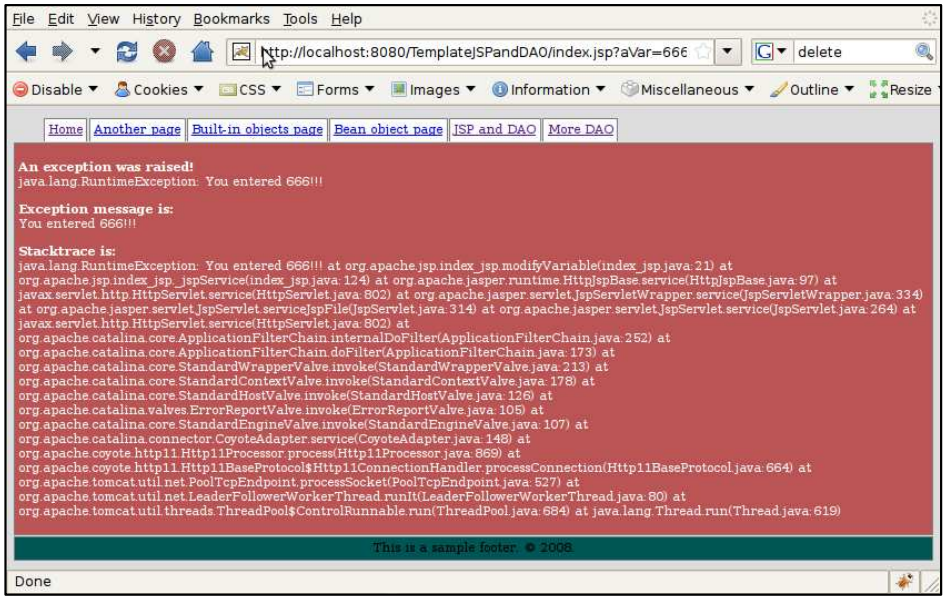
failure.jsp

- *Easter egg*: se viene inserito il valore '*666*' nella *form* della pagine *index.jsp*, la successiva submission dei dati della form causerà il lancio di una eccezione

↓

Tomcat rileva l'eccezione e, come indicato nelle direttive della pagina *index.jsp,* esegue il forward della richiesta (e dell'eccezione) alla pagina *failure.jsp*

- **direttive**
  - dichiarazione che trattasi di pagina di errore
  - si ottiene in questo modo accesso all'oggetto *exception* di tipo built-in
  - …ad esempio per stamparne lo *stacktrace* come in questo caso
  - …è invece buona norma gestire le eccezioni in modo da evitare output e comunicazioni "tecniche" all'utente



another.jsp

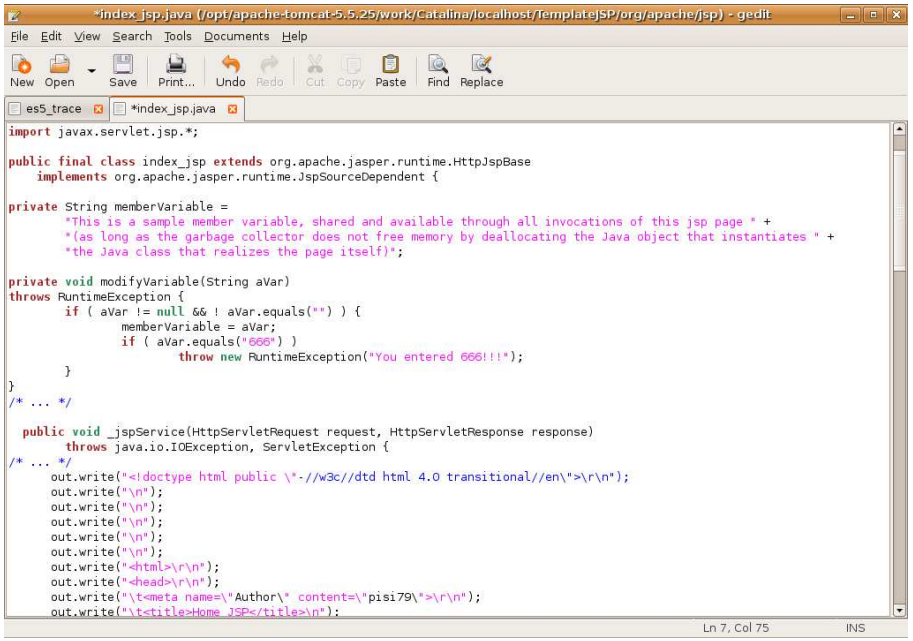
- Questa volta la variabile settata tramite la form è definita all'interno di uno scriptlet, non di una dichiarazione
- L'effetto è quello di avere una variabile in *stack* al momento dell'esecuzione del metodo *service()* della Servlet derivata dalla JSP:
  - cancellato al termine di ogni esecuzione
  - richieste successive verso la stessa pagina non ritrovano il valore precedentemente impostato

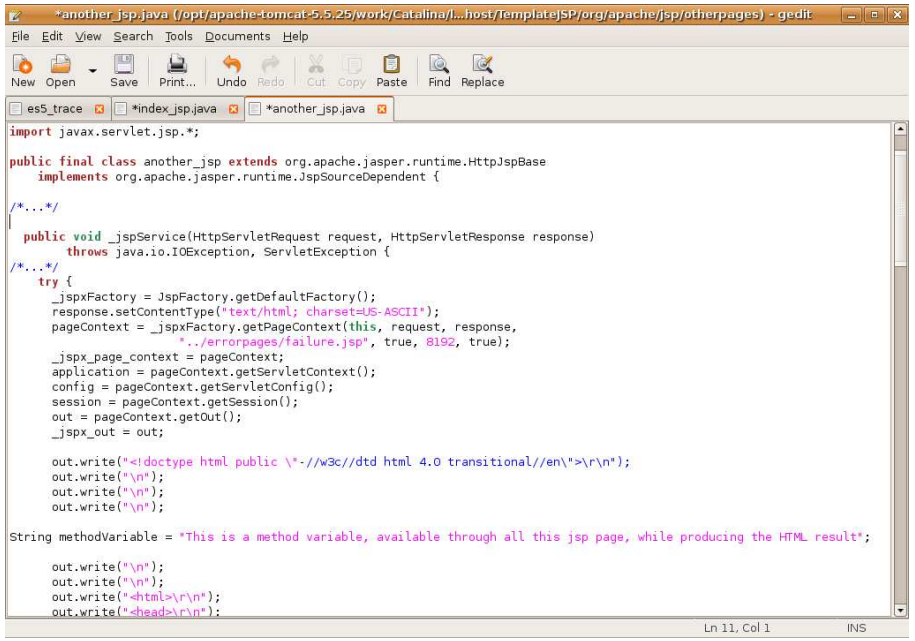
Codice a confronto

- Il codice delle Servlet generate da Tomcat in corrispondenza delle richieste alle pagine *index.jsp* e *another.jsp* è visibile su file system in:

```
$TOMCAT_HOME/work/Catalina/localhost/TemplateJSPandDAO/org/apache/jsp
```

- In caso di dubbi può essere utile aprire i sorgenti e analizzare come vengono tradotti i diversi costrutti JSP



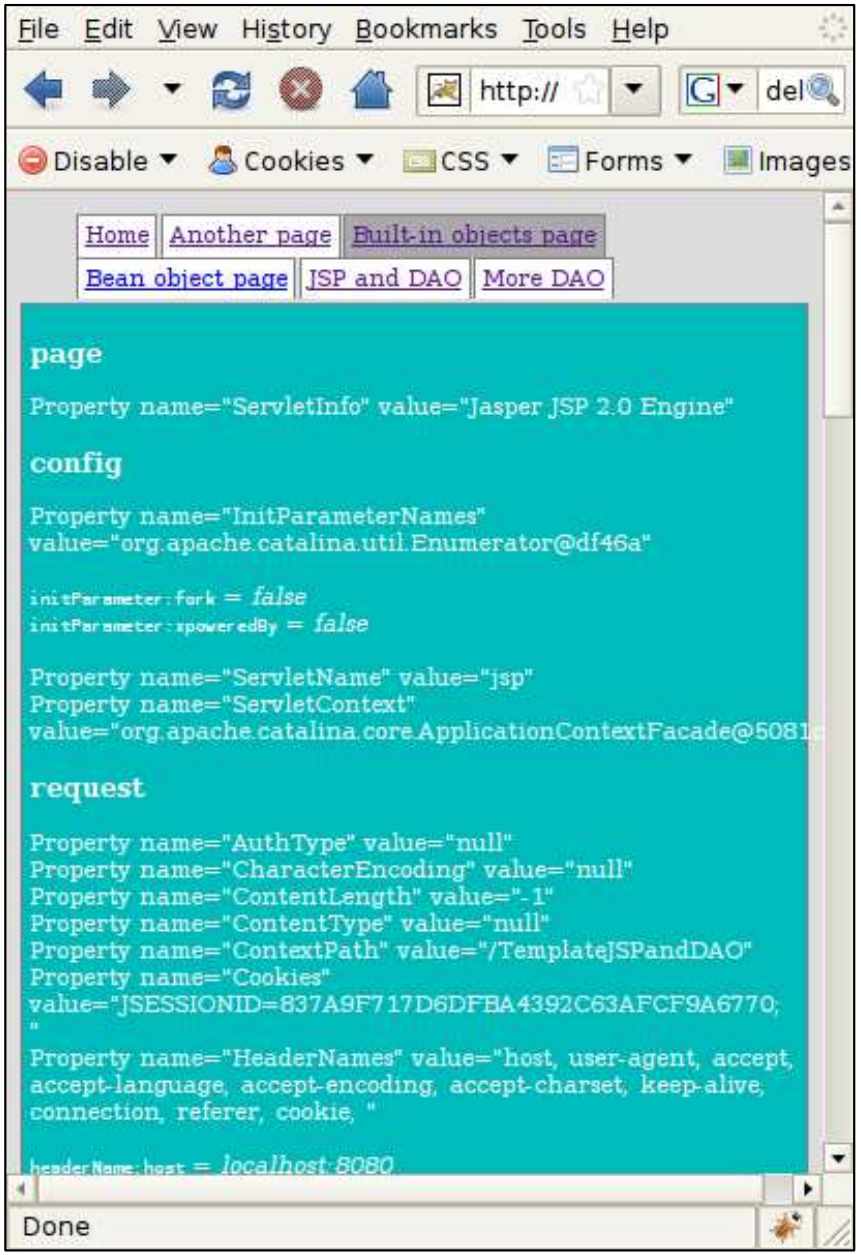


builtin.jsp

- Una lunghisssssssima sfilza di stampe a video dei (piu' significativi) valori…
  - …accessibili attraverso i 6 oggetti *built-in*
  - …dall'interno di *scriptlet* ed *espressioni*

- Utile come riferimento di
  - cosa c'è
  - come vi si accede

- E basta.



bean.jsp

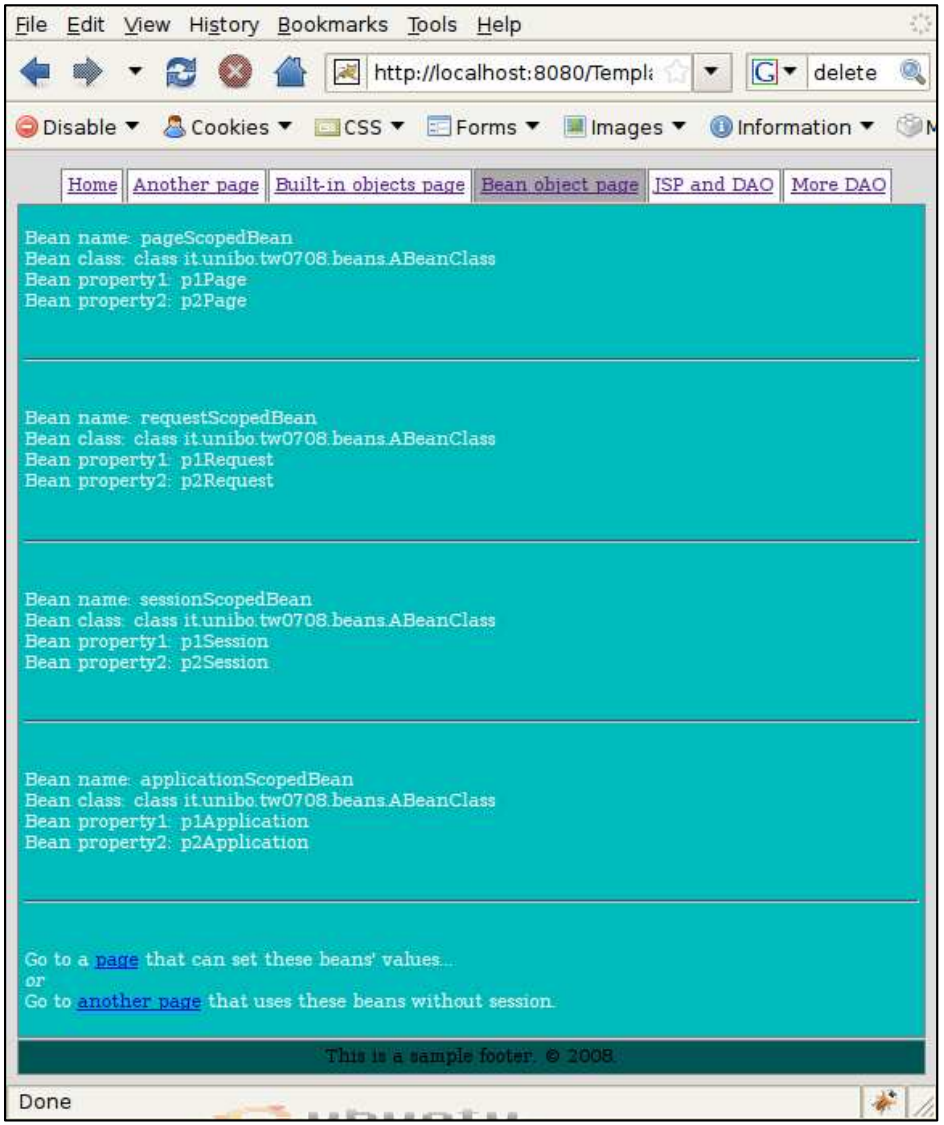
- …e pagine collegate
  - utilizzo di oggetti Java all'interno delle pagine JSP
  - gestione di scope differenti in funzione del tipo di semantica desiderata

page

request

session

application

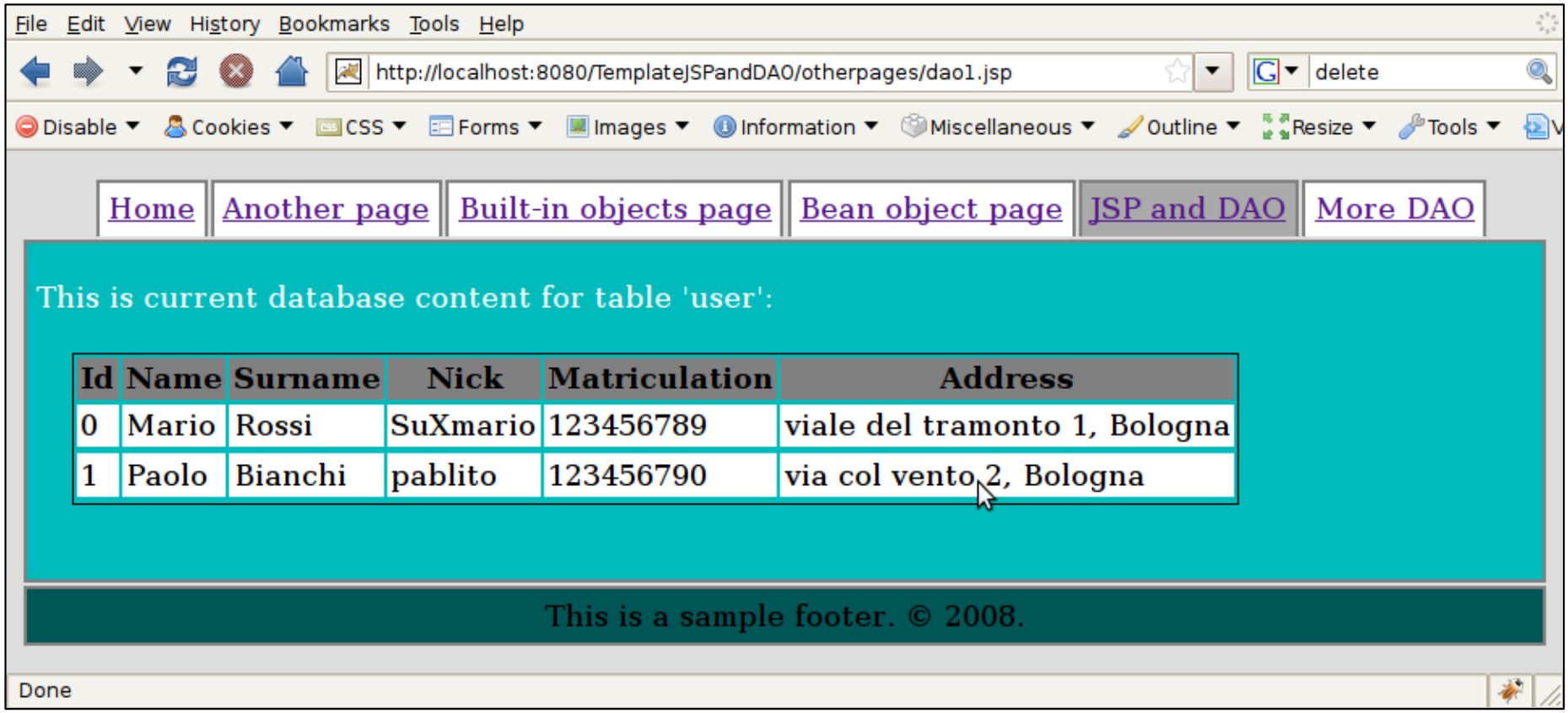


- Provare a navigare seguendo i link per comprendere il funzionamento

- La pagina *bean3.jsp* mostra un po' di EL

dao1.jsp

- Una pagina JSP che legge da database un insieme di valori e li stampa a video in una *table* all'interno dell'HTML finale che produce



- Avete lanciato il database? E' un programma a parte!
  - *…nuovo target di ANT*

Pattern DAO: alla conquista del database

- Un “pattern” è un “modo di fare le cose” in modo efficace

- Il pattern “DAO” rappresenta “***IL***” modo di separare tra loro:
  - logica di business (es: Servlet, pagine JSP, …)
  - logica di persistenza (es: scritture su DB, letture, …)

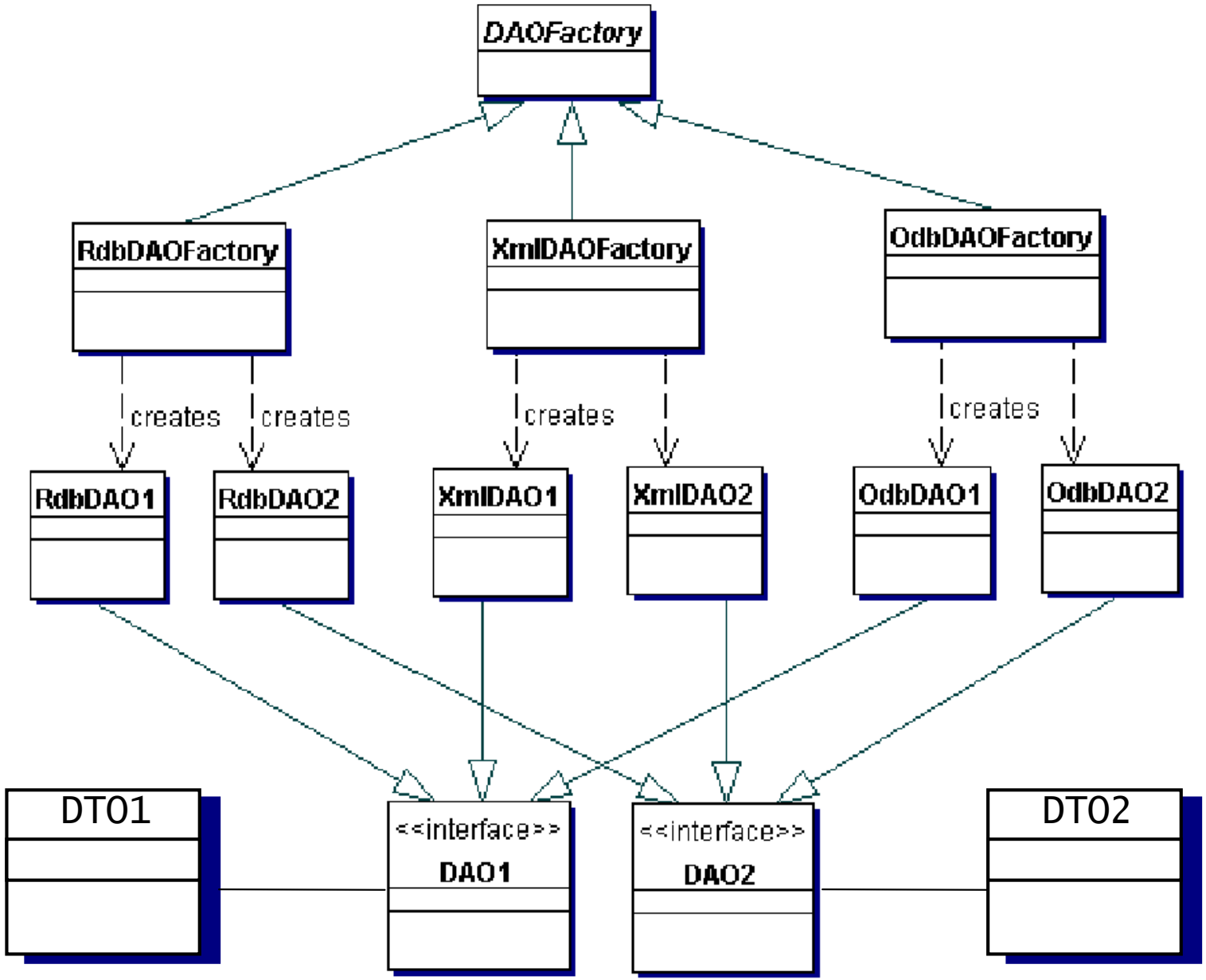
- I componenti della logica di business NON DOVREBBERO MAI CONTENERE CODICE CHE ACCEDE DIRETTAMENTE AL DATABASE
  - scarsa manutenibilità
  - sovrapposizione di responsabilità

- Solo gli oggetti “DAO” hanno il permesso di “vedere” il DB
  - espongono metodi di accesso adatti per tutti gli altri componenti

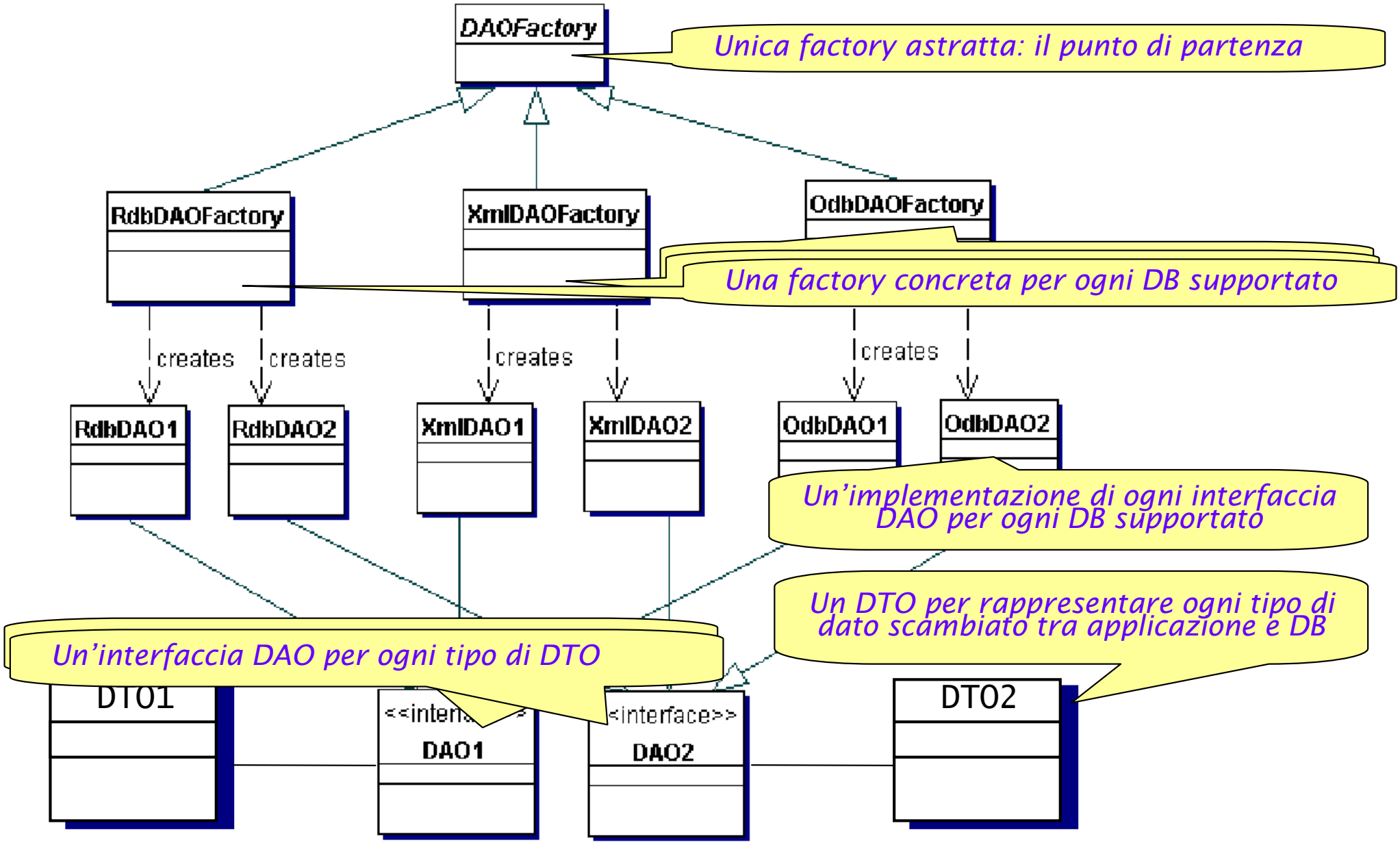
Pattern DAO: principi fondamentali

- I valori scambiati tra DB e resto dell'applicazione (attraverso gli oggetti "DAO") sono racchiusi in oggetti detti ***Data Transfer Object (DTO)***:
  - campi privati per contenere i dati da leggere/scrivere su db
  - metodi getter e setter per accedere dall'esterno a tali campi
  - metodi di utilità (confronto, stampa, calcolo dell'hashcode, …)
- Le operazioni che coinvolgono tali oggetti sono raggruppati in interfacce che definiscono i ***Data Access Object (DAO)*** disponibili
  - metodi *Create, Read, Update, Delete (CRUD)*
  - altri metodi (tipicamente di lettura con parametri custom)
- **Diverse implementazioni** di tali interfacce permettono l'accesso a diversi database
  - anche se si fa uso di un solo database, tale separazione migliora comunque la divisione delle responsabilità tra le parti dell'applicazione
  - diventa facile migrare l'applicazione su DB diversi un domani)
- Le implementazioni degli oggetti DAO **non sono istanziate direttamente** dai componenti (facendo ~~*new `QlcDAO()`* per capirci~~), ma:
  - sono ottenute attraverso metodi *factory* *(come faremo noi)*
  - sono settate nei componenti da qualche altra entità (es: il container)

The DAO pattern: UML schema

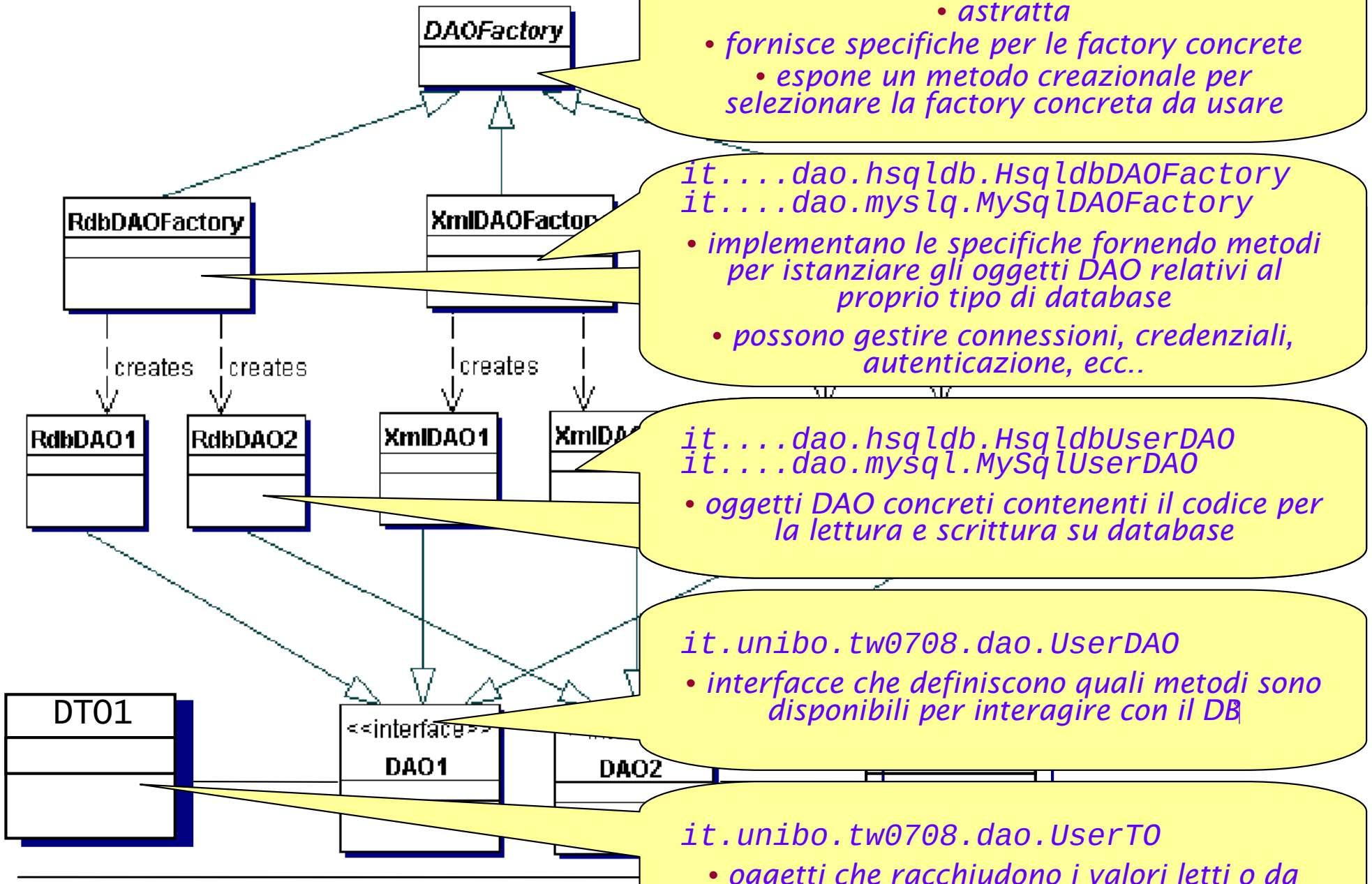


The DAO pattern: UML schema



In genere si ha corrispondenza 1:1 tra i DTO gestiti dalle interfacce DAO e le tabelle su DB, ma non è strettamente necessaria!

The DAO pattern: UML schema



it.unibo.tw0708.dao.DAOFactory

- *astratta*
- *fornisce specifiche per le factory concrete*
- *espone un metodo creazionale per selezionare la factory concreta da usare*

it....dao.hsqldb.HsqldbDAOFactory
it....dao.myslq.MySqlDAOFactory

- *implementano le specifiche fornendo metodi per istanziare gli oggetti DAO relativi al proprio tipo di database*
- *possono gestire connessioni, credenziali, autenticazione, ecc..*

it....dao.hsqldb.HsqldbUserDAO
it....dao.mysql.MySqlUserDAO

- *oggetti DAO concreti contenenti il codice per la lettura e scrittura su database*

it.unibo.tw0708.dao.UserDAO

- *interfacce che definiscono quali metodi sono disponibili per interagire con il DB*

it.unibo.tw0708.dao.UserTO

- *oggetti che racchiudono i valori letti o da scrivere su DB*

JDBC

- Resta ancora da capire come scrivere il codice di questi DAO
- La risposta (per noi) è JDBC
  - Java fornisce una API standard per formulare statement SQL verso database relazionali
    - *SELECT * FROM …. WHERE …., UPDATE …. SET … WHERE …, ecc…*
    - Semplicemente si passano tali stringhe a opportuni metodi che le cui implementazioni sono in grado di sottoporle al database
      - query parametriche
      - lettura dei risultati
      - gestione delle connessioni

Uno sguardo al codice vale più di mille parole!

  - ogni database fornisce un "driver" che implementa tale API, sotto forma di una libreria *.jar* da includere nel *path* dell'applicazione
    - *hsqldb.jar* per HSQLDB
    - *mysql-connector-xxx.jar* per *MySql*
    - ecc…

***Nota**: nel caso di HSQLDB, il* jar *contiene sia il driver JDBC che il programma database!*
- *il driver serve nella webapp, quindi il file sta in WEB-INF/lib*
- *il db lo lanciamo da ANT, quindi il* jar *fa parte del* classpath *definito in* build.xml
- *infine, usiamo il driver anche attraverso 2 classi con* main() *lanciate via Eclipse: il* jar *è anche nel* buildpath

  - Gli statement SQL tuttavia non sono sempre "portabili": DB differenti parlano diversi dialetti
    - in realtà non su tutto… molte query sono standard, ma le ottimizzazioni no!
    - nel progetto di oggi già c'è un esempio di statement "dialettale"
    - …ed ecco perché il pattern DAO

JDBC: regole auree (@see HsqldbUserDAO.java)

- Il consiglio è mio personale, ma questo modo di procedere nello scrivere il codice degli oggetti DAO basati su JDBC è… a prova di bomba!
  - Ogni metodo…
    1. *dichiari una variabile dove collocare il proprio risultato*
    2. *controlli la bontà dei parametri attuali ricevuti*
    3. *apra la connessione al DB*
    4. *formuli gli statement SQL che lo riguardano e imposti il risultato*
    5. *preveda di gestire le eventuali eccezioni*
    6. *rilasci SEMPRE E IN OGNI CASO la connessione in uso*
    7. *restituisca il risultato (eventualmente di fallimento)*
  - E per quanto riguarda gli **statement SQL** veri e propri
    1. *crei (se senza parametri) o prepari (se con parametri) lo statement da proporre al DB*
    2. *pulisca e imposti i parametri (se ve ne sono, ovviamente)*
    3. *esegua l'azione sul DB ed estragga il risultato (se atteso)*
    4. *cicli sul risultato (se presente) per accedere a ogni sua tupla e impostare il proprio risultato con i valori in essa contenuti*
    5. *rilasci la struttura dati del risultato stesso*
    6. *rilasci la struttura dati dello statement*

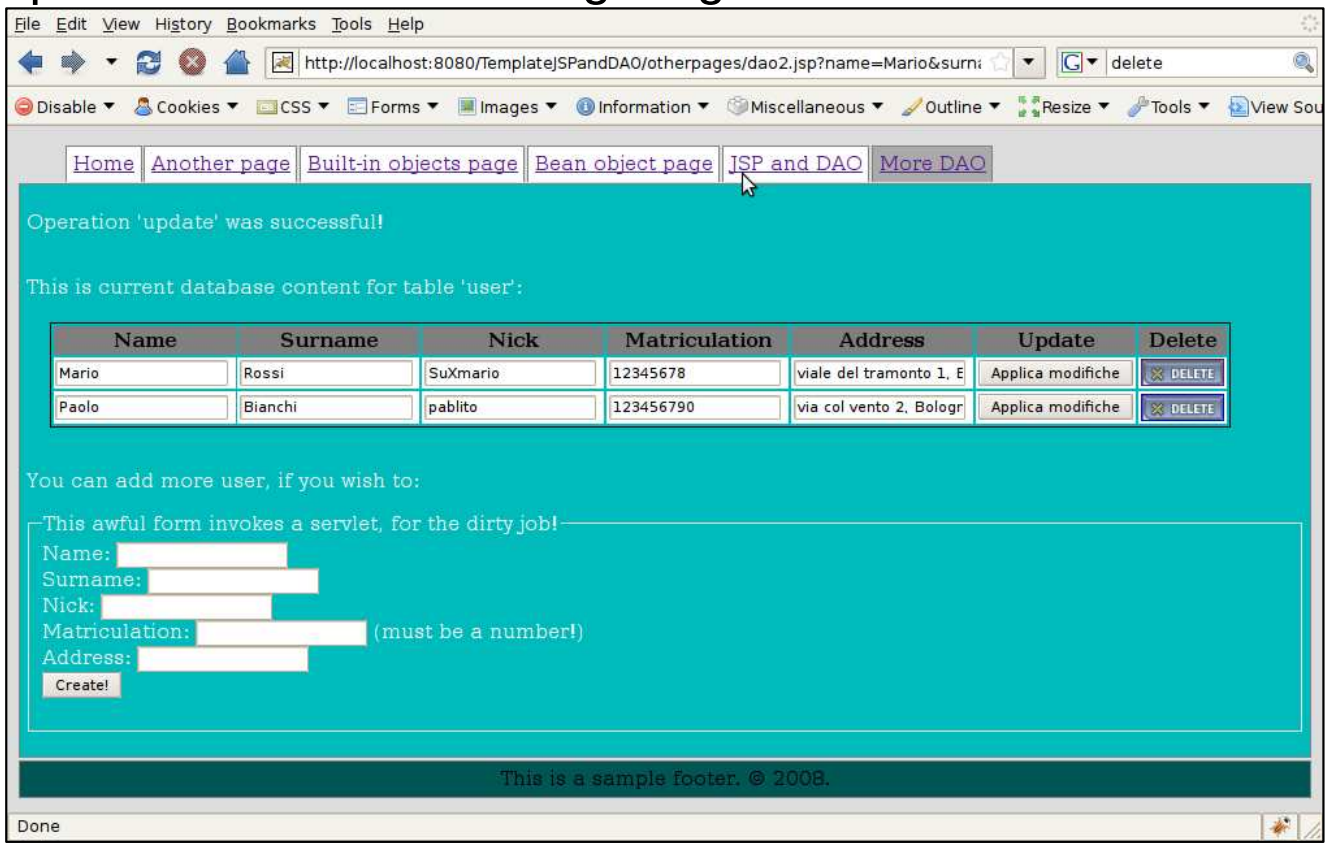
Anche in questo caso, uno sguardo al codice – COMMENTATISSIMO! – vale più di mille parole!

dao1.jsp

- Dichiarazioni:
  - metodi di utilità per la scrittura dell'HTML finale
- Scriptlet:
  - lettura del tipo di factory DAO da utilizzare a partire dal ServletContext
    - oggetto builtin di nome application,
    - settato qui in maniera dichiarativa via *web.xml*
  - istanziazione di un oggetto DAO per accedere al DB degli utenti
  - invocazione di un metodo di lettura
  - stampa a video del risultato

dao2.jsp

- Anziché scrivere del testo nelle celle della tabella…
  - una form per ogni riga
  - un input field per ogni cella
  - un campo nascosto per veicolare l'id
  - un campo di tipo submit per lanciare la form di ogni riga
  - …e inoltre, per ogni riga, un anchor che cabla nell' URL del proprio *href* l'id della riga in cui si trova e ordina un'operazione diversa rispetto al pulsante della form



Non solo pagine JSP

- Il pattern DAO non è limitato all'uso da parte di pagine JSP
  - disponibile anche per le Servlet
  - disponibile persino da Eclipse (due classi da lanciare col tasto destro del mouse, rispettivamente per inizializzare e ripulire le tabelle su DB)
  - ecc…

- La pagina *dao2.jsp* contiene anche una form la cui *action* chiama in causa la servlet *UpdateServlet.java*
  - lettura del tipo di factory DAO da usare dal *ServletContext*
  - lettura dei parametri della richiesta
  - esecuzione della logica desiderata
  - ….anziché produrre un HTML in risposta, attraverso il *RequestDespatcher* viene '*forwardata*' la gestione della richiesta nuovamente alla pagina *dao2.jsp*