Anno Accademico 2007-2008

Laboratorio di Tecnologie Web

Esempio di progetto

http://www-lia.deis.unibo.it/Courses/TecnologieWeb0708/

Template project (refactored)

> A sample project that:
  - runs on Tomcat and exploits HSQLDB
  - leverages the JSF framework
  - performs database access via DAO objects
  - integrates UI components from third-party vendors

> You can, as usual:
  - download *TemplateProjectRefactored.zip* file from the course web site
  - import it in Eclipse
  - set up '*J2EE 1.4 API*' user library (by including Tomcat libraries), code compliance (must be 5.0 or higher) and installed JRE (found a JDK5 on your machine and select it)
  - modify *ant/environment.properties* accordingly to your machine settings (or replace this file with a working copy from a previous project)

In the beginning...

> Let's start analyzing project from the web deployment descriptor (*web.xml*) file:

- `javax.faces.webapp.FacesServlet` servlet mapping
- additional faces config files (to separate things)
- servlet context listener declaration *(...this should solve problems under Win32)*
- welcome page list
- extension filter mapping (needed by the extra features from Apache Tomahawk that this project exploits to let you upload files)

*...from the **Sun's J2EE tutorial**:*

"A **filter** is an object that can transform the header or content or both of a request or response.

Filters differ from Web components in that they usually do not themselves create a response.

Instead, a filter provides functionality that can be "attached" to any kind of Web resource. As a consequence, a filter should not have any dependencies on a Web resource for which it is acting as a filter, so that it can be composable with more than one type of Web resource."

JSF and DAO features

> *faces-config.xml* file:
  - just one default locale: *it*
  - just one JSF bean: *manager*

> *path-config.xml* file:
  - simple navigation rules to directly access pages by basing on outcomes, independently of the view-id the user comes from

> DAO pattern to...
  - provide an abstract factory for the database type selection: `it.tecnologieweb.pizza.web.dao.DAOFactory`
  - provide concrete factory implementations (only one available: HSQLDB): `it.tecnologieweb.pizza.web.dao.hsqldb.HsqldbDAOFactory`
  - provide DAO interfaces and implementations to access database tables: `it.tecnologieweb.pizza.web.dao.ProfessoreDAO` and `CorsoDAO` `it....dao.hsqldb.HsqldbProfessoreDAO` and `HsqldbCorsoDAO`
  - provide transfer objects to exchange arguments/results with these DAOs `it.tecnologieweb.pizza.web.dao.ProfessoreTO` and `CorsoTO`

Let's have it work

> **Web server** commands:
  - start up tomcat via the prompt: *$TOMCAT_HOME/bin/startup.sh|bat*

    or
  - launch tomcat via ANT: *tomcat.launch* in *ant/tomcat-build.xml* file

    (provided manager credentials in *$TOMCAT_HOME/conf/tomcat-users.xml* match those in *ant/environment.properties*)

> **Database** commands:
  - launch HSQLDB via ANT: *launch.database* in *ant/build.xml* file
  - initialize tables via ANT: *init.database* in *ant/build.xml* file

> **Web application** commands:
  - ANT-assisted file copying: *deploy.as.dir|war* in *ant/build.xml* file

    or
  - automatic undeployment (in case), rebuild, packaging, deployment at once via ANT: *webapp.redeploy* in *ant/tomcat-build.xml* file

> Act as the final user:
  - browse to *http://localhost:8080/TecnologieWeb*

Entering the web application

> *home.jsp*:

- a JSP-assisted redirect, similar in concept to that of *index.html*'s REFRESH *http-equiv* in previous JSF projects (but displays no *'wait....'* message!)

> *index.jsp*:

- the actual homepage
- embedded menu tabs
  - in form of a JSF fragment (*menu.jsp*)
  - see how difficult ensuring fragment consistency can be...
  - some graphics
  - page layout via an external stylesheet (*tabs.css*)
    - div positioning
    - colors and background colors
    - tab links highlighting on *mouseOver* event

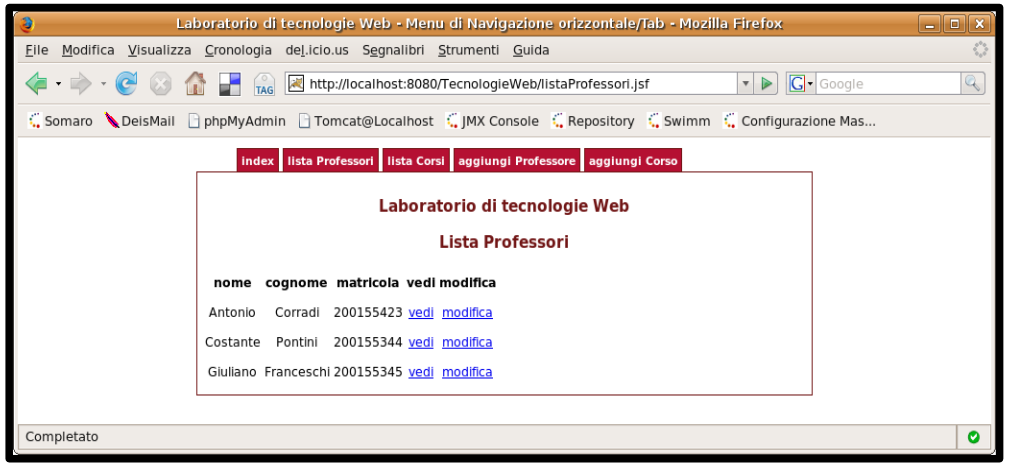
listaProfessori.jsp

> A page holding a simple *<h:dataTable>* element that iterates over a list of `ProfessoreTO` objects, directly fetched from the database by the *manager* bean

- **What is binding?**

<h:dataTable value="#{Manager.professori}" var="prof"
binding="#{Manager.dataProfessori}" *> ... </h:dataTable>*

- Just another way of **backing UI component data** on the web server memory, **via their corresopnding UI component objects** held as member fields of the page backing bean
  - sometimes UI component objects can provide useful extra features, such as direct access to the selected item in a data table (thanks to parameters that JSF pages can send along with the user requests)



  - that's how 'details' and 'modify' links can work differently from each other, depending on the row they belong to

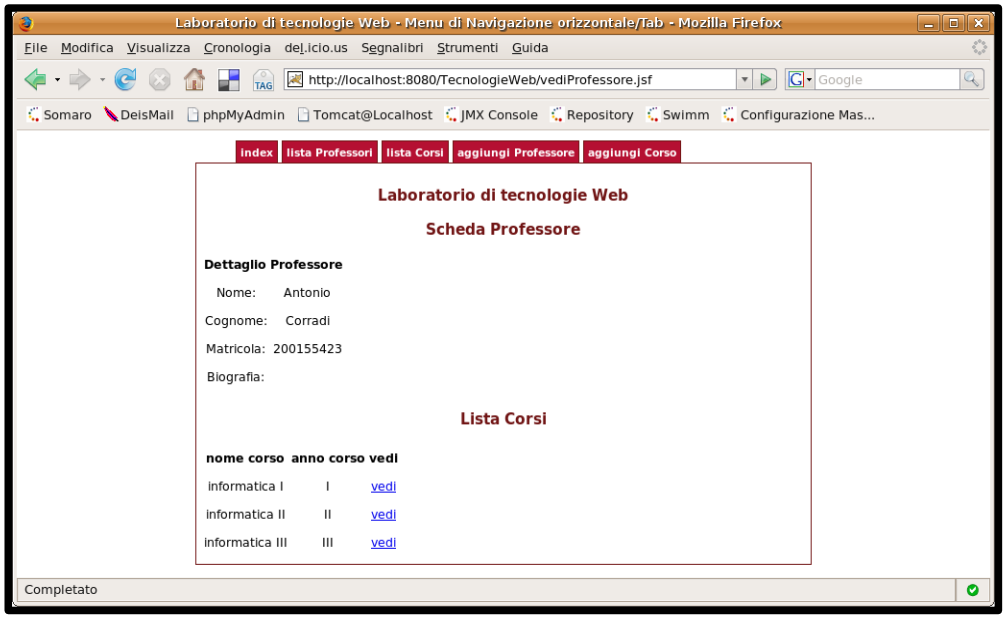
vediProfessore.jsp

> An *<h:panelGrid>* used to to show details on the selected professor

- *<h:outputText>* values just refer to a `ProfessorTO` object which is held by the *manager* bean

> *<h:commandLink>* from the previous view activates a *manager* action that fetches extra professor data from the database (i.e., his/her teachings)

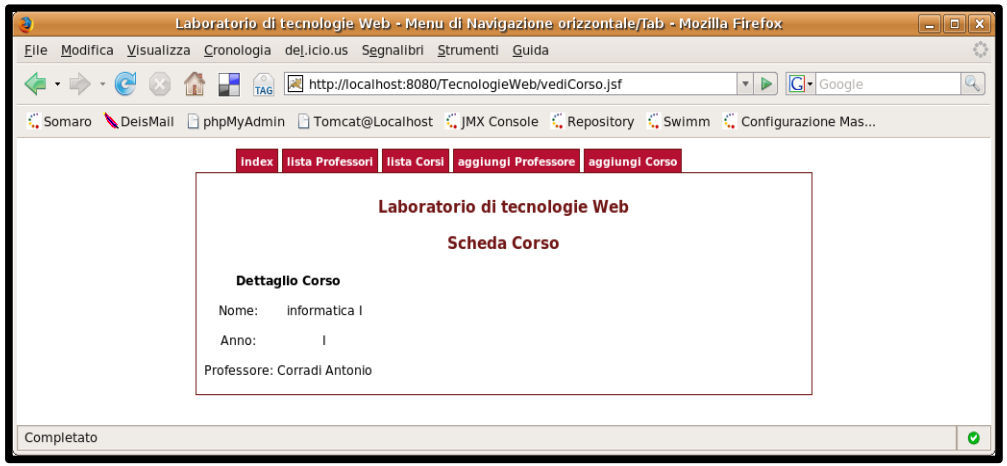
- *listaProfessori.jsp* view and *vediProfessore.jsp* one use two different methods from the `ProfessorDAO` object
- this way, the query that reads the complete professor list can work faster, while the one for retrieving details of a single professor can do the full job (a LEFT JOIN SQL statement between *'professore*' and '*corsi*' tables).
- this time, DAO queries are provided as static public properties of the `HsqldbQuery` class (which is not a DAO pattern requirement).



vediCorso.jsp

> Simply a replica of the *vediProfessore.jsp* view, on a different type of data

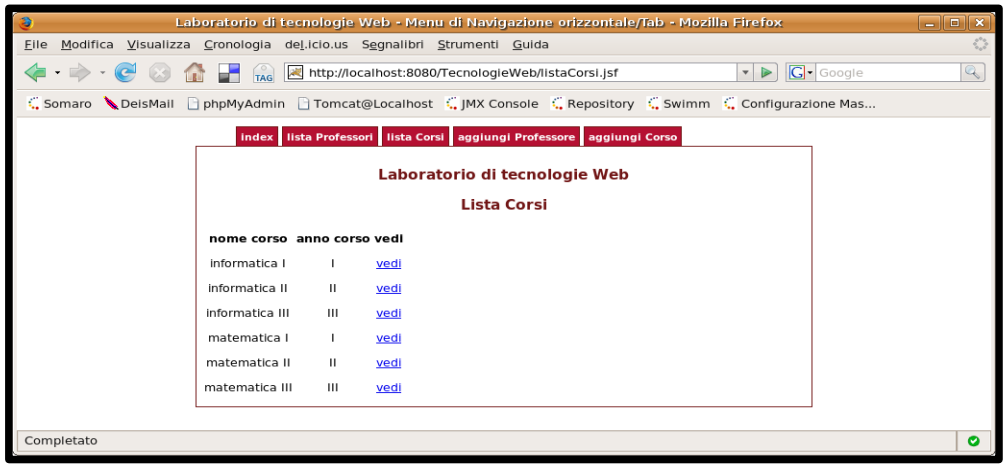
- an *<h:panelGrid>* shows teaching details
- fields are mapped to the *manager.corso* field, of type `CorsoTO`
- *manager* action methods fetches extra data from the database (the related professor features) and then returns the outcome that leads to this page



listaCorsi.jsp

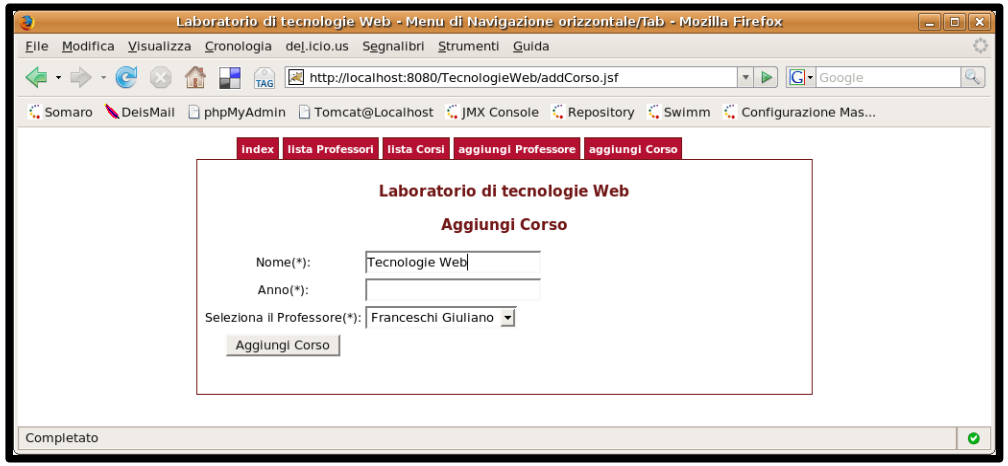
> Simply a replica of the *listaProfessori.jsp* view, on a different type of data

- *<h:dataTable>* iterates over a list of `CorsoTO` objects, fetched from the database by the *manager* bean (through the `CorsoDAO` object) and then binds itself to an UI component object which is a member field of the bean itself



addCorso.jsp

> A renewed instance of the `CorsoTO` object in the *manager* bean is used to back values from the *<h:inputText>* fields



- the *manager* bean action/navigation method that directs to this view also performs the operations needed to reset *manager* internal `CorsoTO` data, then returns the correct outcome string
- how can the manager get involved?
  - differently than in the *CdCollectionManager* webapp, this time **menu tab links do not target faces pages / outcome strings directly, but triggers actions from manager bean methods**, via *<h:commandLink>*s! → this give us the chance to perform additional logic before displaying the view

> See the usage of *<h:message>* errors to notify user of missing required fields

- *for* attribute binds message tag to the component that generates its messages
- we need to wrap both tags into an *<h:panelGroup>* not to scrumble grid cells in case messages appear

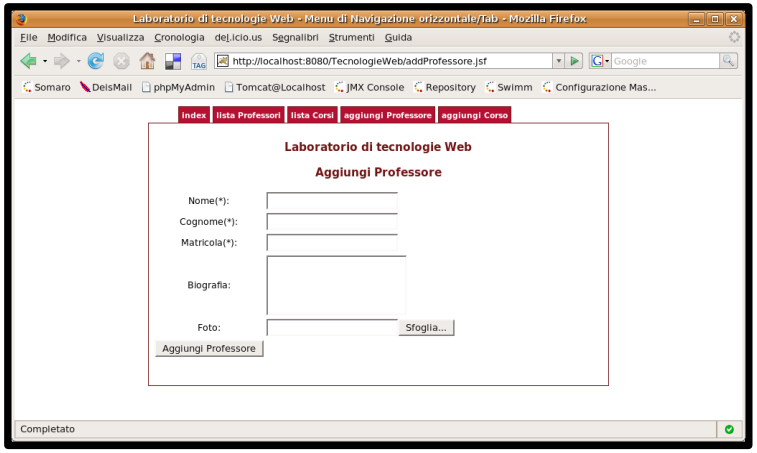
addProfessore.jsp and modProfessore.jsp

> A renewed instance of the `ProfessorTO` object in the *manager* bean is used to back values from the input fields

- it works the same way it did for the *addCorso.jsp* view

> We leverage the **third-party provided** *Apache Tomahawk*'s **file upload input control** *<t:inputFileUpload>* to let user upload the professor photo

- this UI component automatically places the uploaded content in the object that value-binds to its *value* attribute, provided it is of type `org.apache.myfaces.custom.fileupload.UploadedFile`
- we hold this object as a member field of `ProfessorTO`



- practically speaking, anyway, this is a sort of **transient field** of the transfer object: we do not store it directly to the datbase!
- rather, we extract the single parts we are interested in from this transient field and persistently save them only: photo bytes (a `byte[]`), file name and type
- see *manager*'s `insertProfTO()` and `updateProfTO()` methods to see how it happens

back to the *vediProfessore.jsp* view

> In case database holds the photo, the *vediProfessore.jsp* view manages to show it

- but that's simply **a byte array read from a database tuple**: how can it be **conveyed as an image over an HttpResponse**?
- *<h:graphicImage>* in the page does not directly target its *url* attribute to an image resource somewhere, but to **a jsp page that can produce the desired image as an HTTP response to the browser's HTTP GET request**
  - it's a sort of trick, though it works! browsers request images via HTTP GET requests for their URLs; web servers examine those URLs and return images in response → **we let an image URL correspond to a JSP page that pretends to be the image** (**by leveraging the image `byte[]`** in the `ProfessorTO` object from the manager bean)!

fileupload_showimg.jsp

> retrieves the *manager* JSF session bean

- notice the alternate method to obtain this bean (with respect to `Utilities.getManagedBean(String)` that we used in previous projects):

```
Manager manager = (Manager)session.getAttribute( "Manager");
```

- this is a simpler way to do that, but it works only in jsp pages (where you have the built-in `session` object at hand!)

> fetches foto bytes, type and name from the *manager* bean

```
String contentType = (String)manager.getProfessore().getFotoContentType();
String fileName = (String)manager.getProfessore().getFotoName();
...
byte[] bytes = (byte[])manager.getProfessore().getFotoBytes();
```

> sets up and manages response headers

- *see code*

> **write bytes to the response HTTP stream**

```
response.getOutputStream().write(bytes);
```

(very important: this only works if the response stream hasn't been used yet, until you reach this point → otherwise, generated-HTML and bytes mess up!)

Project is over (by now)

> in the time that's left, we'll try to extend some parts of it

> for instance...

- show program in teaching details
- exploit RowData behaviour to perform additional tasks
  - check row selection within the UI component that binds the dataTable
  - add user and/or professor deletion features
- add message bundles and internationalization support
- trick with missing styles
  - grid, columns, ...
- update only modified resources on the web server
  - where it does works, where it does not
- ...

Extras: show program in teaching details (1/2)

> Checking the DAO elements

- `queryCorsiId` in `HsqldbQuery` retrieves the '*programma*' field: ***OK***
- `getCorso(Integer)` method in `HsqldbCorsoDAO` does not use it: ***TO FIX***

```
corso.setProgramma(set.getString("programma"));
```

- *<h:outputText>*s for *programma* is missing in the *vediCorso.jsp* view: ***TO ADD***

```
<h:outputText value="Programma:" />
<h:outputText value="#{Manager.corso.programma}" />
```

and we need sample data!

- modify `InitHsqldb` class suitably and re-invoke ANT's *build.xml init.database* target

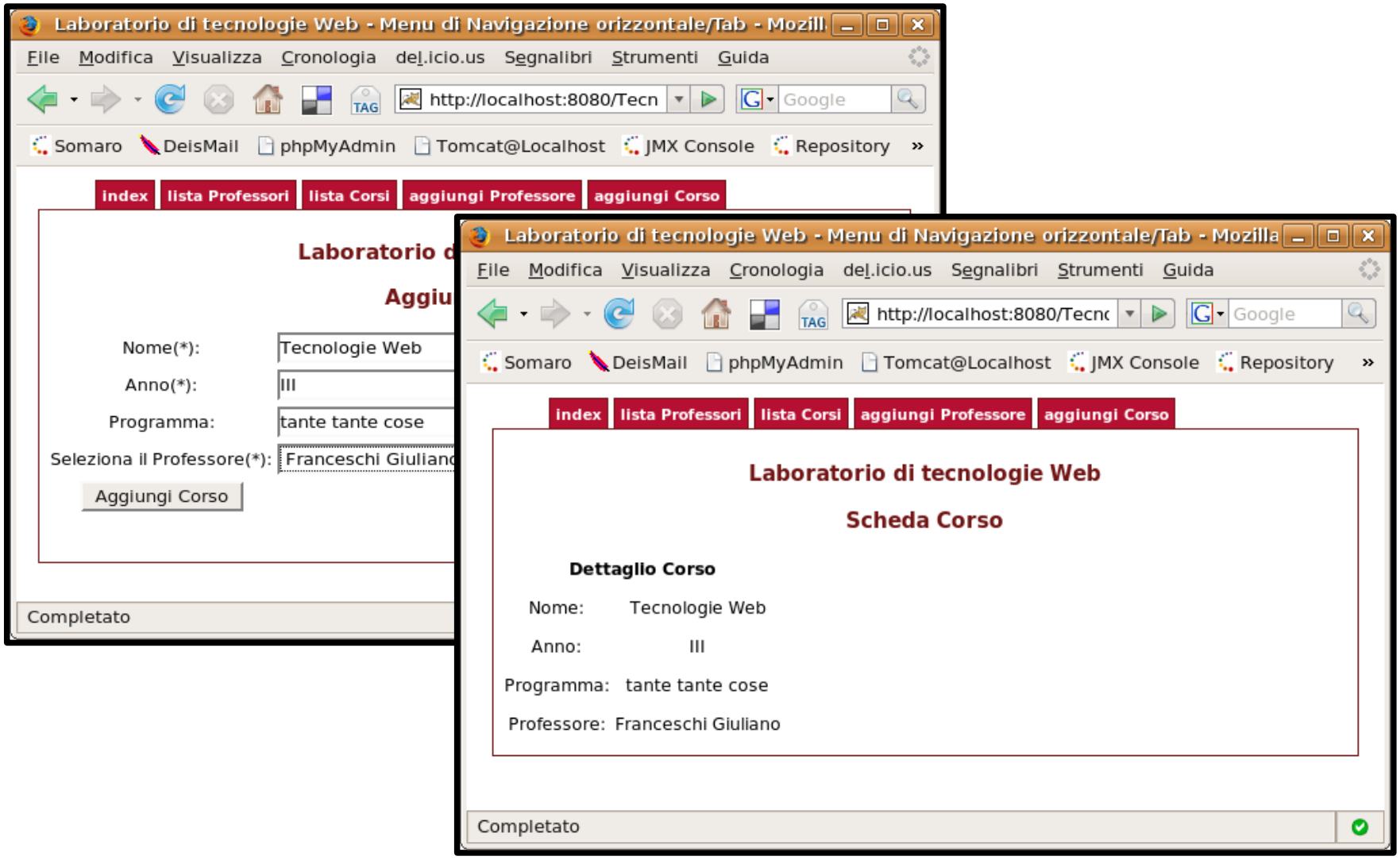
or

- add *<h:inputText>* tag to the *addCorso.jsp* page to bind *manager.corso.programma* field

```
<h:outputText id="outputCorso" value="Programma:"/>
<h:inputText id="inputCorso" value="#{Manager.corso.programma}"/>
```

- add new teaching item from the web app

Extras: show program in teaching details (2/2)



Extras: RowData behaviour (1/2)

> Add a logging feature to check that the right row is selected

- a new column in the *listaProfessori.jsp* view

```
<h:column>
 <f:facet name="header"><h:outputText  value="logga"/></f:facet>
 <h:form >
  <h:commandLink action="#{Manager.loggaProfessore}" value="logga" />
  </h:form>
</h:column>
```

- a new action/navigation method in the `Manager` class

```
public String loggaProfessore() {
    System.out.println( "You have selected: " +
        ((ProfessoreTO)this.dataProfessori.getRowData()).
            getMatricola() );
    return null;
}
```

Extras: RowData behaviour (2/3)

> In the same way, we can add the deletion feature (let's do it for the teachings)

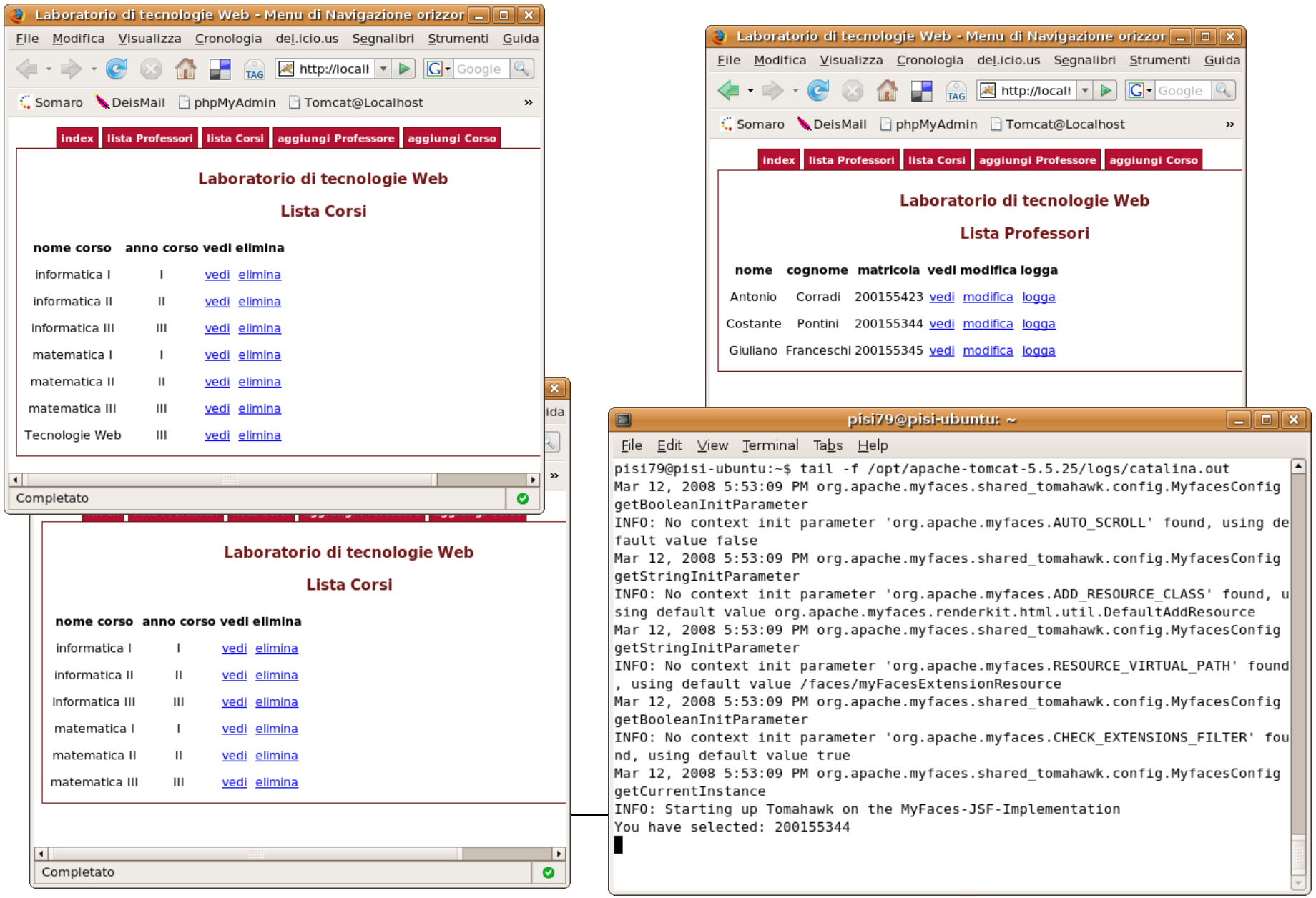
- a new column in the *listaCorsi.jsp* view

```
<h:column>
 <f:facet name="header"><h:outputText  value="elimina"/></f:facet>
 <h:form >
  <h:commandLink action="#{Manager.eliminaCorso}" value="elimina" />
  </h:form>
</h:column>
```

- two more methods in the `Manager` class

```
public String eliminaCorso() {
    this.deleteCorsoTO_from_selected_RowData();
    return "listaCorsi";
}
private void deleteCorsoTO_from_selected_RowData() {
    this.corso = (CorsoTO) this.dataCorsi.getRowData();
    CorsoDAO c_dao = dao_factory.getCorsoDAO();
    // retrieves full features from the database
    c_dao.delCorso(this.corso.getId());
    this.corso = new CorsoTO();
}
```

Extras: RowData behaviour (3/3)



Extras: internationalization (1/3)

> Provided you enabled suitable locales in *faces-config.xml* (and corresponding message bundles, of course) internationalization works fine just by exploiting the user-agent language provided with HTTP requests

- in *faces-config.xml*

```
<application>
    <locale-config>
    <default-locale>it</default-locale>
        <supported-locale>en</supported-locale>
    </locale-config>
</application>
```

- add a suitable *messages_en.properties* file in path *bundle/it/tecnologieweb/pizza/web*
- redeploy the web application

Extras: internationalization (2a/3)

> To support explicit choice of the locale in use, instead

- just copy and paste `LocaleChanger.java` from one of the previous JSF projects we have seen (i.e., *TemplateJSF* or *CdCollectionManager*)
- add the corresponding *localeChanger* bean definition to *faces-config.xml* (we have no *bean-config.xml* in this project... it works all the same, anyway!)

<managed-bean>
<managed-bean-name>localeChanger</managed-bean-name>
<managed-bean-class>
it.tecnologieweb.pizza.web.beans.LocaleChanger</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>

or

- add *bean-config.xml* to the list of the faces config files in *web.xml* and create that files (with *localeChanger* bean definition inside!)

<context-param><param-name>javax.faces.CONFIG_FILES</param-name>
<param-value>/WEB-INF/path-config.xml,/WEB-INF/faces-config.xml,/WEB-INF/bean-config.xml</param-value></context-param>

Extras: internationalization (2b/3)

> Now you can just

- copy the locale selector input controls from one of the previous JSF projects we have seen (i.e., *TemplateJSF* or *CdCollectionManager*)
- paste it to this project home page (*index.jsp*)
- modify it accordingly to the features of the destination project (e.g., we have no Spanish support this time)
- import extra-resources (such as flag images) or change link appearance

```
<h:panelGrid columns="2"> <h:commandLink immediate="true"
    action="#{localeChanger.changeLocaleToEnglish}" value="English
version"></h:commandLink> <h:commandLink immediate="true"
action="#{localeChanger.changeLocaleToItalian}" value="Italian version">
  </h:commandLink> </h:panelGrid>
```

or

you could make things better by creating a new JSF fragment (e.g., *locale.jsp*) and include it in *index.jsp* **within the *<f:view>* opening and closing tags**!!

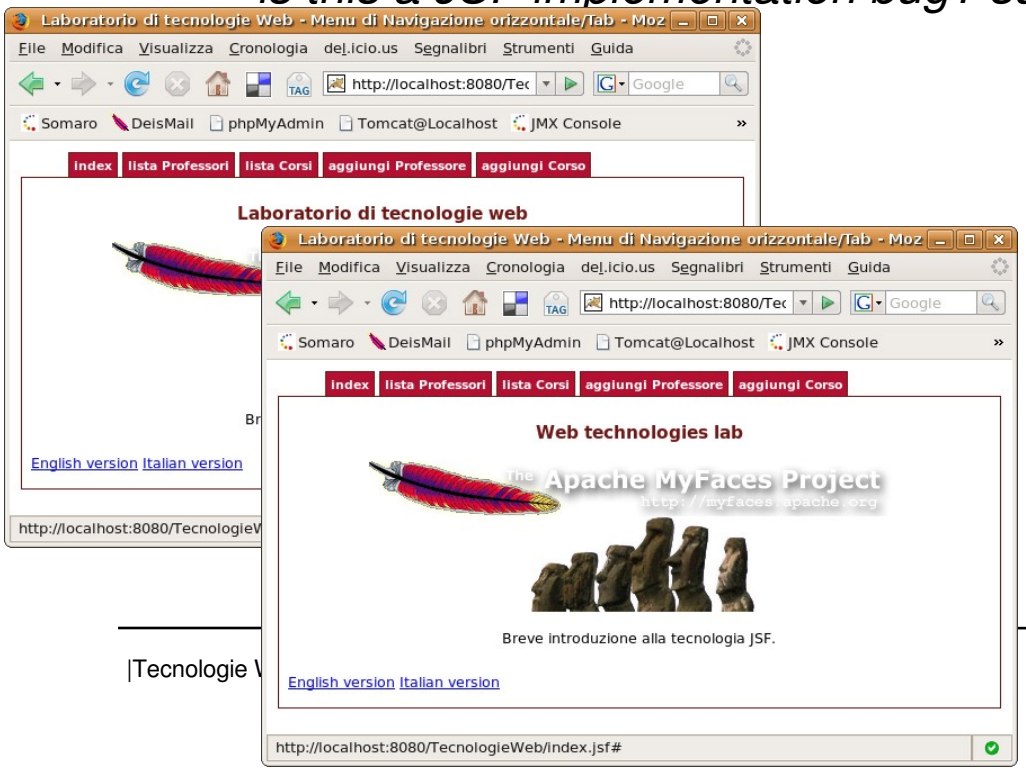
```
<%@include file="locale.jsp" %>
```

- WARNING: *links* must be within an *<h:form>*! Correct *locale.jsp* !!!

Extras: internationalization (3/3)

> Actually, I did some more work:

- added new message property "*title*" in both *it* and *en* message bundle files
- used that property as the value attribute of *<h:outputText>* in *index.jsp*
- why did I do that?
  - for some strange reason, since in *index.jsp* there was nothing that used the bundle, the locale change seemed to take no effect
  - simply adding some bundle-dependent feature we got the right effect
  - *is this a JSF implementation bug? surely it's a strange behaviour....*



please, note: message bundle is loaded by the menu fragment, and menus do not use it, even!!!! → does this help consistency of pages in your opinion? well: in mine, it doesn't. You'd better import the same message bundle twice (with different aliases) in menu and pages and use it with the alias declared by each page in the page where it is defined. This is way it went through in the *CdCollectionManager* project.

Extras: and finally, styles!

> Throughout the project there are *styleClass* (and alike) attributes in JSF tags that correspond to no actual stylesheet rule

- look at the various *<h:panelGrid>* and *<h:dataTable>* tags, for instance
  - you can try to write CSS rules for those classes
  - or you can find out other chances to use JSF *styleClasses* in other tags
- **by adding/editing style-related attribute in JSF pages** and/or **by adding/modifying rules to *tabs.css*** (perhaps with help from *Firebug*!) you can achieve highly picturesque results!
  - as long as you only modify pages, styles, and static resources (I mean: not Java code, XML descriptors and message bundles), you don't need to redeploy your project all the times!!!
  - you can use *update.modified.pages* target in the ANT's *build.xml* file to substitute (on the web server) only the pages you changed (in your Eclipse workspace)! Tomcat will reload them for you upon new user requests!

```
.list-column-center {
        background-color: gray;
}
.list-column-right {
        background-color: yellow;
}
.list-header {
        color: red;
}
```