Anno Accademico 2007-2008

Laboratorio di Tecnologie Web

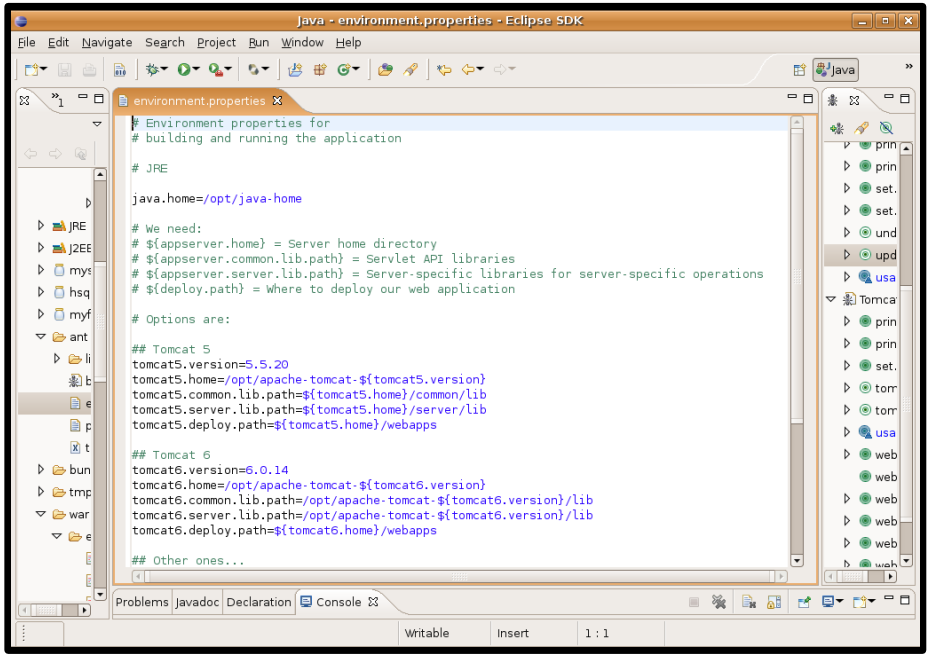
Esempio di applicazione web basata su JSF e DAO

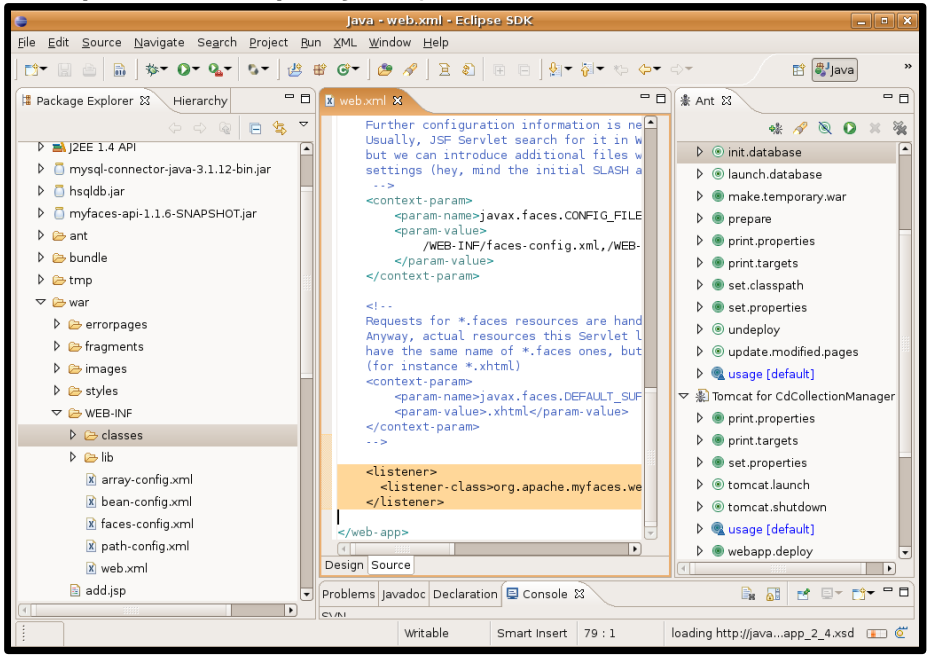
http://www-lia.deis.unibo.it/Courses/TecnologieWeb0708/

Another project from the course web site

> You can handle it as usual:

- download the *CdCollectionManager.zip* archive file
- import it in Eclipse
- set up '*J2EE 1.4 API*' user library (by referring Tomcat libraries), code compiance (must be 5.0) and installed JRE (found a JDK5 on your machine)
- modify *ant/environment.properties* accordingly to your machine settings (or replace this file with a working copy from a previous project)

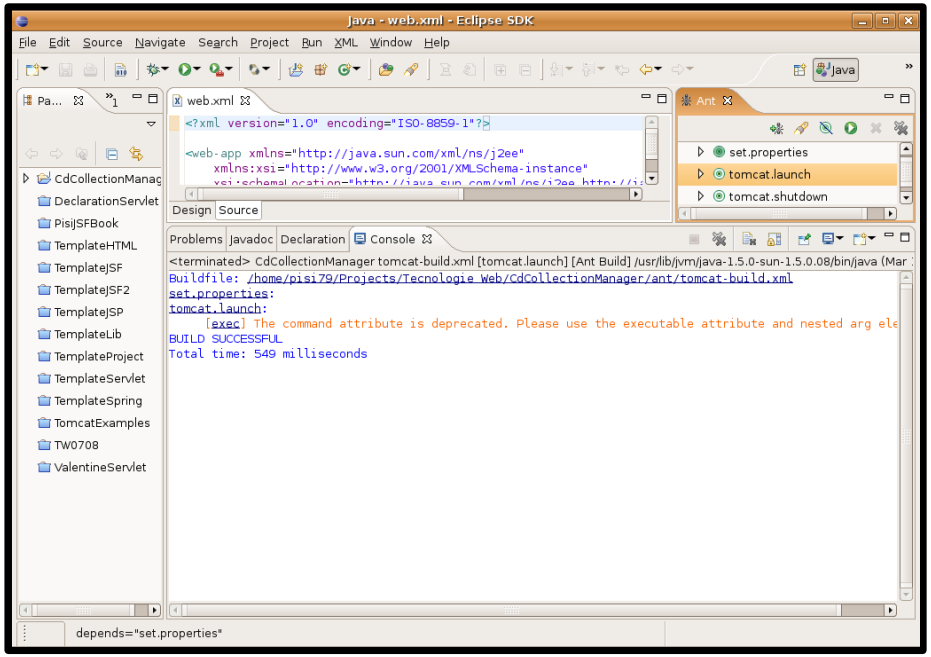




Another project from the course web site

> You can handle it as usual:

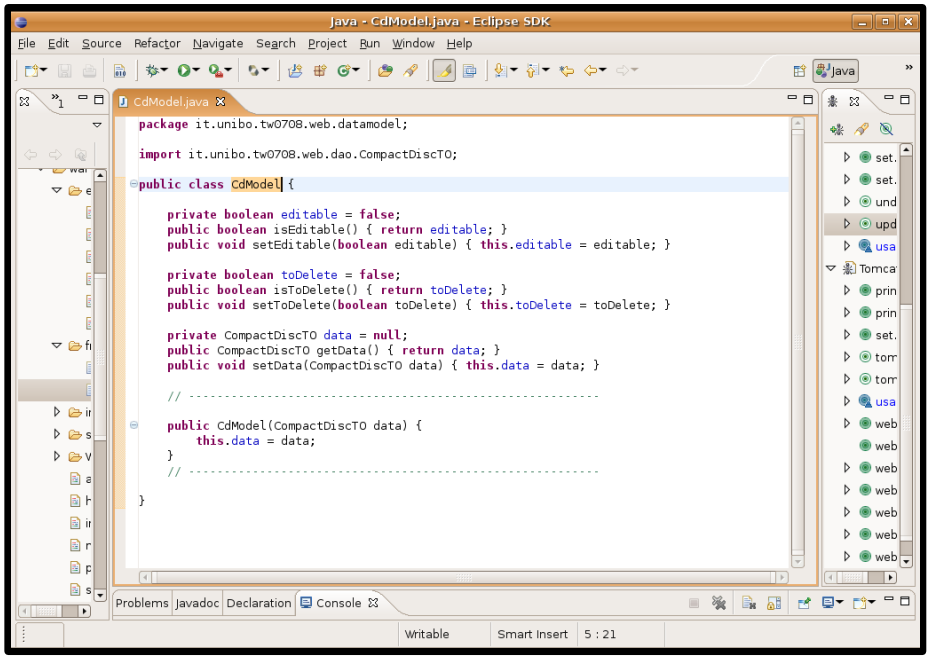
- setup Tomcat users (**either** in *ant/environemnt.properties* **or** in *$TOMCAT_HOME/conf/tomcat-users.xml* – you can find a sample file in the project *tmp* directory)
- launch Tomcat (**either** by issuing prompt commands **or** via the *ant/tomat-build.xml*'s *tomcat.launch* target)

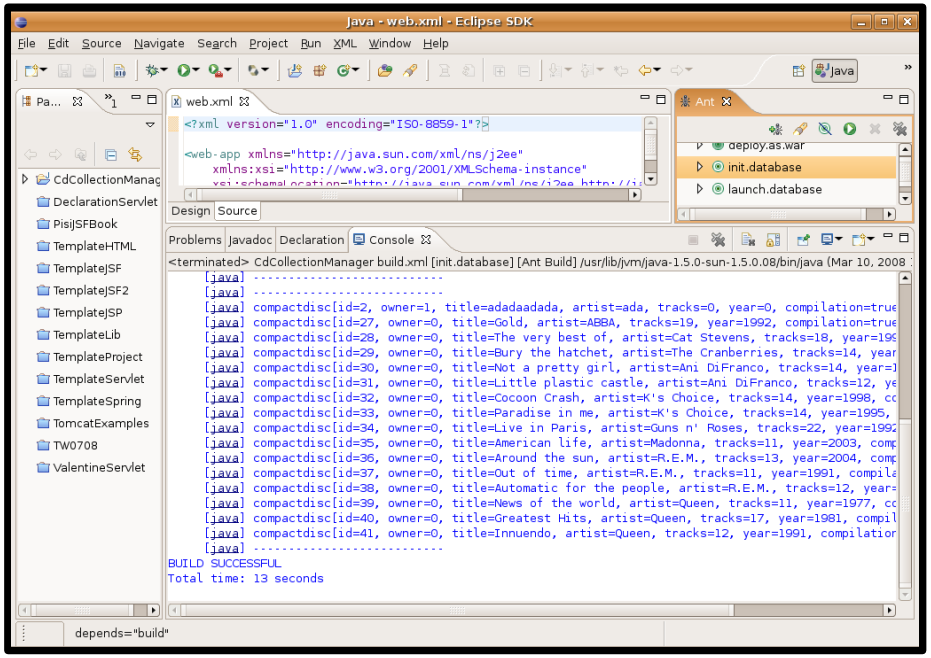


Another project from the course web site

> Besides, this time (the same way you did in the *TemplateJSF* project, only):

- launch the HSQLDB database server (via the *ant/build.xml*'s *launch.database* target)
- initialize database tables (via the *ant/build.xml*'s *init.database* target)





Are you ready?

> Now you can deploy the web application:

- by just copying the war archive (**xor** its corresponding exploded dir) to the *$TOMCAT_HOME/webapps* path (via the *ant/build.xml*'s *deploy.as.war* or *deploy.as.war* target)
- by remotely managing Tomcat (via the *ant/tomcat-build.xml*'s *webapp.redeploy* target

  → see the chain of dependencies among target that this entails!

  → NOTICE: double-clicking this target in the *Ant View* is sufficient to **redeploy your web application if you modify it**: it does everything that's needed to completely perform this task (i.e., cleaning, recompiling, re-packaging, undeploying, deploying and removal of temporary files)

  ...provided you configure Tomcat's users correctly, of course.

Let's see what's there... (1/3)

- A few JSF beans
  - coded in the `it.unibo.tw0708.web.beans` package
  - mapped in the *war/WEB-INF/bean-config.xml* file
- A few JSP pages that uses tags from the JSF framework
  - *by just declaring their corresponding tag libraries*
  - *war/*.jsp* and *war/errorpages/*.jsp* files
- A few static resources (such as pictures to associate to particular input controls)
  - mapped in the *war/WEB-INF/array-config.xml* file
- A few navigation rules (especially for the user selection process, as we'll see)
  - mapped in the *war/WEB-INF/path-config.xml* file
- Reusable HTML/JSP/JSF fragments
  - for creating footers and navigation tabs (same as in *TemplateJSP* project)
  - *war/fragmens/** files
- ...

Let's see what's there... (2/3)

- ...
- Some graphics
  - *war/images/** files
- Bundle messages for internationalization
  - *bundle/*** files
  - ANT targets package these files just as they were classes, in the */WEB-INF/classes* path
- DAO pattern classes
  - transfer objects and abstract/interface definitions are coded in the `it.unibo.tw0708.web.dao` package
  - HSQLDB implementation is in `it.unibo.tw0708.web.dao.hsqldb` one
- `javax.faces.model.DataModel` -related classes
  - to work fine with *<h:dataTable>* tags
- Utilities (already seen in previous projects and perhaps partially unused here)
  - (optionally) do it yourself: ***log4j*** is a better tool to perform logging → modify `it.unibo.tw0708.web.utils.Logger` class to support it

Let's see what's there... (3/3)

> ...

> Client classes

- coded in the `it.unibo.tw0708.client` package
- provide `main()`s methods to initialize db tables and verify data
- these classes too leverage the DAO objects to access the database!

About DAO: noticeables

> Only *Hypersonic* implementation this time

- you can provide yourself a *MySql / Postgres / DB2 / whatever* one if you need it.. ..or simply want to try
  - you need the database server running
  - you need the JDBC connector libraries in the *WEB-INF/lib* path
  - you also need to add this library to the project's Eclipse build-path
- or you can safely remove the *mysql-connector-java-3.1.12-bin.jar* library
  - from the project's Eclipse build-path
  - and from the *WEB-INF/lib* path on the file system

↓

- ...I left it there intentionally, as **I created the CdCollectionManager by cloning the *TemplateJSF* one**!!!
  - copied and pasted that project in my Eclipse workspace
  - changed the project name
  - changed the webapp name property in *ant/project.properties*
  - changed the project name in *ant/build.xml* and *ant/tomcat-build.xml*

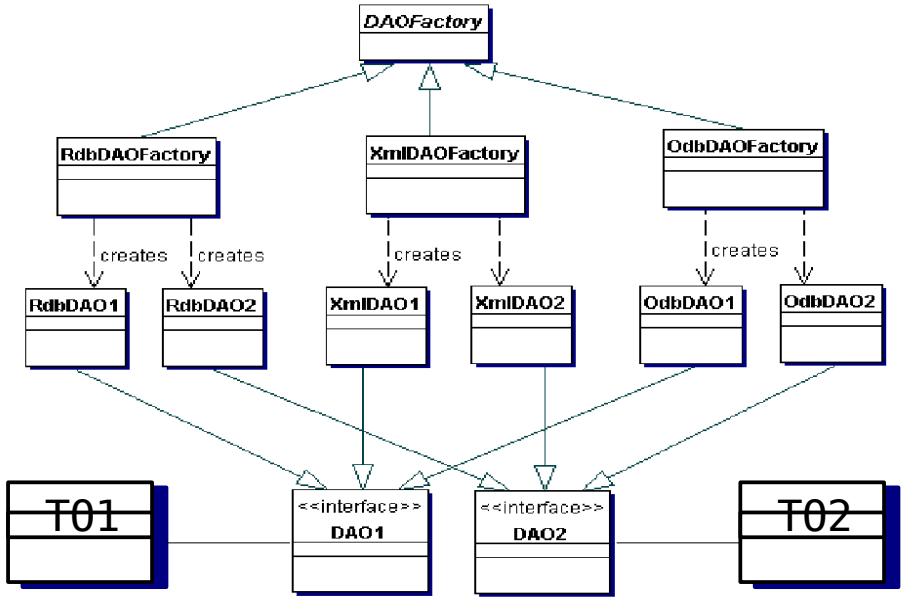
About DAO: things in place

> Two DAO objects: `it.unibo.tw0708.web.dao.UserDAO` and `it.unibo.tw0708.web.dao.CompactDiscDAO`

- and their corresponding transfer ones: `it.unibo.tw0708.web.dao.UserTO`, `it.unibo.tw0708.web.dao.CompactDiscTO`
- DAO objects provide database access by wrapping JDBC calls and exchanging data in the form of transfer objects with their invokers

> Abstract factory `it.unibo.tw0708.web.dao.DAOFactory` lets you instantiate a concrete one

- `it.unibo.tw0708.web.dao.hsqldb.HsqldbDAOFactory` concrete factory (the only one available) provides *HSQLDB* versions of the DAO objects



About the JSF beans

> `it.unibo.tw0708.web.beans.User` provides user selection / addition facilities (data storaging, navigation actions, ...)

- it is mapped as *user* session-scoped bean in *WEB-INF/bean-config.xml* file

> `it.unibo.tw0708.web.beans.LocaleChanger` changes the locale in use

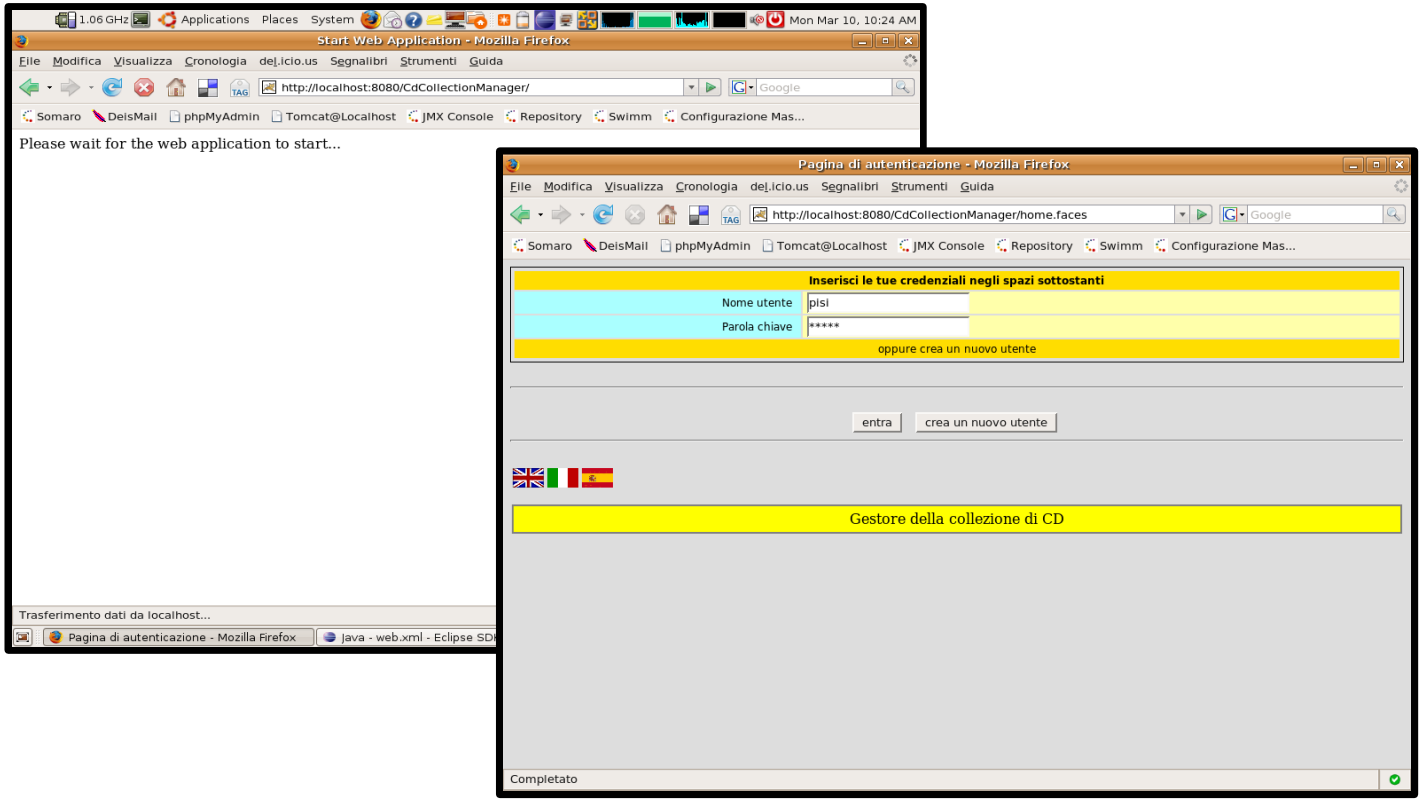
- it is mapped as *localeChanger* session-scoped bean in *WEB-INF/bean-config.xml* file
- all stuff we've already seen in the *TemplateJSF* project
  - hey, notice! this time not all locales provide all message bundles and sometimes not all expected messages are preset in a certain bundle → see how the framework falls back to messages from the default locale (*en*), as mapped in the *WEB-INF/faces-config.xml* file

> `it.unibo.tw0708.web.beans.CdManager` backs the most part of the web application

- it is mapped as *manager* session-scoped bean in *WEB-INF/bean-config.xml*
- we are gonna analyze its behaviour page by page in the following

Introductory pages (1/3)

> *index.html* shows the usual '*please wait*' message and redirects to the actual JSF home page



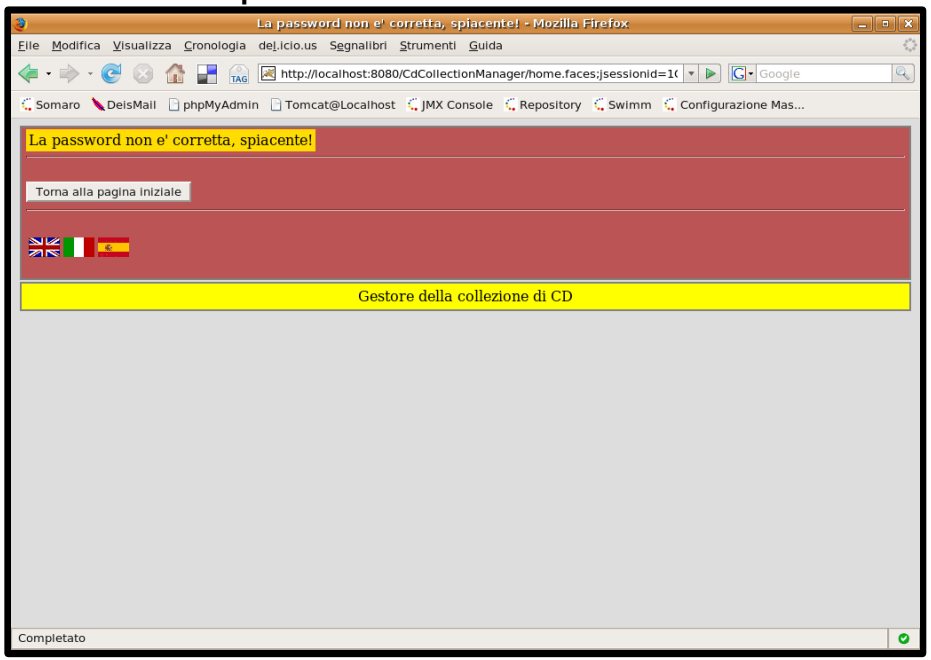
> *home.jsp* provide input controls for specifying user credentials and to change the locale currently in use

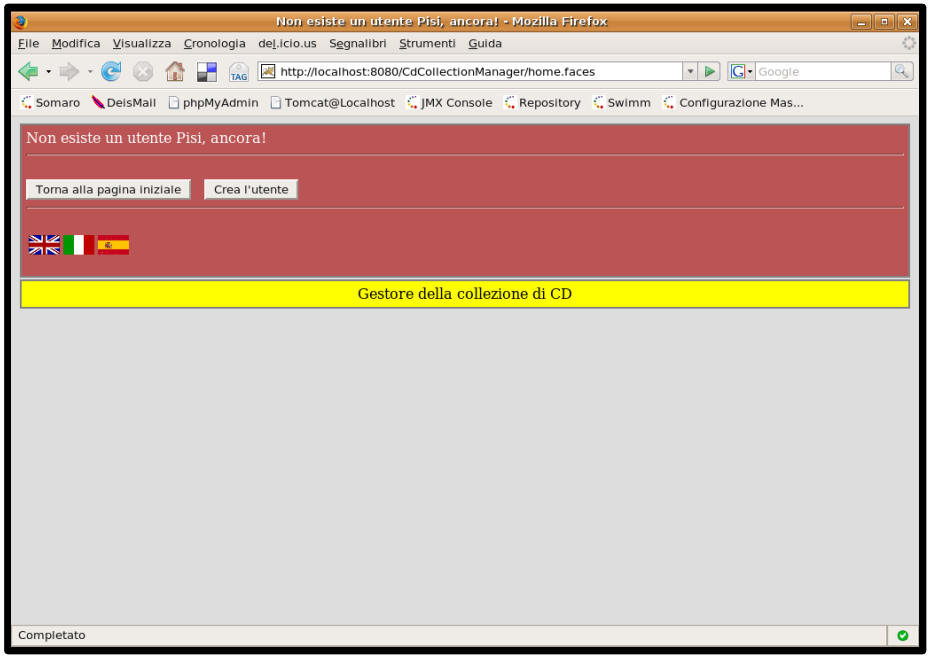
- locale changing works exactly the same as in *TemplateJSF* project

Introductory pages (1/3)

> *home.jsp* stores user input in member variables of the *user* bean and invokes an action method from it

- navigation rules in *WEB-INF/path-config.xml* lead to *errorpages/<err_type>_error.jsp* pages in case something wrong happens
- this time too, **this is not a real authentication process** as **we have not defined any security constraint** in the *WEB-INF/web.xml* web application descriptor

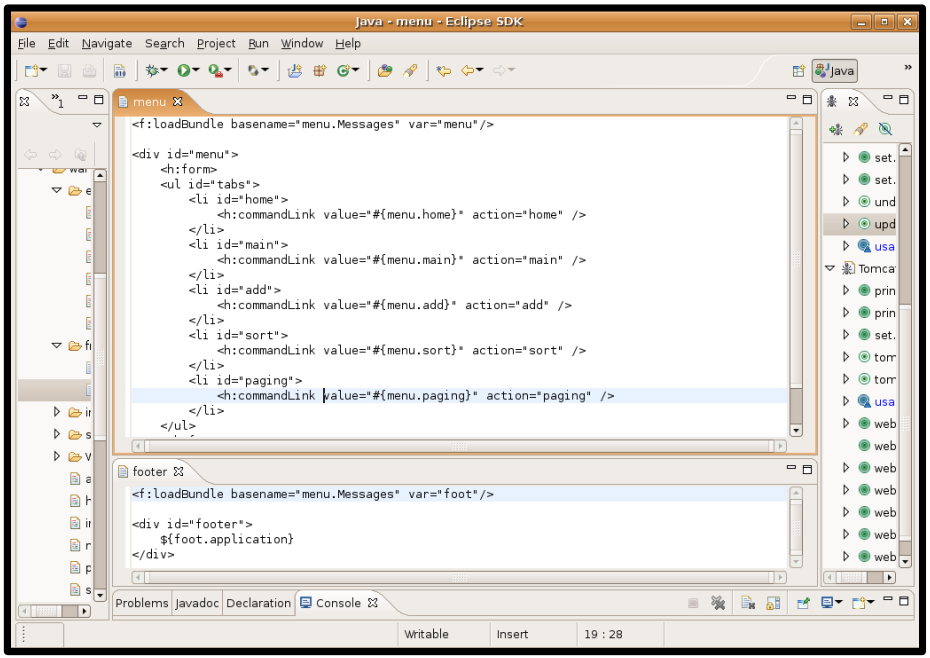




Introductory pages (3/3)

> *fragments/menu* and *fragments/footer* fragments provide code excerpts for rendering a common footer and a tab navigation pane in (almost) all pages of the web application

- we have already embedded JSP fragments in JSP pages in the past → it also works with JSF parts, though it can be somewhat more difficult to keep consistency: look at the message bundle imports and tag library declaration...



User selection errors

> *home.jsp* triggers *user* bean methods that compare input by user with data stored in the user table of the underlying database and return navigation actions accordingly:

- if username exists and password digest matches the stored one the user is allowed to navigate to *main.jsp* page
- if username does not exist or password is not correct the user receives suitable error pages (as mapped in *WEB-INF/path-config.xml* descriptor)

↓

- ...*errorpages/nouser_error.jsp* does not embed *styles/default.css* !
  - you can add it yourself (for instance by copying the corresponding line of code from one of the other *errorpages/xxx_error.jsp* pages)
  - then you can use the *update.modified.pages* target in *ant/build.xml* to update only the modified page on the server!
    - **it is not an actual redeployment**! just a file copy / replacement!
    - **it works only with static resources** (i.e., not with message bundles, deployment descriptors and Java classes)
    - but it is a lot faster than completely redeploy your project (and you ain't losing your session data, this way!)

main.jsp page (1/3)

> *main.jsp* is the first page that gets shown after the 'login-alike' process succeeds

- an *<h:dataTable>* tag iterates over a `javax.faces.model.DataModel` element that wraps the actual list of cd items

> *manager* bean provides

- a getter method for retrieving the datamodel object (by invoking initialization methods and DAO functionalities, in case)
- additional methods for handling editing/deleting or modification discarding
  - all these methods return to the same current view
  - we change the page being shown just by selecting links from the tabs in the upper part of the screen (and not by clicking any page buttton, this time)

> `javax.faces.model.DataModel` is an abstract class: we build up the actual datamodel for <h:dataTable> by using out-of-the-box utility classes (i.e., `javax.faces.model.ListDataModel`) to wrap the cd list

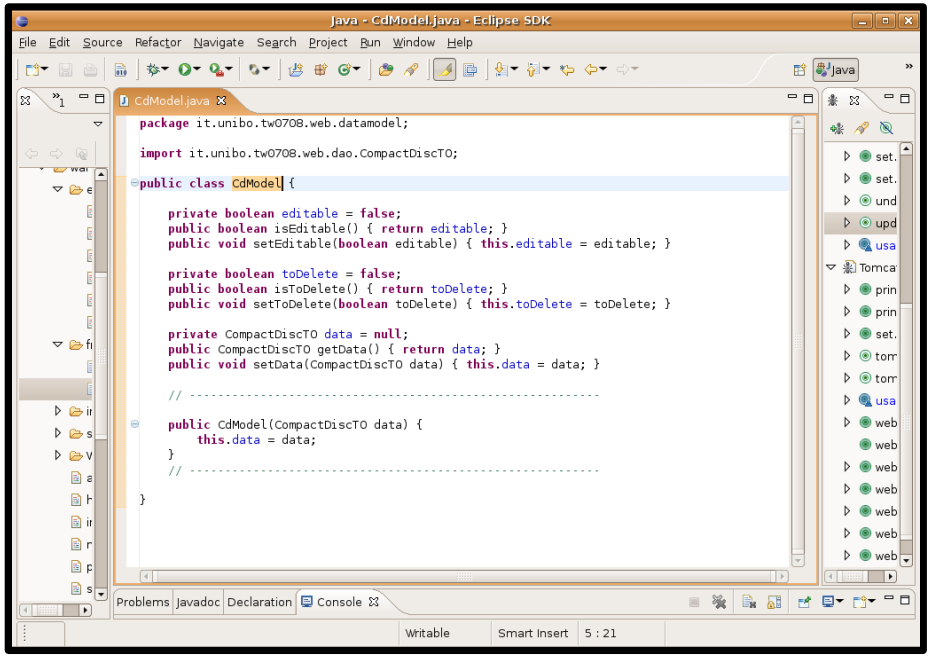
- see `it.unibo.tw0708.beans.CdManager.`*`getCompactDiscs()`* method for details

main.jsp page (2/3)

> to realize the cd items of the datamodel, we don't directly use `it.unibo.tw0708.dao.CompactDiscTO` objects, as we need further features:

- transfer objects are wrapped by `it.unibo.tw0708.datamodel.CdModel` objects
- the latter ones support the extended logic we need

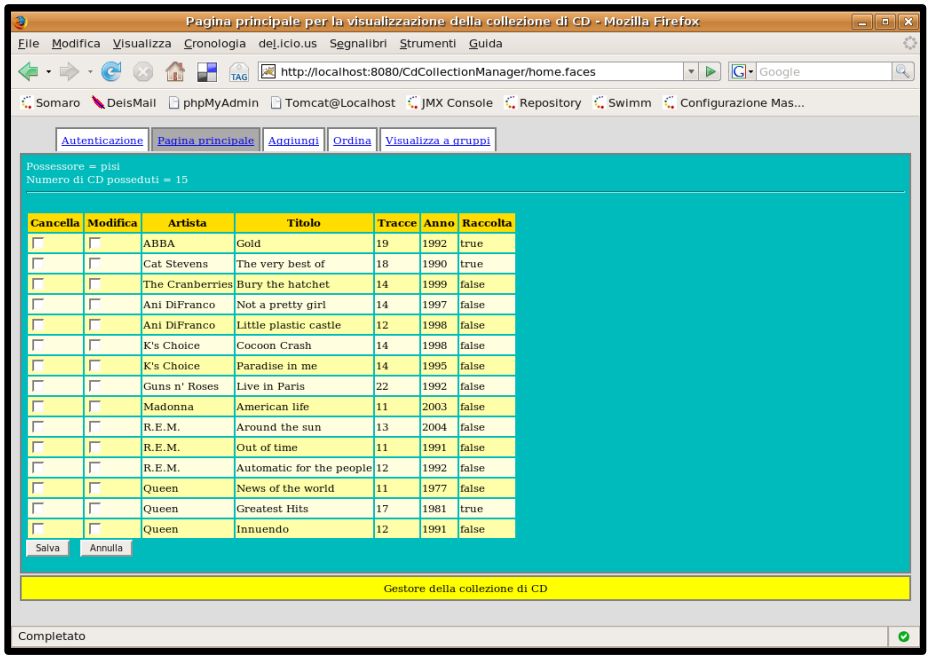
  → some boolean flags to indicate that the cd is being edited or deleted

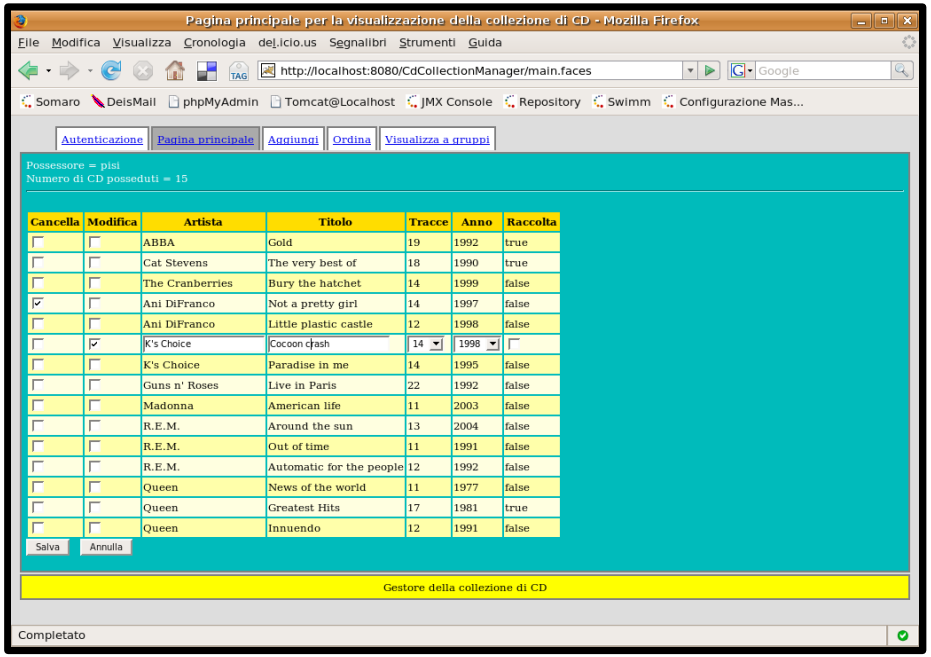


main.jsp page (2/2)

> columns in *<h:dataTable>* display one of two different controls (respectively of input and output type) for each cd field, depending on the status of the boolean flags:

- since flag controls present the *onChange="submit()"* attribute, page view is immediately recalculated after flag modification (by triggering the JSF life-cycle you already know)
- the *manager* bean holds the cd items referenced by the *dataTable* UI component, so that modifications can be applied to the page view, before the user chooses to persist them to the database





CdManager: a deeper look (1/2)

> it keeps track of the current user id, in order to re-act to changes

- for instance, the physical user could re-login with different credentials

> **it holds data for the** *main.jsp* **view** and it provides methods for resetting and refreshing it when needed

- it is necessary to have a member variable where to store modifications provided by the end-user, before and after writing them to the database
- **page UI components** such as *dataTable* **refer to this data** (in this case, the cd list) to read what to show to the user and to store input from the user herself

CdManager: a deeper look (2/2)

> **update methods provide business logic and return suitable navigation actions**

- `it.unibo.tw0708.beans.CdManager.updateCompactDiscs()`:
  - → **write** modifications/deletions **to the database**, if needed
  - → refresh data for the current view (by resetting it = resyncing it to the db)
  - → redisplay the current view (by returning *null*)
- `it.unibo.tw0708.beans.CdManager.updateCompactDiscs()`:
  - → reset data for the current view (= resyncing it to the db status)
  - → redisplay the current view (by returning *null*)

> **the same schema repeats itself** almost un-modified **in all the pages** that are backed by the manager

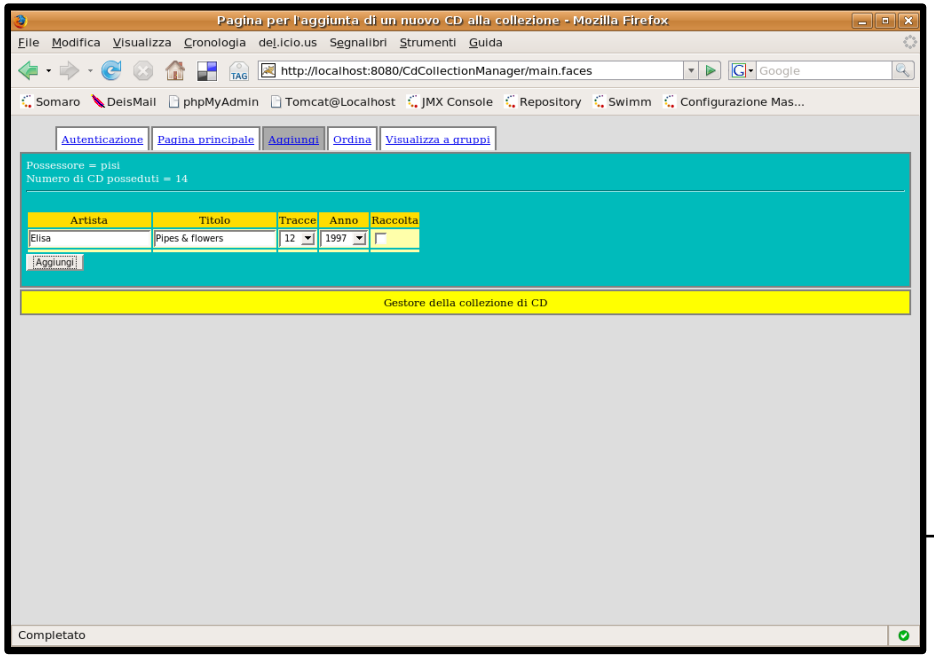
- **to hold the data referred by the page UI components**

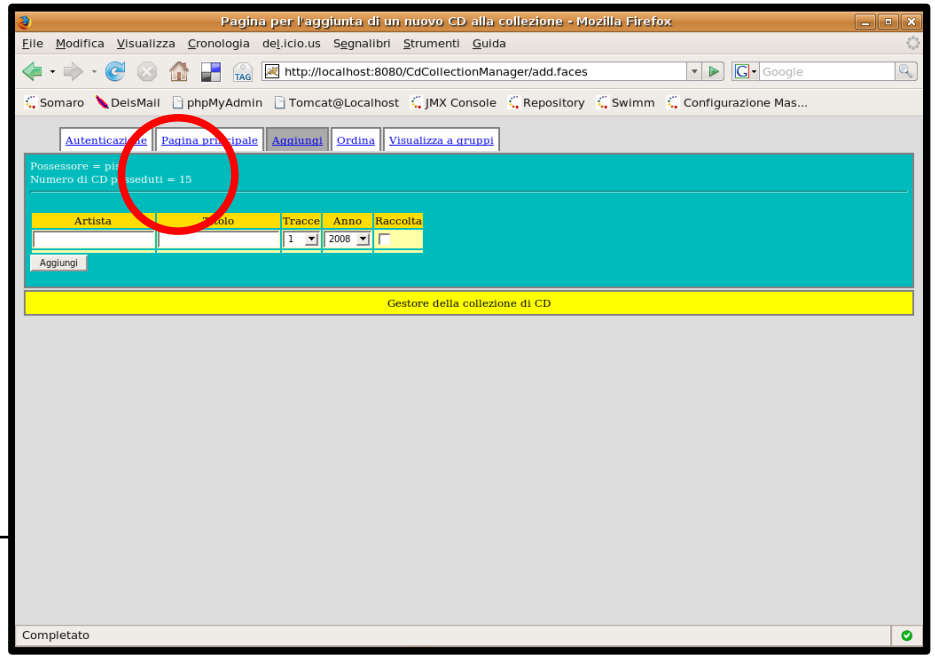
- **to provide the methods that perform business logic on that data** (e.g., updating the database) **and return a suitable navigation action**

add.jsp page

> no *<h:dataTable>* tags and javax.faces.model.DataModel objects this time

- just **a member variable** of type `it.unibo.tw0708.dato.CompactDiscTO` in the backing bean and an *<h:panelGrid>* tag to show the cd fields to edit
- page button invokes *manager* method `it.unibo.tw0708.beans.CdManager.insertCompactDisc()` to perform the business logic that entails:
  - → writing the new entry to the database (by invoking DAO functionalities)
  - → updating the cd list in the *manager* (by re-syncing it to the database)
  - → redisplaying the current view (by returning a null action)





sort.jsp page (1/2)

> Just some tricks you can play on the datamodel before displaying it

- *manager* bean holds **another member variable** (different from the cd list we have already seen) **where to store the sorted datamodel list**
- the abstract `it.unibo.tw0708.datamodel.sorting.SortableDataModel` class defines a private subclass **`Row`** to

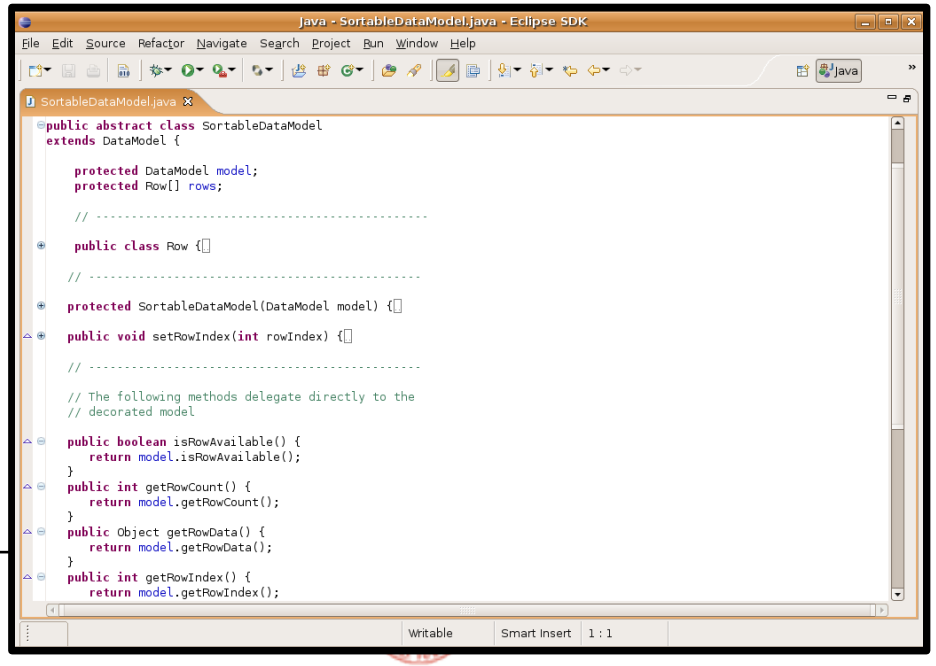
  → **store row index for sorting** (`Row` has just an `int` field)

  → **provide an accessor method for returning the index-corresponding row from the datamodel object being wrapped** by `SortableDataModel` class (this is why `Row` is an inner class of it: *for the sake of visibility scope!*)

- **`SortableDataModel`** then:

  → **wraps the traditional `javax.faces.DataModel` and** holds **the index array** (= array of `Row` objects) that reflects the current sorting

  → **bridges the rest of `DataModel` methods** to the wrapped object



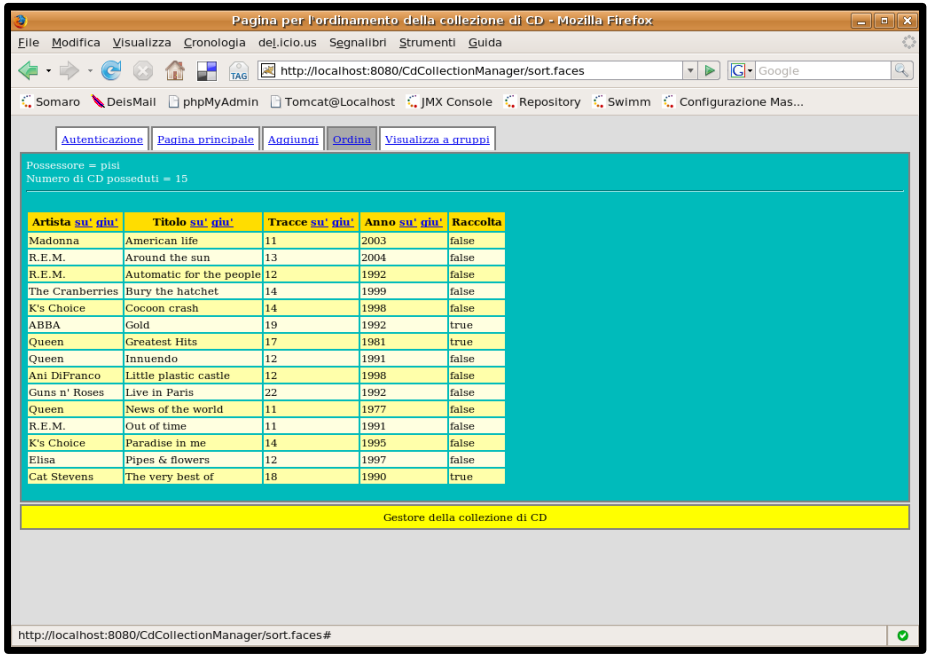
sort.jsp page (2/2)

> Just some tricks you can play on the datamodel before displaying it

- concrete class `it.unibo.tw0708.datamodel.sorting.SortableCdDataModel` extends `SortableDataModel` and provides methods to perform the actual sorting:

  → these methods just leverage `java.util.Array.sort()` function by providing suitable comparator objects...

  → if you want, you can have a look at the comparator classes in `it.unibo.tw0708.datamodel.sorting.comparators` package (they are just object implementing the `java.util.Comparator` interface by provinding a suitable *`compare()`* method that deals with `CdModel` objects)

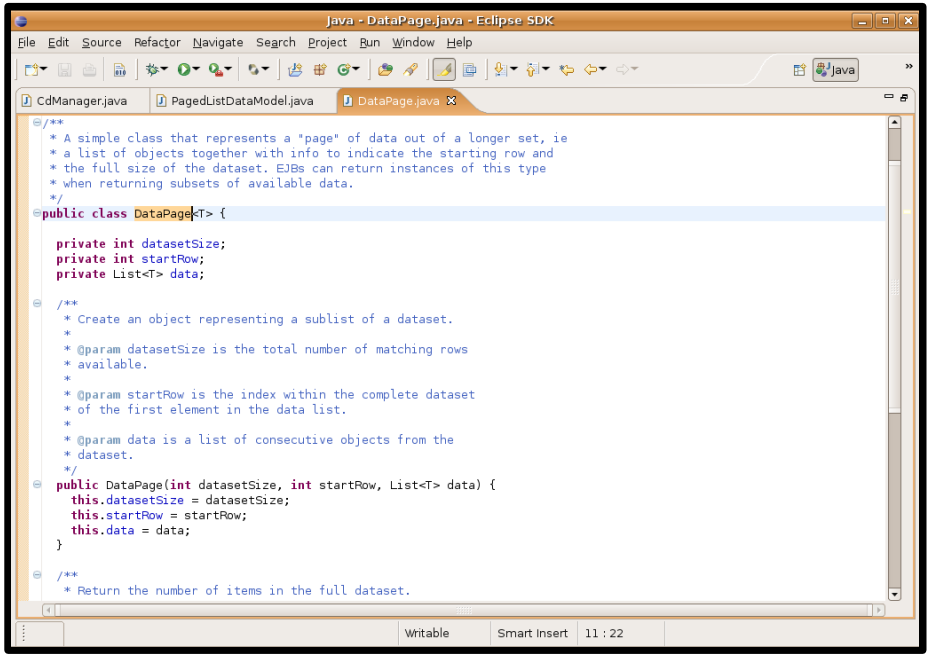


- *<h:commandLink>*s in the page force resorting the model (by invoking *`sort_xxx()`* methods) and redisplaying the *<h:dataTable>* content accordingly

paging.jsp page (1/3)

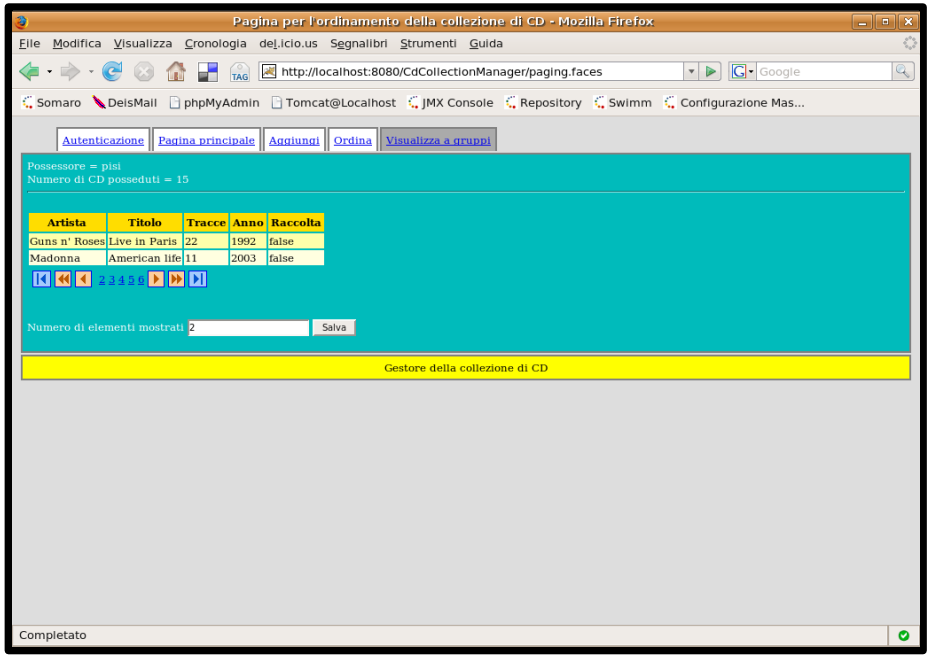
> **paginating** data is a bit more complex (but it **avoids long-lasting queries** when the database holds huge tables → *<h:dataTable>* would load everything at once!!)

- again, we wrap the traditional `javax.faces.DataModel` into an enhanced datamodel object that provides additional features and we use it in the page: `it.unibo.tw0708.datamodel.paging.PagedCdDataModel`
- again, we gather non-datatype-dependent methods into an abstract superclass (→ you can re-use this kind of classes for your own project data!): `it.unibo.tw0708.datamodel.paging.PagedListDataModel`
- this time we also need a generic helper class to store the current data page and additional related information such as starting row, page number, ecc: `it.unibo.tw0708.datamodel.paging.DataPage<T>`



paging.jsp page (2/3)

> furthermore, this time we are not going to create everything on our own: scroller controls **UI components** are **provided by a third-party library**: *Tomahawk* (from the *Apache group*)



- you can find *tomahawk-1.1.5-SNAPSHOT.jar* library under *WEB-INF/lib* path

- the JSF page declares and uses it just like the other tag libraries

<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t" %>

paging.jsp page (3/3)

> *manager* bean holds the current page of the datamodel (= subset of datamodel rows) in another member variable: *pagedCompactDiscs*

- *<h:dataTable>* iterates over data from this object
- controls from the *<t:dataScroller>* UI component do not invoke any cd *manager* method, but directly modify the datamodel object represented by *pagedCompactDiscs*, invoking its *getRowData()* and similar methods
- **it is the ehnanced datamodel object that performs all the logic to return results page by page and to retrieve new data rows only when needed**

  → this is achieved by overriding the default behaviour of *getRowData()* method and alike ones

  → just try to reuse this features into your projects: it's by far simpler than understanding things theoretically