PFC: Implementació d'una funcionalitat en un framework per una aplicació J2EE

Luis Gallego Ruestes
Enginyeria en Informàtica
Consultor: Josep Maria Camps Riba
18 de Juny de 2007

UOC
Universitat Oberta de Catalunya
www.uoc.edu

A Geni

El meu reconeixement a tots els professors de l'UOC responsables de tenir el nivell suficient per arribar fins aquí i en especial el meu agraïment a Josep Maria Camps Riba

----------^----------

Resum

Si comparem el present PFC amb un llarg viatge començaríem per els dos punts de referència: la partida i l'arribada. Per a nosaltres el punt final, l'objectiu principal, és aconseguir la implementació d'una funcionalitat en un *framework* per una aplicació J2EE. Ha de quedar clar que no es tracta només d'obtenir la funcionalitat sinó saber obtenir-la mitjançant una implementació.

En el viatge hauríem de marcar les etapes que nos portaran de l'origen a la destinació, la implementació desitjada. Quin és l'origen? Sense cap dubte hem de sortir amb tot un bagatge de coneixements sobre aplicacions J2EE. Necessitem tenir al nostre abast tot el que representa contenidors, capes, model MVC, *beans*, bases de dades i una llarga llista d'elements que hem aprés principalment en l'assignatura EPCSD.

Quines són les etapes intermèdies entre els punts de sortida i arribada? Realment moltes, tanmateix s'haurien de resumir en dos: conèixer els *frameworks* i utilitzar-los en la creació d'una aplicació J2EE.

Conèixer els *frameworks* ha consistit una etapa dificultosa amb gran quantitat de material per estudiar i analitzar. Era necessari esbrinar el comportament d'un *framework* en una aplicació J2EE. Sortosament nos hem sortit. Hem obtingut uns coneixements de molta utilitat sobre concrets *frameworks*.

La següent etapa era posar en pràctica tot el aprés en l'anterior. Volíem crear una aplicació J2EE mitjançant la integració de diferents *frameworks*. També nos hem sortit i hem aconseguit un programa on tenir el port d'arribada: la implementació desitjada.

Culminar un llarg viatge representa sacrifici i ambició. Però també humilitat. No hauríem arribat al final sense l'ajud de les magnífiques cartes de navegació que han resultat ser certes webs i determinats llibres.

----------^----------

Índex

---- segueix

Capítol 3: Integració i modificació de *frameworks* en una aplicació J2EE

Capítol 4: Implementació d'una funcionalitat en un framework

Capítol 5: Conclusions

----------^----------

Capítol 1: Introducció

Descripció del PFC

Els continguts del projecte vessaran sobre el resultat de conjuntar dos o més *frameworks* en el desenvolupament una aplicació J2EE i amés un dels *frameworks* serà modificat per tal d'oferir una nova funcionalitat o millorar una de establerta a l'aplicació. Abans però s'haurà obtingut una informació suficient respecte a tots els *frameworks* importants i disponibles per a implementar les capes de l'aplicació. Igualment s'estudiaran les possibilitats de modificar un *framework*. També s'hauran copsat els fitxers de configuració i descriptors que requereixen aquestes tecnologies.

El projecte se centra en quatre elements: empresa o activitat, usuaris, aplicació J2EE i els *frameworks*. En tot projecte humà existeix el desig de portar a terme una empresa i si ja està constituïda aleshores desenvolupar en la mateixa un seguit d'activitats a través d'uns mitjans. En el nostre cas els mitjans seran aplicacions J2EE que s'implementaran amb el recurs d'una tecnologia anomenada *framework*. La utilització de l'aplicació J2EE permetrà a diferents usuaris, clients, gestors, estudiants, etc. treure uns beneficis en forma de informació, lleure, negoci, etc. El interès últim del projecte és aprofitar tota la força dels *frameworks* mitjançant el coneixement de les seves característiques, estructura i possible integració de varis d'ells en una sola aplicació i aconseguir noves funcions a través de la modificació d'un *framework*.

Bàsicament està constituït per tres apartats: 1) recull de informació sobre els *frameworks* ; 2) integració de *frameworks* en una aplicació J2EE i 3) implementació d'una funcionalitat en un *framework*.

En el capítol 2 anomenat 'Conèixer els *frameworks*' les principals tasques consistiran en l'obtenció de bibliografia en paper i on-line sobre aquestes tecnologies. Un cop seleccionades les fonts de informació per el seu abast i claredat en la exposició, així com l'aportació d'exemples i codi font, es passarà a l'extracció dels conceptes, característiques i funcionalitats més importants dels *frameworks*, saber les capes on actuen, acoblament entre si, facilitat de ús i adaptació, i disposar de criteris fiables per tal de poder decidir quan son útils o necessaris i quan no. Igualment es destacarà la importància de saber augmentar el rendiment d'un *framework* per part del desenvolupador si li sap afegir el codi adequat i aconseguir un 'nou' *framework* més adequat a l'aplicació que es vol crear.

És important conèixer para que serveixen els *frameworks*, com funcionen, quins hi ha, així mateix hem de disposar d'exemples i de codi font reutilitzable.

En el capítol 3 anomenat ‘Integració i modificació de *frameworks* en una aplicació J2EE’ s’aplicaran els coneixements elaborats en el capítol 2 per a construir una aplicació web que pugui ser emprada realment. L’aplicació complirà els requeriments des de els vessants de l’empresa, l’usuari, J2EE i els *frameworks*. Ja s’indica per endavant que la formaran tres capes amb contenidors sota el patró MVC. Per la capa de vista es construirà la implementació amb el *framework* JavaServer Faces (JSF). Hi haurà un servlet com controlador i així mateix s’aplicarà la tecnologia del *framework* Hibernate per la capa de persistència.

En la integració intervindrà un tercer *framework* del tipus d’aplicació, *Spring*, que forjarà l’acoblament de *Hibernate* i *JSF* en el nostre programa.

En el capítol 4 ‘Implementació d’una funcionalitat en un framework’ utilitzarem de nou els coneixements del capítol 2 per una labor fonamental i raó última d’aquest projecte: la modificació d’un *framework* per tal d’ampliar o millorar el seu rendiment en l’aplicació J2EE. En concret consistirà en la implementació d’un validador en el *framework* JSF.

Per tal de portar a terme el PFC hem necessitat a banda dels *frameworks*, coneixements i codi, uns elements imprescindibles o en tot cas molt importants en la nostra aplicació: el gestor de dades i el servidor. L’aplicació MySQL és el candidat triat per gestor de BD amb el seu Query Browser que ens permet carregar i actualitzar les taules i continguts. Per el que respecta al servidor de l’aplicació serà Tomcat 5.5.x.

En la elaboració del programa també és necessari un compilador, el més emprat i el que nosaltres farem córrer serà Ant. En la edició de fitxers, tant si són Java con JSP o fins i tot XML, hem utilitzat JCreator, una bona eina auxiliar per Java, edita i compila. En comptes de NetBeans o Eclipse hem fet anar el propi terminal MS-DOS per compilar el fitxer war i cridar el servidor per el desplegament.

Finalment per el que fa a navegadors no hem de tenir cap problema, des de Firefox fins a un Netscape o un IE7 seran perfectes monitors de la nostra aplicació. Nosaltres en particular nos hem decidit per *Firefox 2*.

----------^----------

Objectius generals i específics

Objectius generals:

1) Conèixer els *frameworks*
2) Com utilitzar-los en una aplicació
3) Com modificar-los

Objectius específics:

1) Desenvolupar aplicacions J2EE mitjançant *frameworks*
2) Integrar *frameworks* en el desenvolupament d'aplicacions J2EE
3) Mostrar com un *framework* pot ajudar a modificar el comportament d'un altre *framework* mitjançant integració entre tots dos en l'aplicació J2EE
4) Comprovar les mancances d'un *framework* davant de certs requeriments d'usuari en la creació d'una aplicació J2EE
5) Aconseguir una nova funcionalitat en una aplicació J2EE per mitjà de modificar un o més *frameworks*
6) Aplicar la implementació de la funcionalitat en la modificació del *framework* en altres aplicacions J2EE
7) Editar una metodologia de implementació de funcionalitats en *frameworks* i el seu ús en el desenvolupament d'aplicacions web

----------^----------

Temporització

La planificació del PFC quedarà marcada per les següents dates:

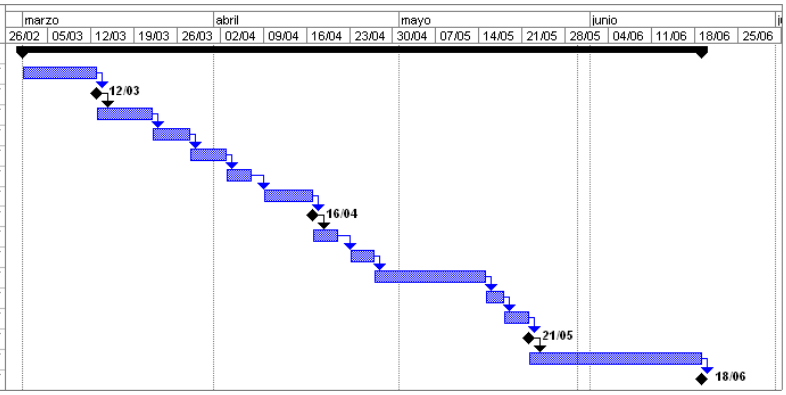
✓	Pla de Treball---------- 01/03/07	12/03/07
✓	Informació sobre els *frameworks* ---- 13/03/07	21/03/07
✓	Frameworks de presentació---------- 22/03/07	27/03/07
✓	Frameworks de persistència---------- 28/03/07	02/04/07
✓	Frameworks d'aplicació----------03/04/07	06/04/07
✓	Possibilitats dels frameworks---------- 09/04/07	16/04/07
✓	Informació integració de frameworks-- 17/04/07	20/04/07
✓	Anàlisi i disseny aplicació J2EE----------23/04/07	26/04/07
✓	Implementació aplicació J2EE----------27/04/07	14/05/07
✓	Implementació funcionalitat----------15/05/07	17/05/07
✓	Proves modificació en aplicació--------- 18/05/07	21/05/07
✓	Edició Memòria i Presentació Virtual--- 22/05/07	18/06/07

Les principals fites del PFC:

✓	Pac 1: Lliurament de la Pac 1	12/03/07
✓	Pac 2: Lliurament de la Pac 2	16/04/07
✓	Pac 3: Lliurament de la Pac 3	21/05/07
✓	Lliurament final: Entrega Memòria i P. Virtual	18/06/07

El detall de la planificació es mostra en el següent diagrama:

Id	Nombre de tarea	Duración	Comienzo
1	**Treball Fi de Carreral**	**78 días**	**jue 01/03/07**
2	Pla de Treball	8 días	jue 01/03/07
3	Pac 1	0 días	lun 12/03/07
4	Informació framewor	7 días	mar 13/03/07
5	Frameworks presenta	4 días	jue 22/03/07
6	Frameworks persistèn	4 días	mié 28/03/07
7	Frameworks d'aplicaci	4 días	mar 03/04/07
8	Possibilitats i modificac	6 días	lun 09/04/07
9	Pac 2	0 días	lun 16/04/07
10	Informació aplicació J2	4 días	mar 17/04/07
11	Anàlisi i disseny aplica	4 días	lun 23/04/07
12	Implementació aplicació	12 días	vie 27/04/07
13	Impolementació funcior	3 días	mar 15/05/07
14	Proves modificació en	2 días	vie 18/05/07
15	Pac 3	0 días	lun 21/05/07
16	Edició Memòria i Prese	20 días	mar 22/05/07
17	Lliurament final	0 días	lun 18/06/07



Enfocament i mètode seguit

Els objectius generals nos han guiat des de el punt de partida: obtenir tota la informació necessària sobre els *frameworks* per tal de crear una aplicació J2EE mitjançant el seu ús, en concret volíem, i aquest ha consistit l'enfocament, una aplicació J2EE obtinguda per la integració de diferents *frameworks*. Consideraven també en l'enfocament la possibilitat d'haver de canviar el comportament estàndard d'algun dels *frameworks*, més encara, era el nostre objectiu implementar una funcionalitat que no estigues en el *framework* de presentació.

Aleshores el mètode seguit ha resultat bàsicament :

1) en la cerca de

- ✓ Informació general sobre els *frameworks,* estructura, fitxers de configuració, capes que implementen, etc.
- ✓ Informació sobre els *frameworks* de presentació i en concret sobre **JSF**
- ✓ Informació sobre els *frameworks* de persistència i en concret sobre **Hibernate**
- ✓ Informació sobre els frameworks d'aplicació i en concret sobre **Spring**
- ✓ Exemples d'aplicacions amb integració de frameworks i si era possible amb els tres *frameworks* citats

2) estudiar el màxim de casos de modificacions en els frameworks, en especial

- ✓ Casos on Spring intervé en la implementació de la capa de presentació i canvia el comportament de JSF
- ✓ Casos on Spring intervé en la implementació de la capa de persistència i canvia el comportament de Hibernate
- ✓ Casos on cal modificar, afegir mappings o settings, als fitxers de configuració de JSF, Spring o Hibernate
- ✓ Casos on calia una funcionalitat nova en pàgines JSP associada a funcions de afegir, modificar o esborrar algun element

3) construir una aplicació J2EE amb JSF, Hibernate i Spring

4) implementar una funcionalitat en JSF en l'aplicació creada

5) provar la implementació obtinguda

6) abstreure la implementació obtinguda per altres aplicacions .

Productes obtinguts

La relació dels productes desenvolupats en el trancurs del PFC han resultat els següents:

1) La Memòria formada per un document en base Word i que amb posteritat s'ha editat en format pdf
2) Un programa consistent en una aplicació J2EE en format WinRAR: *PFCAteneu20.rar*
3) Els fitxers obtinguts en la implementació de la funcionalitat en JSF:
   - ✓ 2 classes Java (directori *validator*):

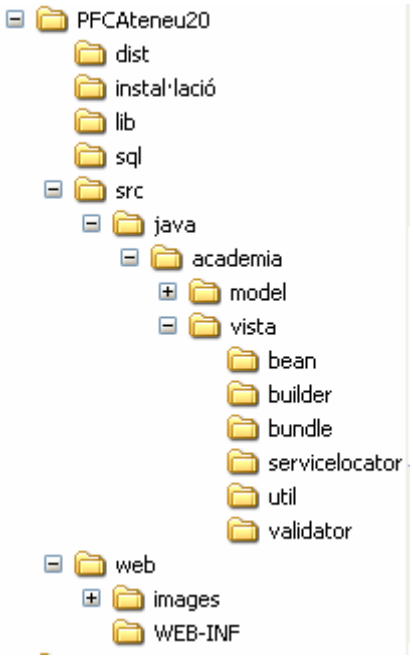
     ValidadorSeccionsSeleccionades.java
     ValidadorSeccionsSeleccionadesEtiqueta.java

   - ✓ 1 etiqueta personalitzada en format tld (directori *WEB-INF*):

     academia.tld

   - ✓ 1 fitxer de configuració en format XML (directori *WEB-INF*)

     validador-config.xml



4) Una presentació d'un resum del PFC en format PowerPoint

Descripció dels capítols 2, 3, 4 i 5

Capítol 2: Consisteix en un recull de conceptes sobre els *frameworks*, descripcions de les seves estructures i funcionalitats, característiques principals i capes de les aplicacions web on actuen. Acompanyen les explicacions imatges i codi quan cal. Destacariem els apartats sobre els fitxers de configuració i fitxers característics (beans, xml, hbm.xml, jsp, etc)

Capítol 3: Consisteix en la creació d'una aplicació J2EE basada en la integració de tres frameworks, JSF, Spring i Hibernate. Es descaten els apartats on apareixen les limitacions o mancances d'algun *framework* en la implementació d'algun servei o funcionalitat, i com es solucionen. Esquemes i diagrames acompanyen les explicacions.

Capítol 4: Consisteix en la implementació d'un validador, funcionalitat requerida en els procediments de donar d'alta un aula o actualitzarla. Es detalla el procés pas a pas i s'indiquen els fitxers obtinguts, des de etiqueta personalitzada fins les pàgines JSP editades. Es mostren pàgines web de ilustració de les explicacions.

Capítol 5: Treure les conclusions sobre tot el projecte, coneixements assolits dels frameworks, experiència en l'el·laboració de l'aplicació amb la integració de frameworks i finalment resultats sobre la implementació d'un validador i possible metodologia per altres casos semblants.

----------^----------

Capítol 2: Conèixer els *frameworks*

1.- Introducció.-

En la creació d'una aplicació J2EE intervenen els components i contenidors, però volem centrar-nos en les parts que els formen: fitxers Java, XML, JSP, scripts SQL, etc. Arribats a aquest punt hem de introduir el **patró MVC-model 2**. El model-2 permet implementar la capa de presentació mitjançant un servlet controlador entre les peticions client i les respostes JSP. Les capes de negoci i presentació es poden implementar amb objectes distribuïts EJB.

Així mateix assenyalar que en la creació d'una aplicació J2EE utilitzarem els ***frameworks***. En el Model-2 tenim un conjunt de tasques repetitives comunes per a totes les aplicacions J2EE. Aleshores aquestes tasques es podrien implementar en un *framework* per a la capa web o presentació que utilitzin tots els desenvolupaments i així no tenir-les que refer en cada aplicació. Igualment de convenient serà emprar un *framework* per implementar les capes de persistència i fins i tot la capa de lògica de negoci.

Finalment volem ressaltar com l'objectiu principal del capítol l'estudi de l'estructura i principals característiques dels *frameworks*, els seus **fitxers de configuració** i com tractar-los per tal de modificar el comportament del *framework*.

----------^----------

2.-Els frameworks.-

Què és i para que serveix un *framework*? En primer lloc un *framework* no és específic d'una aplicació J2EE. En el desenvolupament de programari, com és una aplicació J2EE, un *framework* és una estructura de sosteniment definida per la qual un altre projecte de programari permet ser organitzat i desenvolupat. Bàsicament un *framework* inclou suport de programes, llibreries i un llenguatge de *scripting* entre altres per ajudar a desenvolupar i enllaçar els diferents components d'un projecte.

2.1.- Tipus de frameworks.-

En concret, els desenvolupadors crearan la capa web utilitzant les classes i estenent les interfícies d'un *framework* de presentació. Fonamentalment el *framework* de la capa web desacobla aquesta capa de la de negoci en components separats. Tenim un ampli sortit de *frameworks* per a la capa de presentació. Destacarem els següents:

- ✓ *Struts*
- ✓ *Java Server Faces*
- ✓ *WebWork*

Així mateix els desenvolupadors poden implementar la capa d'integració o persistència per mitjà d'un *framework* de mapeig objecte/relacional. Fonamentalment aquest tipus de *framework* automatitza el procés de creació, lectura, actualització i escriptura d'entitat a les bases de dades relacionals i la majoria d'ells ofereixen una eina perfecta de mapeig objecte/relacional. Destacarem els següents:

- ✓ *Hibernate*
- ✓ *Castor*
- ✓ *Jakarta OJB*

Acabarem aquest apartat d'introducció als tipus de *frameworks* anomenant un dels genèrics més emprats, *Spring*. És un *framework* que mostra una gran flexibilitat i que s'integra perfectament amb els de presentació i persistència i aleshores ofereix suport als desenvolupadors en la creació d'ambdues capes.

3.-Frameworks de presentació.-

A l'apartat anterior hem avançat el perquè de l'ús del *framework* de presentació per part dels desenvolupadors i ara detallarem les seves avantatges:

- ✓ Desacoblament de les dues capes en components separats
- ✓ Simplificació i estandardització de la validació dels paràmetres d'entrada
- ✓ Simplificació de la gestió del flux de navegació de l'aplicació J2EE
- ✓ Ofereix un centre de control
- ✓ Permet una gran reutilització
- ✓ Imposa idèntica arquitectura a tots els desenvolupaments
- ✓ Simplificació de moltes tasques repetitives

A continuació parlarem amb més profunditat de dos dels més coneguts *frameworks* d'aquesta capa: *Struts* i *JavaServer Faces*.

3.1.-Struts.-

Struts és la implementació del patró de disseny **Model-Vista-Controlador** (MVC) per les JSP. *Struts* forma part del projecte Apache Jakarta i és de codi obert. És un *framework* que va perfectament per qualsevol mida d'aplicació.

Per entendre millor el paper d'aquest *framework* observarem la següent figura on es detallen les parts de MVC i tot seguit la explicarem així com el rol que exerceix *Struts*:

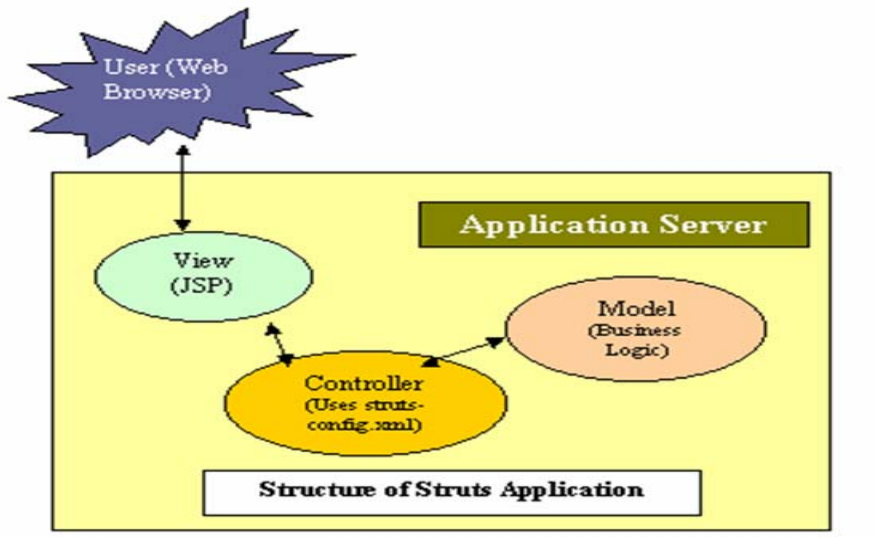


Figura 1.- Estructura de l'aplicació Struts

El controlador (*Controller*) és el intermediari entre el model (*Model*) i la vista (*View*), és responsable de rebre la petició del client. Un cop s'ha rebut la petició s'executarà apropiadament la lògica de negoci des de el model i a continuació es produirà la sortida cap el client mitjançant la vista. Els components bàsics que formen part de *Struts* són: ***ActionServlet***, ***Action***, ***ActionForm*** i ***struts-config.xml***.

Abans d'entrar en la part específica de la tasca dels *Struts* hem de destacar la seva gran potència, està compost per unes 300 classes i interfícies organitzades en uns 12 paquets a nivell superior. També proveeix de les classes i interfícies per a treballar amb el controlador i presentació de dades mitjançant l'ajuda de les llibreries de etiquetes d'usuari.

Bàsicament els *Struts* implementen el Servlet (*ActionServlet*) que actua en qualsevol moment que l'usuari faci una petició, aleshores el Servlet serà el intermediari de la mateixa: intercepta la URL i en base als fitxers de configuració dona el control de la petició a la classe *Action*. La classe *Action* és una part del Controlador i responsable de la comunicació amb la capa model.

3.1.1.- Funcionament de Struts.-

Sense voler oferir un tutorial complet sobre els *Struts* si que considerem necessari explicar el seu funcionament a través dels principals fitxers que intervenen.

1.- Sabem que en tota aplicació J2EE hi ha un descriptor de desplegament habitualment anomenat ***web.xml*** i guardat en el directori *WEB-INF*. El fitxer *web.xml* serà llegit per el contenidor on es desplega l'aplicació. Conté la configuració definida per a la nostra aplicació web que inclou l'índex, la pàgina d'entrada per defecte, el mapeig dels servlets i els paràmetres d'inici dels elements del context entre altra informació. En el *web.xml* hem de configurar el ***ActionServlet***, el component central del Controlador de *Struts*, que manegarà a un mapa determinat totes les demandes fetes per els navegadors web. *ActionServlet* és una extensió de la classe *HttpServlet* i porta a terme les següents tasques:

- ✓ Quan el contenidor s'engega llegeix el fitxer de configuració i ho carrega a memòria (mètode ***init()***)
- ✓ Intercepta la petició HTTP i la manega apropiadament (mètodes ***doGet()*** i ***doPost()***)

2.- En Struts també disposem d'un altre fitxer de configuració habitualment anomenat ***struts.config.xml***, el seu nom pot ser modificat en el fitxer ***web.xml***. El fitxer ***struts.config.xml*** es guarda sota el directori **WEB-INF** de l'aplicació. És un document XML que descriu en part o totalment l'aplicació *Struts*. Conté informació sobre els recursos i configura la seva interacció. S'utilitza per associar els camins dels components del controlador conegudes com classes *Action*, exemple:

```
<action path ="/login" type = "LoginAction">
```

Aquesta etiqueta li diu a *ActionServlet* que quan la petició sigui **http://myhost/myapp/login.do** aleshores ha d'invocar (senyala el camí) al component *LoginAction*. Destacarem el **.do** escrit en la URL, obligats sempre que es rebi una petició amb l'extensió **.do**.

3.- Per a cada acció (*action*) hem de configurar els *Struts* amb els noms de les pàgines que es mostraran com a resultat de l'acció. En l'aplicació pot haver més d'una pàgina per a cada acció. Per exemple una seria resposta reeixida i l'altra d'error. L'acció li diu a *ActionServlet* que ha set reeixida si s'ha pogut complir i el contrari, error, si ha fallat. La lògica de negoci és invocada des de dins dels components del Controlador (***ActionServlet***).

4.- L'acció pot estar associada amb un *JavaBean* en el fitxer de configuració (***struts.config.xml).*** Recordarem que un *JavaBean* no és més que una classe que conté els mètodes **get()** i **set()** per tal de comunicar-se les parts Vista i Controlador de MVC. Aquests *JavaBeans* són validats mitjançant el mètode **validate()** de la classe **ActionForm** i així aconseguir l'ajuda del sistema *Struts*.

Quan el client envia una petició a través d'un formulari normal emprant els mètodes **Get** i **Post** aleshores Struts actualitza les dades en el Bean abans de cridar els components del Controlador.

5.- La vista que nosaltres emprem a *Struts* pot ser JSP però no és obligat. En *Struts* hi ha un bon conjunt d'etiquetes per JSP però també fitxers HTML poden ser inserits en una aplicació amb *Struts*. Ara bé, el *framework Struts* permet mitjançant les etiquetes JSP la creació d'interfícies d'usuari que interactuen de forma elegant amb els beans de la classe ***ActionForm***.

Aquestes etiquetes permeten als desenvolupadors implementar la interacció entre usuari i controlador sense escriure massa codi Java.

A continuació, en la figura 2, mostrem un esquema del procés en el *framework Struts*:

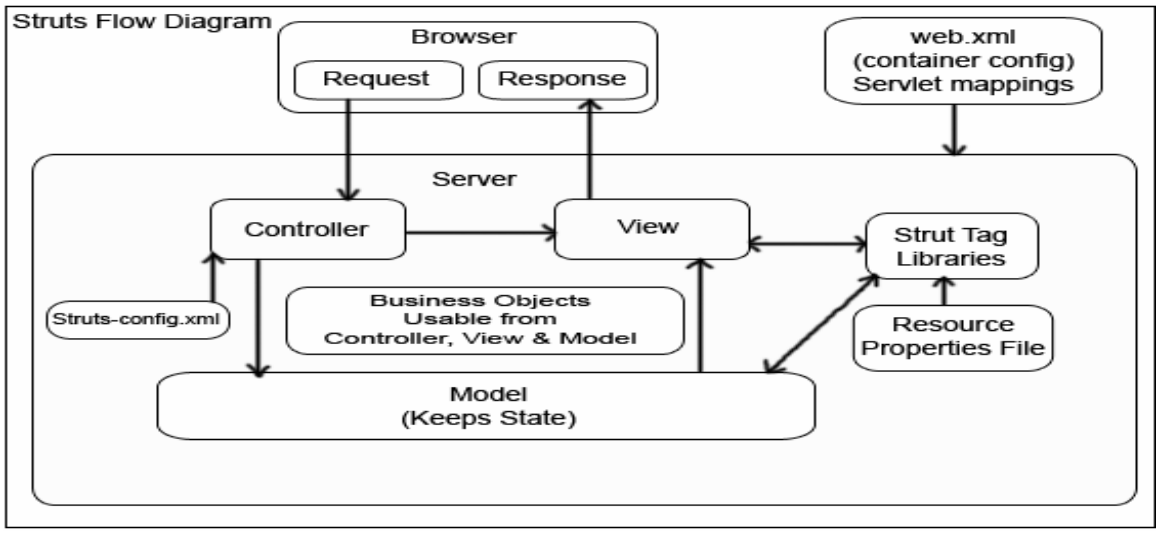


Figura 2.- Diagrama del flux en Struts

La corresponent explicació:

web.xml: Tan bon punt s'engega el contenidor el primer que fa és comprovar el web.xml i determinar quin ***ActionServlet*** existeix. El contenidor és responsable del mapeig de les demandes al correcte ActionServlet.

petició: en aquest pas la petició és interceptada per el Controlador.

controlador: el centre del contenidor. La majoria d'aplicacions amb *Struts* només tenen un controlador que és el ***ActionServlet***, responsable per encaminar diferents accions. El controlador determina quina acció es requereix i envia la informació per tal de ser processada per un ***Bean***.

struts.config.xml: és el fitxer de configuració per emmagatzemar el mapeig de les accions. Si s'empra aquest fitxer no cal massa codi per implementar el mòdul cridat dins d'un component. Altre responsabilitat del controlador és comprovar el *struts.config.xml* per tal de determinar quin mòdul ha de ser cridat després de la petició. *Struts* només llegeix el *struts.config.xml* a l'inici.

Model : és bàsicament la part de la lògica de negoci que pren la petició de l'usuari i guarda el resultat durant tot el procés. És el lloc on es porta a terme la preparació del processament de les dades rebudes en la resposta. És possible reutilitzar la mateixa plantilla per diferents pàgines de resposta. *Struts* proporciona les classes ***ActionForm*** i ***Action*** que poden tenir extensions per a crear els objectes del model.

Vista : en *Struts* la vista és principalment una pàgina ***JSP*** responsable de la sortida cap l'usuari

Llibreries d'etiquetes Struts: ajuden a integrar els components de *Struts* dins de la lògica del projecte. Les etiquetes són emprades dins de les ***JSP***. Això significa que tant el controlador com el model no poden utilitzar aquestes llibreries però en canvi empren la llibreria de classes per el control del procés.

fitxer Property: per emmagatzemar els missatges que empraran els objectes i pàgines. Són utilitzats per guardar títols i altres dades en format *String*.

objectes negoci: on són les regles de la lògica de negoci. Són els mòduls que tot just regulen les activitats del dia a dia.

resposta: la sortida de l'objecte JSP de la vista.

3.1.2.-Controlador de Struts.-

Per acabar aquesta breu descripció de Struts i sense estendre-nos tot i que el tema dona per molt més farem una completa explicació de la part central dels *Struts*: el **controlador**. També proporcionarem unes pautes per tal d'implementar el fitxer clau del controlador: ***struts-config.xml***

El controlador *ActionServlet* se configura com Servlet en el ***web.xml*** tal com es mostra a continuació (observar la línia de color vermell):

```
<!-- Standard Action Servlet Configuration (with debugging) -->
<servlet>
    <servlet-name>action</servlet-name>
     <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
     <init-param>
       <param-name>config</param-name>
       <param-value>/WEB-INF/struts-config.xml</param-value>
     </init-param>
    <init-param>
       <param-name>debug</param-name>
       <param-value>2</param-value>
       </init-param>
       <init-param>
       <param-name>detail</param-name>
       <param-value>2</param-value>
    </init-param>
   <load-on-startup>2</load-on-startup>
</servlet>
```

Com s'ha dit abans, el Servlet és responsable de manegar tota petició en la plataforma dels Struts, l'usuari pot mapejar l'específic patró de petició al *ActionServlet* , l'etiqueta ***<servlet-mapping>*** en el fitxer ***web.xml*** especifica el patró URL per a ser manegat per el servlet. Per defecte és ****.do***, però pot ser canviat per qualsevol altre. El següent codi de ***web.xml*** ens mostra el mapeig:

```
<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
   <servlet-name>action</servlet-name>
   <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

3.1.3.-Fitxer struts-config.xml.-

El mapeig encamina totes les demandes acabades en **.do** al *ActionServlet* que utilitza la configuració definida a ***struts-config.xml*** per a decidir la destinació de la petició. Les definicions (veure més endavant) són emprades per manejar qualsevol acció. Un exemple que mostrarem serà la pàgina ***Welcome.jsp*** que rebrà el mapeig de la petició *Welcome.do*:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html locale="true">
<head>
   <title><bean:message key="welcome.title"/></title>
   <html:base/>
</head>
  <body bgcolor="white">
  <h3><bean:message key="welcome.heading"/></h3>
  <p><bean:message key="welcome.message"/></p>
</body>
</html:html>
```

"*Action Mapping Definitions*" és l'apartat més important de ***struts-config.xml***. És la secció que agafa un formulari definit en la secció anomenada "*Form Bean Definitions*" i la mapeja cap a una classe d'acció.

El codi següent sota l'etiqueta ***<action-mappings>*** serveix per a encaminar la petició a la pàgina ***Welcome.jsp***:

```
<action  path="/Welcome"
     forward="/pages/Welcome.jsp"/>
```

Per a invocar el fitxer ***Welcome.jsp*** utilitzarem el codi següent:

*<**html:link** page="**/Welcome.do**">Primera petició al controlador</html:link>*

Un cop l'usuari clica en l'enllaç Primera Petició al controlador en la pàgina índex, la petició (per *Welcome.do*) s'envia al Controlador i la porta cap a la pàgina ***Welcome.jsp***. Finalment el contingut de ***Welcome.jsp*** es mostrarà a l'usuari.

----------^----------

3.2.- JavaServer Faces.-

JavaServer Faces (JSF) és un *framework* per la creació de interfícies d'usuari en aplicacions web. JSF resulta una magnífica combinació de *Swing* (un *framework* de interfícies d'usuari en estàndard Java per aplicacions de PC) i *Struts* (un *framework* per aplicacions web basat en JSP). Igual que *Struts*, JSF proveeix de la gestió del cicle de vida (*veure següent apartat*) de l'aplicació web mitjançant el seu *servlet* controlador i com *Swing*, JSF proveeix d'un complet component de model per manipulació dels esdeveniments i intèrpret de components.

De manera resumida direm que JSF facilita la tasca del desenvolupador d'aplicacions web perquè:

- ✓ Permet la creació de interfícies d'usuari a partir de conjunts d'estàndard i reutilitzables components de la part del servidor
- ✓ Proveeix d'un conjunt d'etiquetes JSP per accedir als components anteriors
- ✓ De forma transparent guarda informació de l'estat i reomple els formularis quan aquests són de nou mostrats
- ✓ Proveeix d'un *framework* per implementar components personalitzats
- ✓ Encapsula la manipulació d'esdeveniments i interpretació de components de tal manera que podem utilitzar components JSF estàndard o personalitzats per suportar llenguatges de marques que no siguin HTML

3.2.1.- Tecnologia JSF

La tecnologia JavaServer Faces (JSF) és un framework de interfícies d'usuari del costat del servidor per aplicacions Web bassades en tecnologia Java. Els principals components de la tecnologia JSF son:

- ✓ Un API i una implementació de referència per tal de: representar ***components UI*** (*) i manipular el seu estat; manipulació d'esdeveniments, validació del costat del servidor i conversió de dades;definir la navegació entre pàgines;suportar internacionalització i accessibilitat; i proporcionar extensibilitat per a totes aquestes característiques.
- ✓ Una llibreria d'etiquetes JavaServer Pages (JSP) personalitzades per dibuixar components UI en una pàgina JSP.

Tant el model de programació com la llibreria d'etiquetes per components UI faciliten significativament la tasca de construcció i manteniment d'aplicacions web amb els UI del costat del servidor. Podem amb poc esforç:

- ✓ Connectar esdeveniments generats en el client a codi de la aplicació en el costat del servidor.
- ✓ Mapejar components UI a una pàgina de dades del costat del servidor.
- ✓ Construir un UI a partir de components reutilitzables i extensibles.
- ✓ Gravar i restaurar l'estat del UI més enllà de la vida de les peticions de servidor.

En la figura 3 es veu que la interfície d'usuari creada mitjançant tecnologia JSF (representat per *myUI*) s'executa en el servidor i és construïda i mostrada en el client.

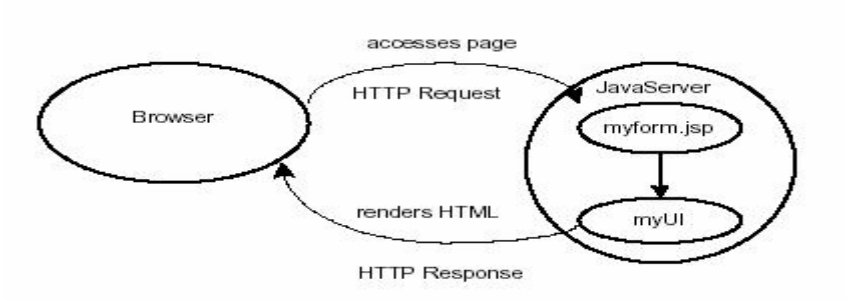


Figura 3.- Tecnología JSF per interfície d'usuari

La pàgina ***myform.jsp***, dibuixa els components de l'interfície d'usuari amb etiquetes personalitzades definides por la tecnologia JSF. El UI de la aplicació Web (representat per **myUI** en la figura) manipula els següents objectes referenciats per la pàgina JSP:

- ✓ Els objectes components que mapejan les etiquetes sobre la pàgina JSP.
- ✓ Els escoltadors d'esdeveniments, validadors, i els convertidors registrats en els components.
- ✓ Els objectes del model que encapsulen les dades i les funcionalitats dels components específics de l'aplicació.

(*)**UI** és l'acrònim per *User Interface* o interfície d'usuari. Hi ha dos principals tipus de **components UI**, uns que inicialitzen una acció, com per exemple un botó, i altres que proporcionen dades com és un camp d'introducció de text.

3.2.2.- Cicle de vida de JSF.-

JSF manipula les peticions HTTP amb sis diferents fases com es mostren a continuació:

1. Restaurar la vista
2. Aplicar els valors de la petició; processar esdeveniments
3. Processar validacions; processar esdeveniments
4. Actualitzar els valors del model; processar esdeveniments
5. Cridar l'aplicació; processar esdeveniments
6. Construir-mostrar resposta

La següent figura ens mostra tots els passos del cicle de vida petició-resposta del *framework JavaServer Faces*:

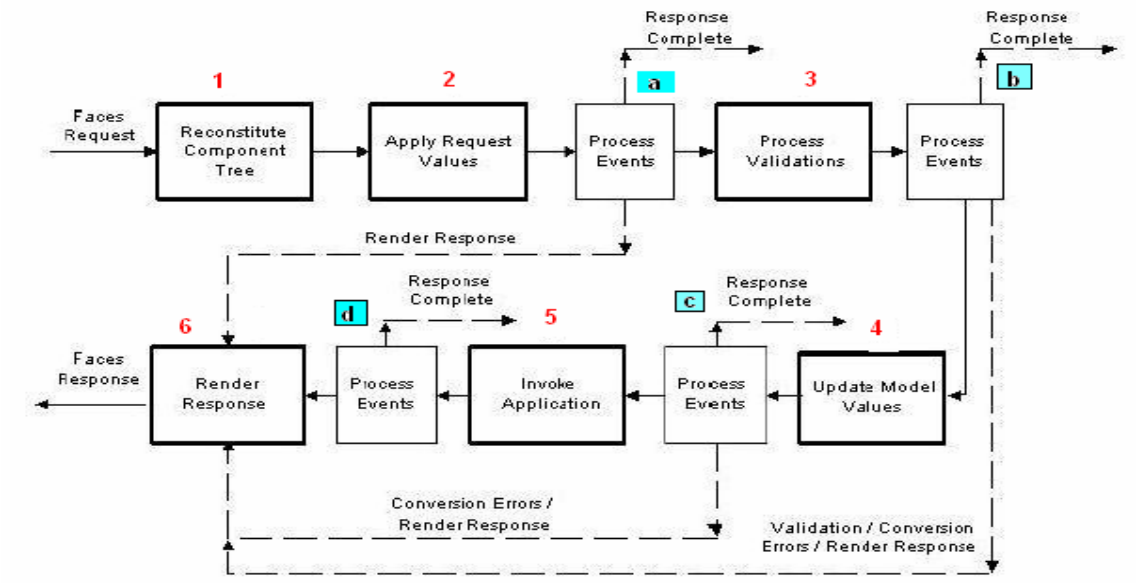


Figura 4.- Cicle de vida petició-resposta de JSF

Primer de tot dir que la majoria dels usuaris de JavaServer Faces no necessitaran conèixer a fons el cicle de vida de processament d'una petició. Ara bé, si sabem que és el que fa la tecnologia JSF a l'hora de processar una pàgina, el desenvolupador ja no li cal que es preocupi dels problemes de *rendering* (vocable difícil de traduir i que podríem acceptar com construir o muntar) associats amb altres tecnologies UI. Per exemple en el canvi d'estat dels components individuals. Per exemple si la selecció d'un *checkbox* afecta a l'aparença d'un altre component de la pàgina, la tecnologia JSF manipularà el esdeveniment apropiadament sense permetre obviar el canvi en el dibuix de la pàgina.

Anem a explicar la figura 3 amb el detall de les sis fases:

(1) Restaurar vista - Reconstituir l'arbre de components.- Sempre que s'invoca una pàgina JSF com és el cas de prémer un botó o un enllaç, JavaServer Faces inicia l'estat ***Reconstituir l'arbre de Components***. Durant la fase JSF construeix l'arbre de components de la pàgina demanada, connecta els manipuladors d'esdeveniments i guarda l'estat en el **FacesContext** (*). L'arbre de components d'una pàgina tipus *welcome.jsp* de l'aplicació *EndivinaNumero* podria semblar aquest:

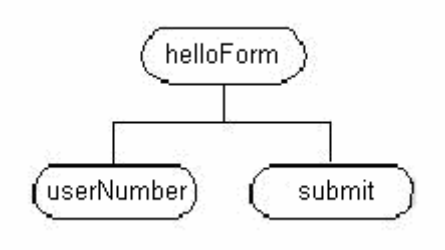


Figura 5.- Arbre de components

(*) **FacesContext** conté tota la informació de l'estat de la petició relacionada amb el processament d'una simple petició JSF, i la construcció de la corresponent resposta.

(2) Aplicar els valors de la petició.- Un cop construït l'arbre de components, cadascun d'ells extrau el seu nou valor a partir dels paràmetres de la petició amb el mètode ***decode*** (). Aleshores el nou valor és guardat de forma local en el component. Si falla la conversió del valor, es genera un missatge d'error associat al component i es col·loca a la cua de *FacesContext*. Aquest missatge es mostrarà durant la fase 6 ***Construir la Resposta***, junt amb qualsevol error de validació resultant en la fase 3 ***Processar Validacions***.

Si durant aquesta fase es produeix un esdeveniment, la implementació JSF emet aquest esdeveniment cap els escoltadors interessats (*Process events*).

En aquest punt, si l'aplicació necessita redirigir-se a un recurs d'aplicació Web diferent o generar una resposta que no contengui components JSF, pot cridar a ***FacesContext.responseComplete*** (a).

En el cas del component *userNumber* de la pàgina welcome.jsp, el valor pot ser qualsevol que l'usuari posi en el camp. Com la propietat de l'objecte de model associada al component té un tipus *Integer*, aleshores JSF converteix el *String* a *Integer*.

Finalment ara s'han actualitzat els nous valors en els components i tant els missatges com els esdeveniments s'han situat a les corresponents cues.

(3) Processar validacions.- Durant aquesta fase, la implementació JSF processa totes les validacions registrades amb els components de l'arbre. Examina els atributs del component especificats per las regles de validació i les compara amb el valor local emmagatzemat en el component. Si el valor local no es vàlid, la implementació JSF afegeix un missatge d'error al ***FacesContext*** i el cicle de vida va directament a la fase 6 ***Construir la Resposta*** per tal que la pàgina sigui muntada de nou amb el missatge d'error inclòs. Igualment es mostraran els possibles missatges d'error de la fase anterior ***Aplicar els Valors de la Petició***.

Ara, si l'aplicació necessita redirigir-se a un recurs d'aplicació Web diferent o generar una resposta que no contengui components JSF, pot cridar a ***FacesContext.responseComplete*** (b).

Si durant aquesta fase es produeix un esdeveniment, la implementació JSF emet aquest esdeveniment cap els escoltadors interessats (*Process events*).

En la pàgina *welcome.jsp*, la implementació JSF processa el validador sobre l'etiqueta *input_number* de *UserNumber*. Verifica que la dada introduïda per l'usuari en el camp de text és un enter entre 0 i 10. Si la dada no és vàlida, o s'ha produït un error de conversió durant la fase anterior ***Aplicar els Valors de la Petició***, el processament va directament a la fase 6 ***Construir la Resposta*** per tal que la pàgina *welcome.jsp* sigui muntada de nou amb els missatges d'error inclosos en el component associat amb l'etiqueta *output_errors*.

(4) Actualitzar els valors del model.- Un cop que la implementació JSF determina que la dada és vàlida, pot entrar en l'arbre de components i configurar els valors de l'objecte de model corresponent amb els valors locals dels components. Això si, només s'actualitzaran els components amb l'expressió ***valueRef***. Si la dada local no es pot convertir als tipus especificats per les propietats de l'objecte del model, el processament va

directament a la fase 6 ***Construir la Resposta*** per tal que la pàgina *welcome.jsp* sigui muntada de nou amb els missatges d'error inclosos.

Ara, si l'aplicació necessita redirigir-se a un recurs d'aplicació Web diferent o generar una resposta que no contengui components JSF, pot cridar a ***FacesContext.responseComplete*** (c).

Si durant aquesta fase es produeix un esdeveniment, la implementació JSF emet aquest esdeveniment cap els escoltadors interessats (*Process events*).

En aquesta fase, a la propietat *userNumber* de ***UserNumberBean*** se li dona el valor del component *userNumber*.

(5) Cridar l'aplicació.- Durant aquesta fase, la implementació JSF manipula qualsevol esdeveniment a nivell d'aplicació, per exemple enviar un formulari o enllaçar a un altre pàgina.

Ara, si l'aplicació necessita redirigir-se a un recurs d'aplicació Web diferent o generar una resposta que no contengui components JSF, pot cridar a ***FacesContext.responseComplete*** (d).

La pàgina *welcome.jsp* de l'exemple *endivinaNumero* té associat un esdeveniment a nivell d'aplicació amb el component *Command*. Quan l'esdeveniment és processat, una implementació per defecte de ***ActionListener*** recupera la sortida, "success", des de l'atribut ***action*** del component. L'escoltador (listener) passa la sortida a ***NavigationHandler*** per defecte, i aquest últim compara la sortida amb les regles de navegació definides en el fitxer de configuració (*faces-navigation.xml* o *faces-config.xml*) de l'aplicació per tal de determinar quina pàgina s'haurà de mostrar a continuació.

Llavors, la implementació JSF configura en la nova pàgina l'arbre de components de la resposta. Finalment, la implementació JavaServer Faces traspassa el control a la fase 6 ***Construir la Resposta***.

(6) Construir la resposta.- Durant aquesta fase, la implementació JSF invoca les propietats de codificació dels components i dibuixa els components de l'arbre de components guardat en el ***FacesContext***.

Si al llarg de les fases 2, 3 i 4, es van trobar errors, aleshores es dibuixarà la pàgina original. Si les pàgines contenen etiquetes ***output_errors***, qualsevol missatge d'error que estigui a la cua serà mostrat en la pàgina original.

Es poden afegir nous components en el cas que l'aplicació inclogui constructors (renderers) personalitzats que defineixen la forma de

construir un component. Un cop s'ha construït el contingut de l'arbre, aquest es guarda per tal que les següents peticions tinguin accés i estigui disposat per la fase 1 ***Reconstituir l'arbre de Components*** de les futures crides.

En el Capítol 3 tornarem a parlar de ***JavaServer Faces*** i en especial sobre els seus fitxers de configuració.

----------^----------

4.-Frameworks de persistència.-

El *framework* de persistència dirigeix i administra la base de dades i el mapeig entre aquesta i els objectes.

En la capa de persistència o també anomenada d'integració es disposa de moltes opcions per implementar-la, entre les quals tenim:

- POJO
- EJB d'entitat
- *Frameworks* de *mapeig* objecte/relacional

Com l'objecte del present document són els *frameworks* passarem directament a parlar dels anomenats de *mapeig* objecte/relacional.

Constitueixen una de les més actuals alternatives per la implementació de la persistència en una aplicació J2EE. Entre les seves característiques destacarem:

✓ Automatitzen els processos de creació, lectura, actualització i escriptura (CRUD) d'identitat a les bases de dades relacionals
✓ La majoria ofereixen una eina completa de mapeig objecte/relacional
✓ En moltes ocasions basen la seva arquitectura en el POJO

Entre les seves avantatges tenim:

- ✓ Facilitat d'ús
- ✓ Persistència transparent d'entitats
- ✓ Moltes optimitzacions de rendiment
- ✓ Generació automàtica de les classes Java que representen les entitats
- ✓ Generació automàtica de les relacions a partir de fitxers de definició

Hem de lamentar una gran desavantatge: no és una tecnologia estàndard.

Precisament per eliminar aquesta desavantatge apareix una de les opcions descrites al començament de l'apartat, els EJB d'entitat. Sembla que amb l'aparició de la nova especificació EJB 3.0 la tecnologia EJB podrà cobrir les funcionalitats o característiques detallades abans però de manera estàndard.

A continuació parlarem amb més profunditat d'un dels més coneguts *frameworks* d'aquesta capa: ***Hibernate***.

4.1.-Hibernate.-

És un potent *framework*, amb gran qualitat de implementació de la persistència objecte/relacional i servei de cerca de dades per Java. *Hibernate* ens facilita el desenvolupament de classes persistents seguint el llenguatge comú de Java, amb la inclusió d'associació,herència, polimorfisme, composició i el *framework* de les col·leccions de Java. El llenguatge *Hibernate* per cerques (HQL), definit com una extensió mínima del SQL orientat a objectes, ens proveeix d'un elegant pont entre els objectes i els mons relacionals. *Hibernate* també ens permet d'expressar cerques mitjançant SQL nadiu o en base Java. *Hibernate* és encara avui dia la solució de **mapeig objecte/relacional** més emprada per Java.

Hibernate mapeja les classes Java a les taules de la base de dades. També proveeix de cerca de dades (*query*) i de facilitats per recuperació de les mateixes que redueix el temps de desenvolupament de forma considerable. No és només el més utilitzat per aplicacions centrades en les dades que sols fa servir els mecanismes d'emmagatzematge per implementar la lògica de negoci en la base de dades sinó que també és el més útil en la capa mitja amb llenguatge Java.

Hibernate també permet persistència transparent i dota a les aplicacions de capacitat per controlar qualsevol base de dades. Pot ser emprat en aplicacions Java Swing, en les que hi ha Servlet o J2EE amb EJB de sessió.

A continuació entrarem en l'arquitectura de *Hibernate* i començarem per el següent diagrama que representa la seva arquitectura al més alt nivell:

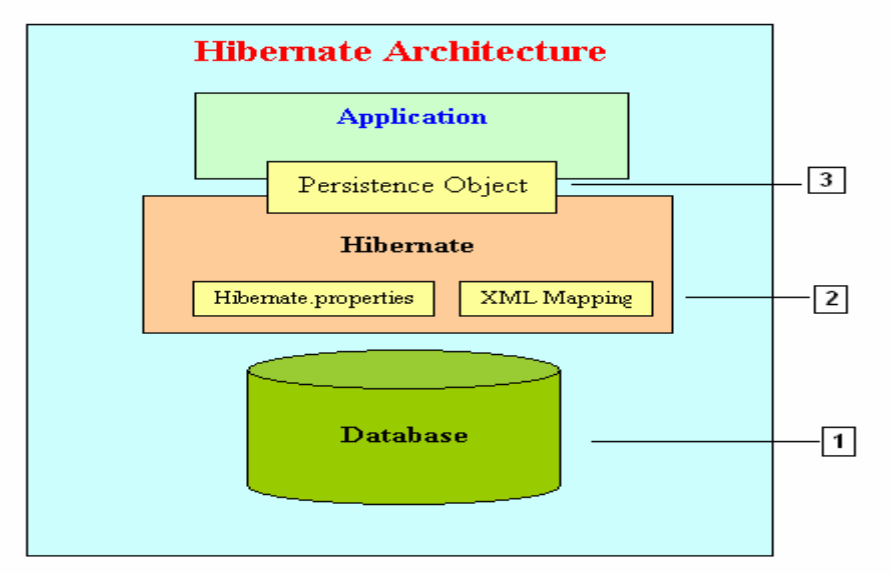


Figura 6.- Arquitectura de Hibernate

El diagrama mostra com *Hibernate* utilitza la base de dades (1) i les dades de configuració (2) per tal de proveir a l'aplicació de serveis de persistència i a la vegada d'objectes persistents (3).

Per tal d'utilitzar *Hibernate* es requereix la creació de classes Java que representin la taula de BD i així mapejar les variables de l'instancia en la classe amb les corresponents columnes en la BD. Aleshores *Hibernate* pot ser utilitzat per desenvolupar operacions en la BD com ***select, insert, update*** i ***delete*** els registres en la taula. *Hibernate* de manera automàtica crea el procés de qüestionar (*query*) per dur a terme aquestes operacions.

L'arquitectura *Hibernate* té 3 components principals:

✓ **Control de connexions:** el servei de control de connexions proveeix d'un eficient gestor de les connexions amb la BD que constitueixen la part més cara de la interacció amb la BD ja que requereix d'una gran despesa de recursos per obrir i tancar cada connexió

- ✓ **Control de transaccions:** el servei de control de transaccions proveeix a l'usuari de la capacitat d'execució de més d'una expressió a la vegada.
- ✓ **Mapeig Objecte/relacional:** és la tècnica de mapejar la representació de les dades des d'un model d'objectes a un model de dades relacional. És la part de *Hibernate* utilitzada per executar les comandes ***select, insert, update*** i ***delete*** registres de la taula. Quan passem un objecte al mètode ***Session.save()***, *Hibernat*e llegeix l'estat de les variables de l'objecte i executa la corresponent pregunta (*query*).

El tercer dels components resulta una molt bona eina però no així els dos primers. Aleshores *Hibernate* s'utilitza junt amb altres eines com *Apache DBCP* per tal de millorar el control de connexions i transaccions. Aquest és el motiu de la flexibilitat d'ús de *Hibernate*, l'arquitectura ***'Lite'*** només empra el mapeig O/R i l'arquitectura ***'Full Cream'*** usa tots 3 components.

A continuació mostrarem un senzill programa que inserirà registres en una base de dades MySQL.

4.1.1.- Les interfícies cabdals de Hibernate.-

Les interfícies són el primer que s'ha d'aprendre sobre *Hibernate* de cara a poder utilitzar-lo en la capa de persistència de la nostra aplicació. El objectiu major del disseny del API és mantenir el més estretament possible les interfícies entre els components de programari. La figura 7 il·lustra els rols de les més importants interfícies de *Hibernate* en les capes de lògica de negoci i persistència. La capa de negoci està sobre de la capa de persistència per el fet d'actuar la primera com el client de la segona en una aplicació per capes tradicional.

Les interfícies de la figura 7 es poden classificar de la següent manera:

- ✓ Les cridades per les aplicacions en l'execució de CRUD (*create, read, update, delete*) bàsic i operacions de qüestionar (*query*). Constitueixen el principal punt de la dependència de la lògica de negoci de l'aplicació en *Hibernate*. Ho formen *Session*, *Transaction* i *Query*.
- ✓ Les cridades per el codi de la infraestructura de l'aplicació en ordre a configurar *Hibernate*. Destacar la interfície *Configuration*.
- ✓ Les que responen a la crida i permeten a l'aplicació reaccionar als esdeveniments que es produeixen dins de *Hibernate*. Destacarem *Interceptor*, *Lifecycle* i *Validatable*.

✓ Finalment les que permeten estendre la funcionalitat del poderós mapeig de *Hibernate* i són implementades per el codi de la infraestructura de l'aplicació. En la figura 7 es mostra *UserType* però també hi són *CompositeUserType* i *IdentifierGenerator*.

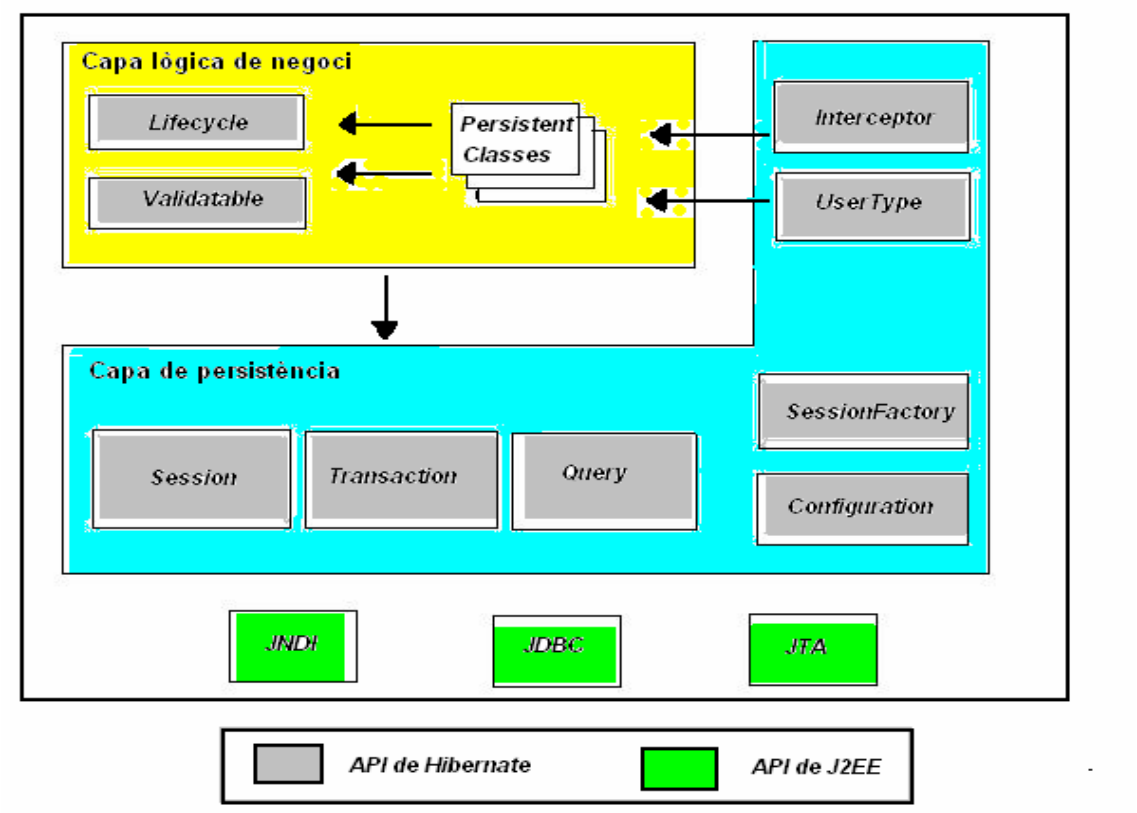


Figura 7.- Arquitectura dels API de Hibernate

Hibernate es serveix dels existents APIs de Java, *JTA, JNDI* i *JDBC*. Aquest últim proveeix d'un rudimentari nivell d'abstracció permeten que gaire bé tota BD amb un ***driver JDBC*** sigui suportada per *Hibernate*. Els dos primers permeten a *Hibernate* integrar-se amb els servidors d'aplicacions J2EE.

4.1.2.- Crear una SessionFactory

Primer de tot avançar que per les dimensions curtes del present treball considerem fonamental reduir també l'espai dedicat a *Hibernate* tot i ser un *framework* capdavanter i bàsic en la implementació de la capa de persistència. Ara bé, no podem obviar una ampla explicació sobre la classe *SessionFactory*.

En ordre a crear una *SessionFactory*, primer crearem una única instància de la classe *Configuration* durant la inicialització de l'aplicació i

utilitzar-la per establir el lloc dels fitxers de mapeig. Una vegada configurada, la instància de *Configuration* és usada per crear *SessionFactory*. Quan aquesta estigui ja creada es pot eliminar *Configuration*. El codi següent inicialitza *Hibernate*:

```
Configuration cfg=new Configuration();
cfg.addResource("hello/Message.hbm.xml");
cfg.setProperties( System.getProperties() );
SessionFactory sessions=cfg.buildSessionFactory();
```

La localització del fitxer de mapeig, *Message.hbm.xml,* està relacionada amb l'arrel del camí de les classes de l'aplicació (*classpath).* Els fitxers de mapeig XML han de situar-se en el *classpath.*

Per convenció, els fitxers de mapeig XML són anomenats per l'extensió *.hbm.xml*. També per convenció és disposar d'un fitxer de mapeig per cada classe. Precisament la documentació de Hibernate recomana que el fitxer de mapeig estigui en el mateix directori que la seva classe. Seguint aquesta convenció podem utilitzar el mètode addClass() i de paràmetre la classe persistent:

```
SessionFactory sessions= new Configuration()
      .addClass(org.hibernate.auction.model.Item.class)
      .addClass(org.hibernate.auction.model.Category.class)
      .addClass(org.hibernate.auction.model.Bid.class)
     .setProperties(System.getProperties() )
     .buildSessionFactory();
```

El mètode *addClass()* assumeix que el nom del fitxer de mapeig té l'extensió *.hbm.xml* i és desplegat junt amb el fitxer de la classe mapejada.

La majoria de les aplicacions només necessiten crear una *SessionFactory*. Si un altre *SessionFactory* fes falta (en cas de varies BD per exemple) aleshores repetiríem el procés. Llavors cada *SessionFactory* treballarà amb la seva BD i un conjunt de mapeigs de classes.

Per descomptat que per configurar Hibernate tenim més a fer que tan sols crear els fitxers de mapeig. Es necessita especificar com obtindrem les connexions de BD tot junt amb una sèrie de *settings* que afecten el comportament de *Hibernate*. Per tal de establir les opcions de configuració disposem de les següents tècniques:

- ✓ Passar una instància de *java.util.Properties* a *Configuration.setProperties().*
- ✓ Establir les propietats del sistema utilitzant *java –Dproperty=value*
- ✓ Situar un fitxer anomenat *hibernate.properties* en el *classpath*
- ✓ Incloure elements <*property*> en el fitxer *hibernate.cfg.xml* situat en el *classpath*

Personalment nos inclinem per les dos últimes tècniques ja que la majoria d'aplicacions necessiten un fix fitxer de configuració. Tant un com l'altre fitxer proveeixen de la mateixa funció: configurar *Hibernate*. Realment triar un d'ells dependrà de les nostres preferències sintàctiques. També és possible combinar ambdues tècniques i disposar *settings* per desenvolupament i per desplegament.

De totes les opcions de configuració les relatives a la connexió amb la BD són les més importants.

4.1.3.-Configuració de Hibernate.-

Es pot utilitzar un fitxer XML per configurar plenament una *SessionFactory*. Diferent de *hibernate.properties*, que conté només els paràmetres de configuració, el ***hibernate.cfg.xml*** pot també especificar la localització dels documents de mapeig. *Hibernat*e utilitza el fitxer ***hibernate.cfg.xml*** per la creació de la connexió i engegar l'entorn correcte. A continuació es mostra un exemple de codi:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory name="java:/hibernate/HibernateFactory">
    <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property
name="hibernate.connection.url">jdbc:mysql://localhost/hibernatetutorial</
property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.connection.pool_size">10</property>
    <property name="show_sql">true</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <!-- Mapping files -->
    <mapping resource="contact.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

En el codi anterior s'especifica que el programa ***hibernatetutorial*** es desplega a ***localhost*** i l'usuari de la BD és root sense contrasenya, així mateix el llenguatge o dialecte és **org.hibernate.dialect.MySQLDialect** per dir-li a *Hibernate* que emprem la base de dades ***MySQL***. És destacable

que *Hibernate* permet moltes bases de dades. La propietat <*mapping resource="**contact.hbm.xml***"/> és el mapeig per la nostra taula *Contact*.

El fet que un dels objectius principals del present document és mostrar els fitxers de configuració així com editar-los per tal de modificar el comportament del framework ens obliga dedicar unes línies a explicar el *hibernate.cfg.xml:*

- ✓ La declaració del tipus de document (*DTD*) és utilitzada per el *parser* de *XML* a l'hora de validar-lo contra la configuració *DTD* de *Hibernate*
- ✓ L'atribut opcional *name* és equivalent a la propietat *hibernate.session_factory_name* i utilitzat en el lligam JNDI de SessionFactory
- ✓ Les propietats poden ser especificades sense el prefix *hibernate*. De totes maneres els noms i valors de les propietats són idènticament programats a altres propietats de configuració
- ✓ Els *mappings* poden ser especificats tant com recursos de l'aplicació com noms codificats de fitxers

Un cop tenim la configuració de Hibernate ja podem inicialitzar-lo:

```
SessionFactory sessions=new Configuration()
        .configure().buildSessionFactory();
```

Com sap *Hibernate* on és guardat el fitxer de configuració?

Quan configure() es crida, *Hibernate* cerca per un fitxer de nom *hibernate.cfg.xml* en el *classpath*. Nosaltres som lliures de utilitzar un altre nom de fitxer XML o tenir-lo en un altre directori. Aleshores hem de indicar-ho al mètode configure():

```
SessionFactory sessions=new Configuration()
        .configure( "/hibernate-
config/auction.cfg.xml").buildSessionFactory();
```

Utilitzar un fitxer XML de configuració és millor que fer anar el *hibernate.properties* o configurar les propietats via programa. Podem, per exemple, utilitzar conjunts de fitxers de mapeig (i diferents opcions de configuració), depenent de la nostra BD i entorn, i fer els canvis d'un a altre programàticament.

En el cas de tenir *hibernate.cfg.xml* i *hibernate.properties* tots plegats en el *classpath*, els *settings* del primer sobreescriuen els establerts en el segon. Aquesta convenció ens permet mantenir certs *settings* bàsics en el *hibernate.properties* i sobreescriure'ls en cada desplegament amb *hibernate.cfg.xml.*

Hem apercebut que *SessionFactory* ha rebut un nom (***name***) en el *hibernate.cfg.xml:*

<session-factory name="java:/hibernate/HibernateFactory">

Hibernate utilitza aquest nom per automàticament lligar el *SessionFactory* a *JNDI* després de la creació.

4.2.-Classe de persistència.-

Hibernate és un dels *frameworks* de persistència que utilitza *POJOs* per mapejar la taula de BD. Podem configurar les variables per manejar la columna de la BD. A continuació es mostra el codi de ***Contact.java***:

```
package lgr.tutorial.hibernate;

/**
 * @author Luis Gallego Ruestes
 * classe Java que mapeja la Taula Contact de la BD MySQL
 */
public class Contact {
  private String firstName;
  private String lastName;
  private String email;
  private long id;

  /**
   * @return Email
   */
  public String getEmail() {
    return email;
  }

  /**
   * @return nom
   */
  public String getFirstName() {
    return firstName;
  }

  /**
   * @return cognom
   */
  public String getLastName() {
    return lastName;
  }
```

```
/**
   * @param string edita el Email
   */
  public void setEmail(String string) {
    email = string;
  }

  /**
   * @param string edita el nom
   */
  public void setFirstName(String string) {
    firstName = string;
  }

  /**
   * @param string edita el cognom
   */
  public void setLastName(String string) {
    lastName = string;
  }

  /**
   * @return ID retorna ID
   */
  public long getId() {
    return id;
  }

  /**
   * @param lD edita el ID
   */
  public void setId(long l) {
    id = l;
  }

}
```

4.2.1.-Mapeig de l'objecte Contact a la taula Contact de la BD.-

El fitxer ***contact.hbm.xml*** s'empra per mapejar l'objecte Contact a la Taula Contact. A continuació es mostra el codi de ***contact.hbm.xml***:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="lgr.tutorial.hibernate.Contact" table="CONTACT">
   <id name="id" type="long" column="ID" >
   <generator class="assigned"/>
  </id>

  <property name="firstName">
     <column name="FIRSTNAME" />
  </property>
```

```
<property name="lastName">
    <column name="LASTNAME"/>
  </property>
  <property name="email">
    <column name="EMAIL"/>
  </property>
 </class>
</hibernate-mapping>
```

4.2.2.-Crear la base de dades a MySQL.-

En el fitxer ***hibernate.cfg.xml*** hem especificat que la base de dades correria en localhost. Aleshores la taula **hibernatetutorial** que crearem s'executarà en localhost ja que el servidor MySQL així ho farà.

4.3.- Desenvolupament de codi per provar Hibernate.-

Ja som en condicions d'escriure el programa per inserir dades a la BD. La sessió *Hibernate* és la principal interfície en l'execució entre l'aplicació Java i *Hibernate*. Primer de tot hem d'aconseguir la sessió *Hibernate* que es crearà mitjançant la classe *SessionFactory* de *Hibernate.org*. *SessionFactory* llegeix la configuració establerta en el fitxer ***hibernate.cfg.xml.*** Aleshores el mètode ***session.save(contact)*** s'utilitza per guardar informació de la taula contact a la BD. A continuació es mostra el codi de la classe de prova de *Hibernate*:

```
package lgr.tutorial.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
/**
 * Exemple Hibernate per inserir dades a la taula contact
 */
public class FirstExample {
  public static void main(String[] args) {
    Session session = null;

    try{
// aquest pas llegirà hibernate.cfg.xml i prepararà hibernate per l'ús
```

```
 SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
      session =sessionFactory.openSession();

//crearà una nova instància de Contact i entrarà les dades en la mateixa mitjançant la
lectura de l'objecte formulari
        System.out.println("Inserint Registre");
       Contact contact = new Contact();
       contact.setId(3);
       contact.setFirstName("Luis");
       contact.setLastName("Gallego");
       contact.setEmail("lgallegor@campus.uoc.es");
       session.save(contact);
       System.out.println("Fet");
    }catch(Exception e){
     System.out.println(e.getMessage());
    }finally{
     // Inserció de dades a contact ocorrerà ara en aquest pas
     session.flush();
     session.close();
     }
 }
}
```

En el Capítol 3 tornarem a parlar de ***Hibernate*** i en especial sobre la seva integració amb ***Spring*.**

----------^----------

5.- Frameworks d'aplicació

Un ***framework de aplicació web*** és un *framework* de programari dissenyat per donar suport al desenvolupament de webs dinàmiques, aplicacions i serveis web. L'objectiu del *framework* és alleugerar el sobrecost que va associat a les activitats comunes en el desenvolupament d'una web. Per exemple, molts *frameworks* proveeixen de llibreries per l'accés a BD, gestors de sessions, i que tot sovint aprofiten el codi d'altres aplicacions. En els capítols anteriors hem vist els *frameworks* per les capes de presentació i integració.

Quan diem ***framework d'aplicació*** volem expressar un tipus de *framework* que a diferència dels que hem vist per les capes de persistència i presentació ajuda a estructurar tota l'aplicació, des de la construcció de pàgines fins els accessos a les BD. Entre els *frameworks* d'aplicació trobem uns de poc coneguts com *OMS Java* i el més famós que és ***Spring***.

5.1.- Introducció a Spring.-

Spring és un ***framework d'aplicació*** (***application framework***) que vol dir que ajuda a estructurar tota l'aplicació a diferència de *Struts* o *Hibernate* que són només per una capa. ***Spring*** és un *framework* de codi obert per aplicacions J2EE en plataforma Java.

Introduir *Spring* és molt més difícil que fer-ho per els altres *frameworks* a causa de no constituir una tecnologia d'un únic propòsit. *Spring* pot ser imaginat com un enorme *framework* de les millors pràctiques per gaire bé totes les àrees de desenvolupament de programari Java. Des de el desenvolupament dels *POJO*s (Plain-Old-Java-Object), al desenvolupament d'aplicacions web, desenvolupament d'aplicacions empresarials,gestió de la capa de persistència, i la programació de l'aspecte orientat (*AOP* *) *Spring* ho suporta tot i ho fa amb el millor i ben dissenyat i durament provat codi de Java. Com parlarem a continuació de les tècniques de la inversió de control i injecció de dependències abans direm que es fan servir per el desenvolupament de *POJO*s.

L'arquitectura en capes de *Spring*, veure figura 8, ofereix enorme flexibilitat. Tota la funcionalitat està construïda sobre els nivells inferiors. Així per exemple es pot utilitzar la gestió de configuració basada en *JavaBeans* sense haver d'utilitzar el patró *MVC* o el suport *AOP*.

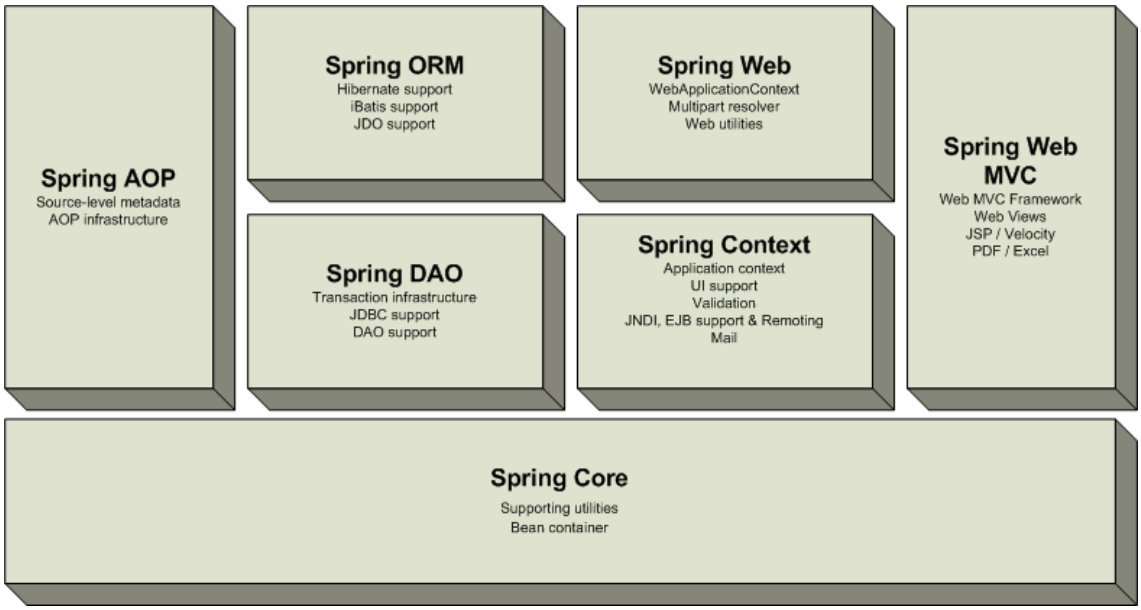


Figura 8. Spring: Arquitectura en capes

(*) Aspect-Oriented Programming (*AOP*) complementa *Object-Oriented Programming* (OOP) proveint d'un altre mitjà de idear l'estructura del programa. A més de les classes, AOP nos dona ***aspectes***. Els aspectes permeten modular els assumptes com la gestió de les transaccions que va en contra dels tipus múltiples i objectes. *Spring AOP* està implementat en Java. No necessita cap procés especial de compilació i tampoc li cal controlar la jerarquia del carregador de classes, i està disponible en un contenidor J2EE o en el servidor de l'aplicació.

5.2.- Inversió de control a Spring.-

La tecnologia que més identifica a *Spring* és la ***Inversió de Control (IoC)***, i més específicament la ***Injecció de dependències*** la distintiva qualitat de la Inversió de Control. Tot sovint es considera *Spring* com un contenidor de IoC quan realment és molt més.

Per millor entendre que és IoC considerarem una tradicional llibreria de classes: el codi de l'aplicació és responsable de tot el flux de control, cridant a la llibreria de classes quan és necessari. Amb IoC, el codi del *framework* invoca el codi de l'aplicació, coordinant tot el flux, en comptes que sigui el codi de l'aplicació qui invoqui el codi del *framework*.

Abans de parlar sobre Injecció de dependències farem cinc cèntims sobre el concepte de dependència aquí, una espècie de lligam entre les classes. Les dependències entre classes es donen quan una classe utilitza una altra classe per a realitzar una determinada acció.

Injecció de dependències es basa en construccions sobre llenguatge Java, més que en l'ús de interfícies específiques del *framework*. En comptes de codi d'aplicació utilitzant els API del *framework* per resoldre dependències de l'estil de paràmetres de configuració i objectes de col·laboració, les classes de l'aplicació exposen les seves dependències mitjançant mètodes o constructors que el *framework* pot cridar per mitjà dels apropiats valors en temps d'execució, basats en la configuració.

La inversió de control és una tècnica que permet minimitzar les dependències entre classes en una aplicació. Per a fer-ho, aporta més control sobre aquestes dependències facilitant formes de configurar-les i canviar-les en temps d'execució. La inversió de control requereix la utilització d'una tècnica anomenada ***Programació contra interfícies***. S'anomena programació contra interfícies al fet de cridar als mètodes d'una interfície en comptes de fer-ho d'una classe concreta.

5.2.1.- Exemple de Inversió de Control.-

Com hem avançat en l'apartat anterior Spring és un *framework* d'inversió de control per injecció de dependències. Facilita la obtenció d'implementacions eliminant tot el codi necessari per a fer-la dels objectes de les nostres aplicacions.

Primer de tot suposem que volem implementar una classe per emmagatzemar les dades d'una aplicació. Una primera aproximació seria fer-ho, com es mostra a l'exemple següent, creant una classe *Persistencia* que implementi un mètode *guardarDades(Dades objecteDades)*.

```
class Persistencia{
    public static void guardarDades(Dades objecteDades){
        //Implementació
        }
    }
```

L'aplicació hauria d'utilitzar aquesta classe de la forma que es mostra al programa

```
class Aplicació{
    public void guardarDades(Dades objecteDades){
        Persistencia.guardarDades(objeceDades);
    }
}
```

Utilitzar aquesta aproximació fa que únicament tinguem una manera d'emmagatzemar les dades de la nostra aplicació, tot i que sempre tindrem la possibilitat de canviar la implementació de la classe *Persistencia* per a canviar el seu comportament.

Però que passa si volem disposar de diverses formes d'emmagatzemar la informació i decidir quina utilitat en temps d'execució?

Aquest requisit s'anomena ampliació per connectors (*plug-ins*). L'aplicació facilita la implementació per defecte, a vegades buida, d'una funcionalitat, permet enganxar noves implementacions i escollir quina s'utilitza en temps d'execució, ja sigui mitjançant fitxers de configuració o a elecció de l'usuari.

Per a afegir aquest comportament a l'aplicació anterior, caldrien tres passos:

- ✓ hauríem de transformar la classe *Persistencia* en una interfície,
- ✓ deixar que l'aplicació utilitzi aquesta interfície per a cridar al mètode *guardarDades(Dades objecteDades)*,
- ✓ i facilitar-li una forma d'obtenir diverses implementacions de la interfície.

A continuació veiem la interfície *Persistencia* i dos implementacions diferents d'aquesta.

```
interface Persistencia{
    public void guardarDades(Dades objecteDades);
}
class PersistenciaFitxers implements Persistencia{
    public void guardarDades(Dades objecteDades){
        //Implementació
    }
}
class PersistenciaBaseDades implements Persistencia{
    public void guardarDades(Dades objecteDades){
        //Implementació
    }
}
```

Exemple: Codi de la classe *Aplicacio* amb Spring

```
public class Aplicacio {
      Persistencia persistencia;

      public Persistencia getPersistencia(){

            return persistencia;
      }

      public void setPersistencia(Persistencia persistencia){
```

```
        this.persistencia = persistencia;
    }

    public void guardarDades(String dades){
        persistencia.guardarDades("Dades importants");
    }
```

L'únic que cal fer per a configurar *Spring* és facilitar-li un fitxer de configuració que indiqui quina implementació de la interfície *Persistencia* utilitzarà la classe *Aplicacio*.

Exemple: Fitxer de configuració d'*Spring* que definirem en format XML i de nom ***applicationContext.xml***:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="fitxer" class="PersistenciaFitxers"/>
   <bean id="bd" class="PersistenciaBaseDades"/>
   <bean id="aplicacio" class="Aplicacio">
      <property name="persistencia">
         <ref bean="fitxer"/>
      </property>
   </bean>
</beans>
```

Ara, crearem una classe per a cridar a *Aplicacio* que serà la que farà ús d'Spring

Exemple: Classe *CridaAplicacio*, crida a l'aplicació injectant les dependències amb Spring

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class CridaAplicacio {

    public static void main(String [] args){
        ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("aplicacio.xml");

        Aplicacio app = (Aplicacio)ctx.getBean("aplicacio");
        app.guardarDades("DadesAGuardar");
    }
}
```

En el següent Capítol 3 tractarem *Spring* des de el vessant de *framework* de suport en la implementació de la capa de lògica de negoci i la integració amb els *frameworks* de les altres dos capes, presentació i persistència.

6.-Fitxers de configuració d'un framework.-

Modificar un *framework* vol dir en aquest context canviar el seu comportament ja sigui variant alguna de les funcions o afegint una de nova. Un *framework* no és una estructura única, constant i immodificable. Disposa d'una sèrie de ***fitxers de configuració*** que li donen flexibilitat i adaptació a canvis i noves funcionalitats.

Tota aplicació web empra un **descriptor de desplegament**, document XML anomenat *web.xml*, que inicialitza els recursos com *servlets* i *taglibs*. De igual forma el *framework* utilitza un **fitxer de configuració** per inicialitzar els seus propis recursos que inclouen:

- ✓ *Interceptors* de les demandes (*requests*)
- ✓ *Action classes* que criden la lògica de negoci i el codi d'accés a les dades
- ✓ *Results* per tal de preparar vistes com *JavaServer Pages*

En temps d'execució només hi ha una configuració per aplicació. Però abans de l'execució la configuració es defineix mitjançant un o més documents XML. Hi ha molts elements per configurar, paquets, espais de noms, accions, excepcions, etc.

Aleshores les configuracions i els documents XML que guardaran aquestes configuracions seran les corresponents a elements com

- ✓ *Beans*
- ✓ Constants
- ✓ Paquets
- ✓ Espais de noms
- ✓ Inclusions

I també

- ✓ Interceptors
- ✓ Accions
- ✓ Resultats
- ✓ Excepcions

6.1.- Configuració de constants a Struts

Hem triat el cas de la ***configuració de les constants*** per considerar-lo clarificador i suficient. Així mateix ho mostrarem en *Struts*, un *framework* força emprat i introduït en l'apartat 4.1 (p. 5), que pot ser configurat a partir dels fitxers ***struts.xml*** i ***struts.properties***.

En *Struts* les constants proveeixen d'un senzill mode d'adaptar a l'usuari una aplicació mitjançant la definició de importants contextos i maneres (*settings*) que modificaran el comportament del *framework*. Hi ha dos rols fonamentals per les constants. Primer, són utilitzades per anular *settings* com la mida del fitxer a carregar o definir si *Struts* estarà en '*devMode*'. En segon lloc, les constants especifiquen quin tipus de *Bean* serà triat en les implementacions.

A continuació parlarem de les constants i dels fitxers on es trobaran. Les constants poden ser declarades en varis fitxers. Per defecte, són cercades en el següent ordre, permeten als posteriors anular la troballa dels anteriors:

1. *struts-default.xml*
2. *struts-plugin.xml*
3. *struts.xml*
4. *struts.properties*
5. *web.xml*

En totes les formes de llenguatge XML la constant requereix dos atributs: nom i valor (1). En el fitxer *struts.properties*, cada entrada és considerada una constant. En el *web.xml* els paràmetres de inicialització *FilterDispatcher* són carregats com constants.

6.1.1.-Exemple d'ús

Per *struts.xml*:

```
<struts>
  <constant name="struts.devMode" value="true" />
  ...
</struts>
```

Per *struts.properties*:

```
struts.devMode = true
```

Per *web.xml*:

```
<init-param>
        <param-name>struts.devMode</param-name>
        <param-value>true</param-value>
 </init-param>
```

Totes les propietats poden ser definides en un fitxer XML (*) emprant el mode explicat abans.

El *framework Struts* utilitza una sèrie de **propietats** que es permeten modificar per tal de satisfer les necessitats del desenvolupador. Els canvis han de especificar la clau de la propietat i el seu valor en el fitxer ***struts.properties*** que sol estar col·locat sota el directori */WEB-INF/class.* El llistat complet de totes les propietats es troba en el fitxer ***struts-default.properties*** (dins *struts2.jar*).

El fitxer descriptor *web.xml* és el nucli de l'aplicació web *Java*, aleshores és lògic que sigui part del nucli de *Struts*. En *web.xml*, *Struts* defineix el filtre de servlets anomenat *FilterDispatcher* que inicialitza *Struts* i dirigeix totes les demandes. És el filtre que conté els paràmetres que entre altres efectes dictaran el comportament del *framework*.

No es recomanable ni tampoc es requereix configurar un ***taglib*** (veure l'exemple). Els *taglib* són inclosos en el fitxer *struts-core.jar* i el contenidor de l'aplicació ho descobrirà de forma automàtica. Ara bé, si per qualsevol raó necessitem configurar un *taglib* dins del *web.xml* aleshores copiem el fitxer *struts-tags.tld* al directori *WEB-INF* i a continuació ja podem afegir l'element *taglib* al *web.xml*.

Exemple de taglib:

```
<taglib>
    <taglib-uri>/s</taglib-uri>
    <taglib-location>/WEB-INF/struts-tags.tld</taglib-location>
</taglib>
```

Hem vist la configuració de constants i els fitxers on es guarden. Es poden afegir, esborrar i canviar nom i valor per tal d'aconseguir una nova funcionalitat per el *framework*, en aquest cas *Struts*.

(*) Llenguatge extensible de creació de pàgines web

6.2.-Un cas simple de configuració / modificació en Hibernate

És un cas presentat tant d'integració com de modificació mitjançant configuració en *Hibernate*. El *framework* que integrarem amb *Hibernate* serà *JUnit*, utilitzat per executar proves sobre aplicacions web. Primer crearem un fitxer anomenat ***hibernate.properties*** (*) on es guardarà la configuració del comportament de *Hibernate*. En segon lloc, per tal de poder executar les proves amb *JUnit* modificarem el fitxer senzillament esborrant el signe de comentari # en la línia:

```
#  hibernate.hbm2ddl.auto=create-drop
```

L'efecte immediat és que *Hibernate* crearà una base de dades en blanc cada cop que s'executi la prova ja que no es guardarà cap dada entre dos proves.Com la prova no s'executarà contra un servidor de BD aleshores s'ha de modificar també la connexió URL i deixar-la així:

```
hibernate.connection.url=jdbc:hsqldb:mem:mydatabase
```

Ara cada cop que JUnit executi una prova obtindrem unes dades en la BD de la nostra memòria sense necessitat d'estar connectats a un servidor BD. És molt útil per tal de conèixer l'estat de la persistència de la nostra aplicació abans de donar-la per acabada.

(*) Fitxer de configuració on es declaren les propietats que ha de tenir *Hibernate* quan s'inicialitza. Aquest fitxer s'ha de col·locar en el camí de l'aplicació per tal de ser detectat i llegit per *Hibernate* sempre que es creï un objecte *Configuration*.

----------^----------

7.-Disseny d'un *framework*

En l'apartat anterior hem vist que la millora d'un *framework* mitjançant una nova funcionalitat no és fàcil aconseguir-ho només tractant el seu fitxer de configuració i que realment apareix un nou *framework* que implementa la nova funcionalitat, en aquest cas *JUnit*.

En l'actualitat els desenvolupadors disposen d'un munt de *frameworks* que faciliten la seva programació d'aplicacions J2EE: *Struts, Spring, Hibernate, Tiles, Avalon, WebWorks, Tapestry*, o *Oracle ADF*, només per anomenar uns quants. Tanmateix molts d'aquests desenvolupadors es troben que els *frameworks* anteriors no són el remei per solucionar tots els problemes que apareixen en la programació. La raó

és que el fet de que siguin de codi obert no vol dir ni de bon tros que resulti fàcil canviar-los o millorar-los.

Per això quan un *framework* no cobreix els requisits d'una area clau, o està molt unflat o és massa car, aleshores es fa necessari construir el nostre propi *framework* per damunt de tot. Ara bé, fabricar un *framework* de l'estil de *Struts* no és gens senzill. Però si que ho és desenvolupar un de manera gradual i que faciliti millores en *Struts* o altres *frameworks*.

7.1.-Disseny sobre Struts

Si agafem com exemple *Struts,* un problema que sorgeix quan l'aplicació J2EE creix és el fort aparellament. *Action* crida un *BusinessManager*, i aquest crida un *DAO*. En una aplicació típica J2EE amb *Struts*, tenim els següents elements:

- ✓ La capa de *HttpServlet* o *Struts Action* que manega *HttpRequest* i *HttpResponse*
- ✓ La capa de lògica de negoci
- ✓ La capa d'accés a les dades
- ✓ La capa de domini que mapeja a les entitats

L'arquitectura que es mostra en la figura 8 té un problema: el fort aparellament. Però funcionarà bé si la lògica que guia a *Action* és simple. Tot es complica si hi ha la necessitat d'accessos a diversos *EJB* o a diferents serveis *Web* des de diferents fonts, etc. Quan l'aplicació Struts creix es pot arribar a un punt on el desenvolupador queda embolicat amb tantes referències entre classes de la capa Web (*Action*) i de la capa de negoci (*Manager*), observem les línies en vermell de la figura 9 per veure el cas.

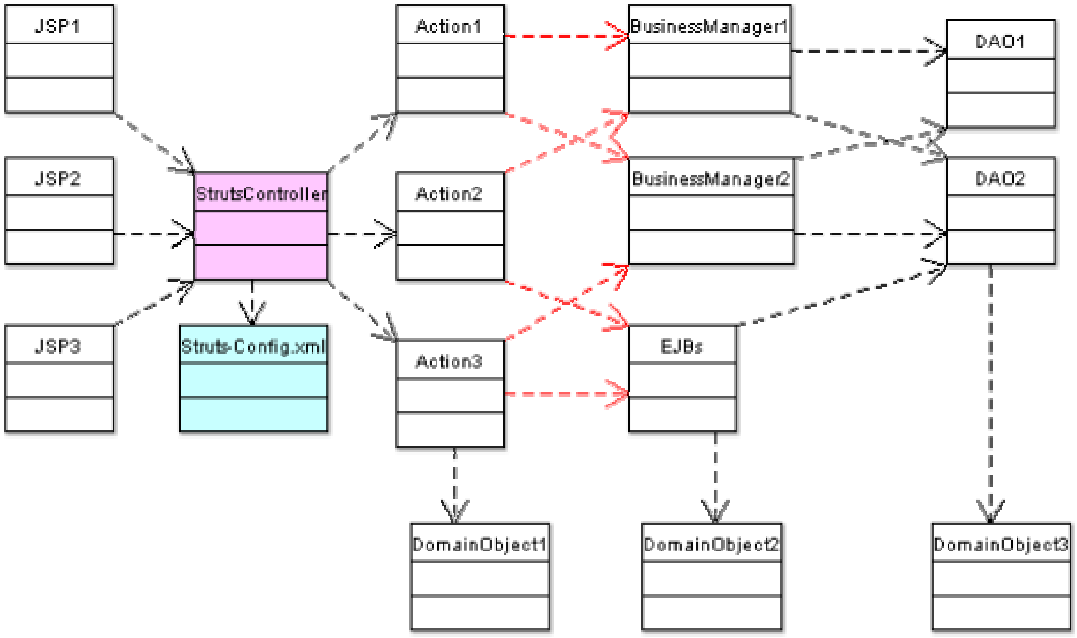


Figura 9. Arquitectura aplicació J2EE amb Struts

Nos permet *Struts* nos permet mitjançant el seu fitxer *struts-config.xml* solucionar el problema del fort aparellament? Malauradament no.

Aleshores hem de pensar en aconseguir mitjançant disseny un nou *framework* que solucioni el for aparellament mostrat en l'arquitectura *Struts*. L'estratègia per obtenir-lo hauria de seguir els següents passos:

- ✓ Comparació amb altres *frameworks* com alternatives possibles i seleccionar el que millor ens ajudi a construir el nostre propi
- ✓ Implementar el codi de forma gradual

Afegirem en el present apartat que el *framework* que apunta una possible solució al nostre cas és *Spring* i el codi que s'implementarà es basarà en el concepte de **servei** (service). Servei és un concepte present en gaire bé tots llocs però neutral, no específic ni determinant. Quelcom pot ser un servei, no només els serveis Web. Per exemple l'element *Action* pot tractar un mètode del *bean* **de sessió i sense estat** com un servei. En el disseny aleshores la forma d'emprar un servei serà genèrica.

L'objectiu és obtenir una nova estructura *Struts* com la mostrada en la figura 9.

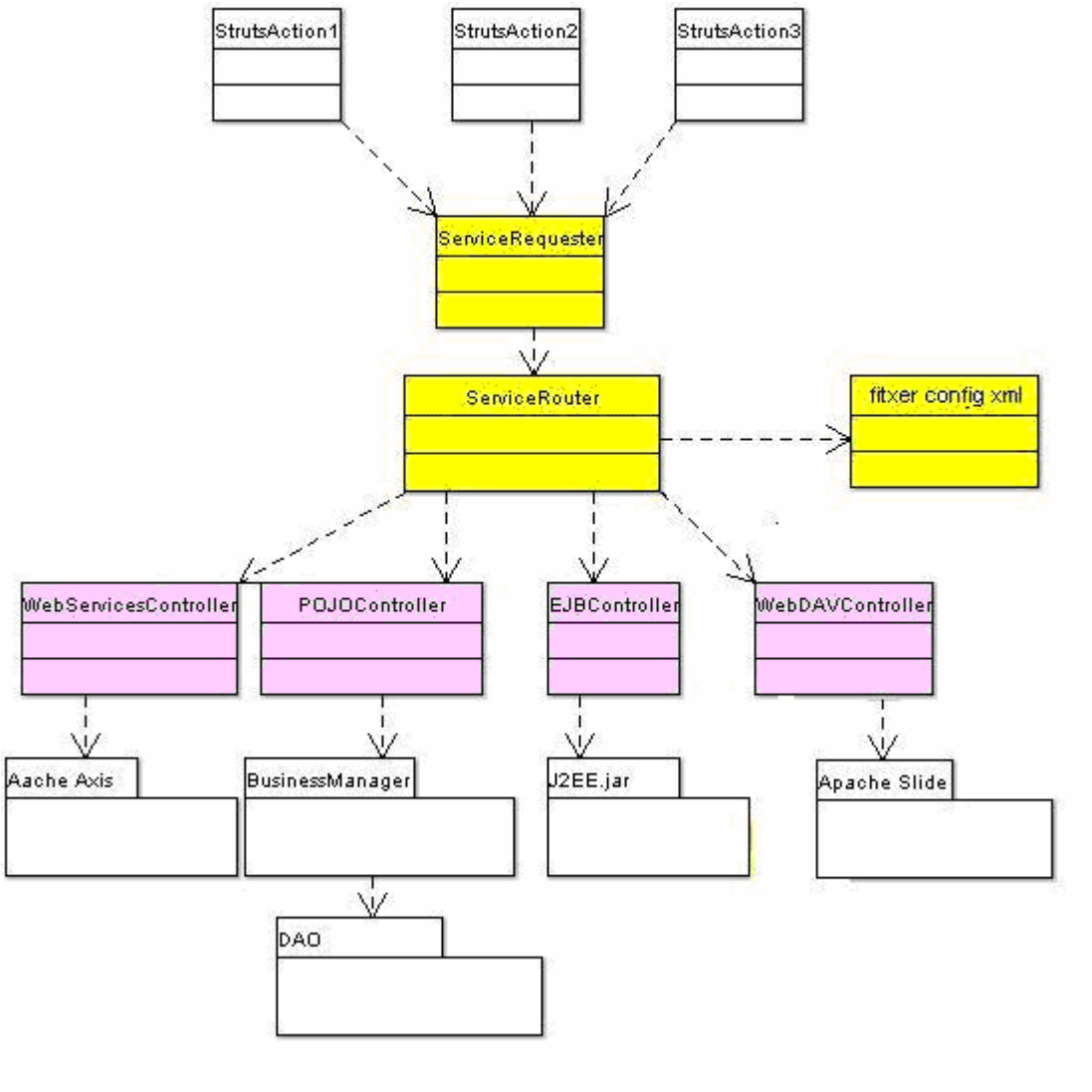


Figura 10. Arquitectura orientada a serveis per aplicació J2EE amb Struts

Per concloure aquest breu espai dedicat al disseny de *frameworks* voldríem senyalar que en la confecció de aplicacions J2EE es treballa bàsicament amb un curt ventall de *frameworks* i que per tal de millorar-los o afegir noves funcionalitats se sol fabricar un de nou i fonamentat en un ja establert que permet eliminar les mancances o limitacions dels *frameworks*-base com *Struts*, *Spring* o *Hibernate*. En l'exemple d'implementació per tal de trencar el dur aparellament s'ha utilitzat *Spring* com font i després s'ha creat codi i fitxer de configuració per un nou *framework (part en color groc de la figura 9)*.

----------^----------

Capítol 3: Integració i modificació de frameworks en una aplicació J2EE

1.- Introducció.-

En el present document no només es tractarà de les opcions que té un desenvolupador per a crear una aplicació web d'acord amb el material disposat en J2EE, EJB, POJO, contenidors, fitxers de desplegament, etc. sinó de les grans possibilitats que ofereixen els *frameworks* i que es van mostrar en el Capítol 2. Tanmateix tot el que es justificarà a nivell teòric també quedarà reflectit en un treball pràctic, l'aplicació J2EE creada.

Aleshores fabricarem una aplicació J2EE de nom Acadèmia a partir d'un projecte Ateneu. L'idea parteix de la necessitat de disposar d'una web amb contingut dinàmic i que els usuaris interactuïn i naveguin per les pàgines. Un usuari privilegiat serà el Gestor que actualitzarà els continguts.

Una associació cultural està dedicada entre d'altres tasques a oferir cursos d'anglès, gimnàstica, música, etc en un entorn que sintetitzarem com Acadèmia. Aquests cursos són a l'abast dels socis de l'entitat. Cada curs és una Secció i es dividirà en Aules amb característiques de nivell, horari o professorat. La web dissenyada reflectirà la situació de la Acadèmia. El soci navegarà per cercar informació sobre les Seccions i Aules. El Gestor ho farà per donar d'alta noves Aules o per donar de baixa a les que no tenen prous alumnes, manca de professor, etc.

L'esquema que surt del paràgraf anterior és molt semblant al de aplicacions web tipus *Categoria-Producte* que trobarem en diferents webs com la mateixa *Sun Microsystems* de Java. La mecànica d'aquest tipus d'aplicacions és clarament comercial (*e-commerce*). La manera de implementar-la és molt variada i també està a l'abast en la web citada abans. Nosaltres emprarem *frameworks* i model-2 *MVC* i ens interessarà un caire docent i no pas de negoci en el producte final.

--------^--------

2.- Projecte Ateneu.-

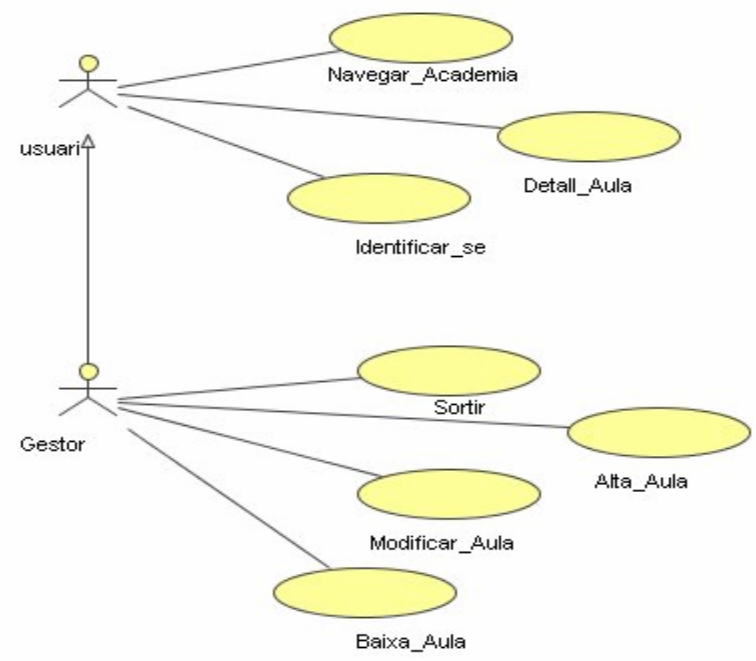
Es tracta de construir una aplicació web amb unes característiques determinades: tipus J2EE, 3 capes, model MVC i *frameworks* entre d'altres. A partir d'ara seguirem el disseny típic d'un programa i començarem per la recollida o anàlisi de requeriments més coneguda com **anàlisi funcional**.

2.1.-Anàlisi funcional.-

La primera etapa de la nostra aplicació web serà recollir els requeriments funcionals del sistema. Es vol que qualsevol usuari pugui navegar per la web per llistar totes les Aules d'una Secció, seleccionar un Aula i obtenir informació sobre la mateixa, professor, horari, matrícula, etc. En canvi només el Gestor podrà aplicar canvis en la Secció, és a dir, donar d'alta un Aula, modificar la seva informació o donar-la de baixa. Per tant bàsicament es mostrarà informació en les pàgines web i estarà sotmesa a la gestió d'un usuari especial, el Administrador o Gestor.

2.2.-Diagrama de casos d'ús.-

Un segon Anàlisi necessari i derivat de l'anterior és l'Anàlisi de casos d'ús i que ho mostrarem mitjançant un diagrama:



--- Figura 9: diagrama de casos d'ús ---

En l'esquema s'indica en primer lloc qui són els actors, l'usuari en general i el seu hereu, el Gestor. En segon lloc es mostra les funcionalitats que pot arribar a utilitzar cadascun dels dos. Quan l'usuari s'identifica com Gestor (nom i contrasenya) accedirà a funcionalitats privilegiades.

2.3.-Regles de la lògica de negoci.-

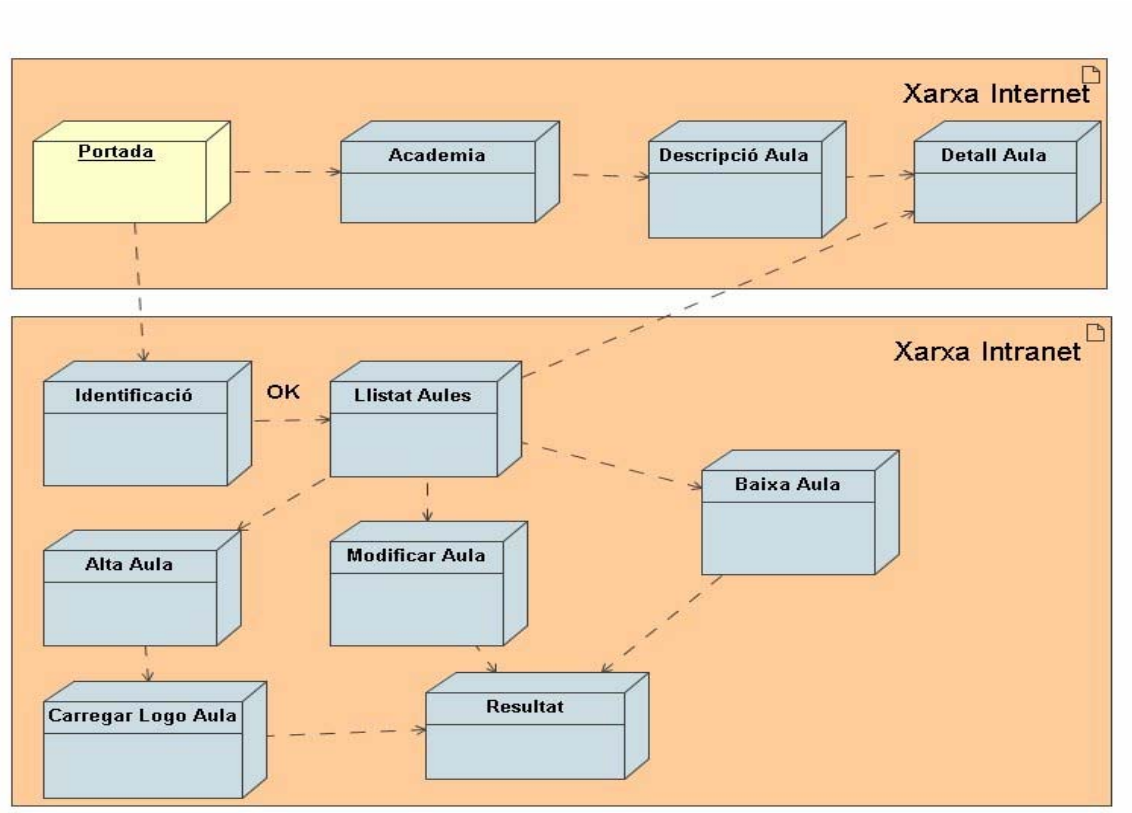
L'aplicació ha de complir les següents regles en la lògica de negoci:

- ✓ Un únic identificador per Aula
- ✓ Aquest identificador no es pot canviar
- ✓ Un Aula només pertany a una Secció
- ✓ Tota Aula pertany a una Secció

Les regles detallades abans són importants en les condicions en que es guardarà la informació en la Base de Dades.

2.4.-Navegació per la web Acadèmia.-

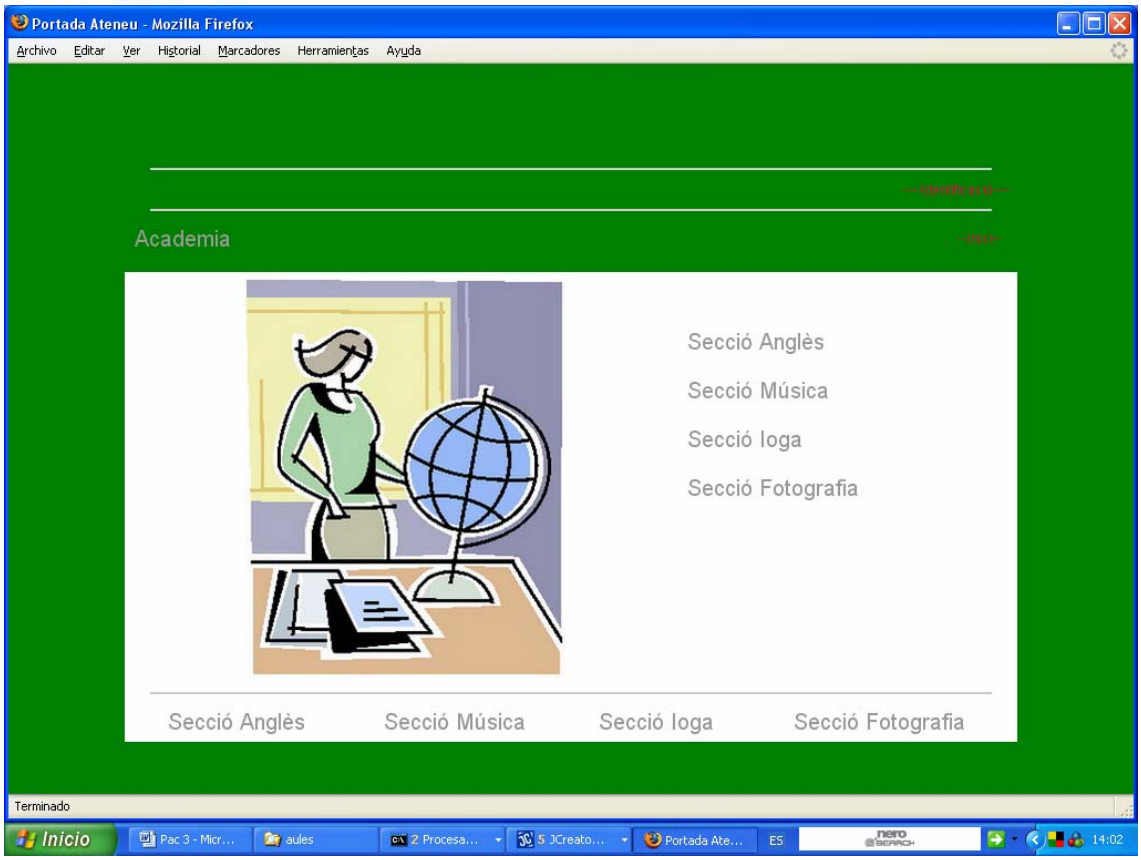
En la figura següent es mostra el flux de navegació per les pàgines de la web Acadèmia:



--- Figura 10: esquema de navegació web Ateneu---

Afegirem una breu explicació. En l'aplicació hi haurà dos conjunts de pàgines, un d'accés públic i un altre només d'accés en la xarxa privada d'intranet per l'usuari privilegiat que s'identificarà correctament. La pàgina *Descripció Aula* és realment un marc desplegat dins de la pàgina *Acadèmia*. El llistat de totes les aules (*Llistat Aules*) només serà mostrat al Gestor, és una pàgina important que conté els enllaços de les funcionalitats d'esborrar (donar de baixa) , afegir (donar de alta) i actualitzar (modificar) Aules.

A continuació mostrarem una de les pàgines de l'aplicació, Portada:



--- Figura 11: pàgina de portada Ateneu ---

2.5.-Disseny de l'arquitectura.-

El disseny de l'arquitectura al més alt nivell de la nostra aplicació serà el següent pas. Correspon a una aplicació de tres capes que contenen diferents components funcionals. Tot i que crearem una aplicació J2EE l'arquitectura en canvi és independent del tipus de tecnologia emprada.

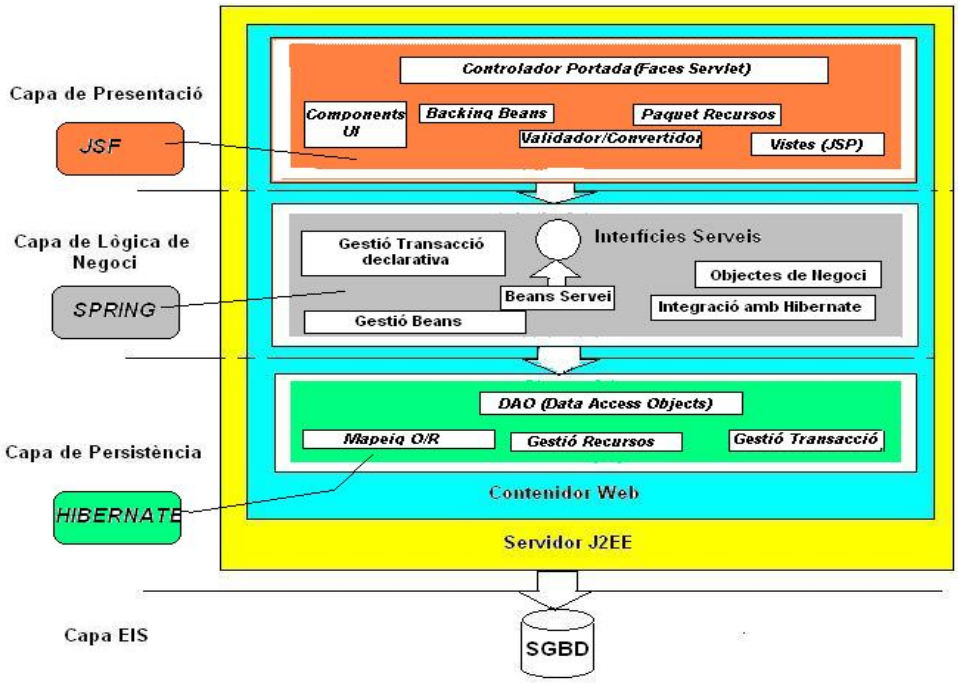
Una arquitectura multicapa divideix el sistema en unitats funcionals separades: client, presentació, lògica de negoci, integració o persistència, i el sistema de informació de l'empresa (EIS). Aquesta separació permet una divisió de responsabilitats i facilita el manteniment i l'extensió del sistema.

Si comparem aquesta arquitectura amb una de les més conegudes i utilitzades com és la client-servidor aconseguim una escalabilitat i flexibilitat que no té l'última per la manca d'una capa de lògica de negoci.

De manera molt breu explicarem les 5 capes de la divisió establerta. La primera de totes és la client, consisteix realment en el navegador i no conté la lògica de la presentació que recau en la següent capa. La segona capa mostra els serveis que s'oferiran al client, sap com processar un requeriment del client, com ha de interactuar amb la capa següent, lògica de negoci, i com seleccionar la corresponent vista a desplegar en la web.

La capa mitja (sovint es considera a la seva fusió amb la capa de integració) és molt important, conté la lògica de negoci on es guarden els objectes o entitats del negoci i els serveis oferts; rep els requeriments del client a través de la capa de presentació, els processa d'acord amb la lògica de negoci programada i facilita l'accés als recursos de la capa EIS. Els components de la capa mitja gaudeixen de serveis a nivell de sistema com són la gestió de la seguretat, dels recursos o de les transaccions.

Finalment disposem de dos capes relacionades amb l'emmagatzematge de la informació, persistència i EIS. La primera fa de pont entre la lògica de negoci i la EIS encapsulant la lògica per tal d'interactuar amb la EIS. La informació de l'aplicació es torna persistent en la EIS, entre els diferents tipus de dades que pot guardar destacarem per el nostre cas, les bases de dades relacionals. A continuació es mostra l'esquema de l'arquitectura dissenyada i explicada anteriorment:



--- Figura 12: esquema de l'arquitectura Ateneu ---

La figura mostra una aplicació d'arquitectura multicapa i no distribuïda. El diagrama serveix igualment per indicar el desplegament dins d'un únic contenidor Web. Destacarem que les interfícies definides entre capes aïllen les corresponents responsabilitats per capa.

2.6.-Tecnologies.-

És moment de discutir i analitzar les possibles tecnologies que s'empraran per implementar cada una de les tres capes, presentació, lògica de negoci i persistència.

Capa de presentació:

La experiència dels desenvolupadors experts aconsella utilitzar un *framework* ja existent per aquesta capa millor que no pas crear un de propi. Disposem d'un bon grapat per implementar la capa web, *Struts, WebWork*, i *JSF* entre d'altres. Ens hem decidit per ***JSF***.

Capa de lògica de negoci:

Si la nostra aplicació fos distribuïda els *EJB* amb les seves interfícies remotes serien millor opció que els ***POJO***. Ara bé com crearem una aplicació que no requereix accés remot aleshores ens inclinem per els *POJO*. Això si, emprarem també el *framework* ***Spring*** per ajudar als *POJO* en la implementació d'aquesta capa.

Capa de persistència:

Aquí és on es manipulen les dades persistents amb les bases de dades relacionals. Disposem de diferents aproximacions entre les quals tenim els *beans* d'entitat i el *framework* de mapeig O/R. Un *bean* d'entitat amb *CMP* és una manera costosa de resoldre l'aïllament del codi de l'accés a les dades i del control de la persistència de les dades en el mapeig O/R i a més lliga l'aplicació al contenidor EJB. Ens decidim per un eficient *framework* de mapeig O/R, centrat en els objectes, ***Hibernate***.

--------^--------

3.- Capa de presentació.-

Primer de tot recordarem que la capa de presentació recull les dades introduïdes per l'usuari, presenta la informació a l'usuari, controla la navegació per les pàgines, i envia a la capa de lògica de negoci les dades introduïdes per al seu processament. Ara bé, no hem d'oblidar dos tasques bàsiques en aquesta capa, la validació de l'usuari i el manteniment de l'estat de la sessió. En els apartats següents explicarem breument els patrons de la capa de presentació i la raó de triar JSF per implementar-la.

3.1.-MVC.-

MVC és l'acrònim per Model-Vista-Controlador que correspon al patró de disseny de l'arquitectura de la capa de presentació. Java-BluePrints recomana MVC per el disseny d'aplicacions interactives. MVC separa els punts de preocupació en el disseny per part del desenvolupador i aleshores disminueix notablement la duplicació de codi, fa que l'aplicació resulti més extensible i amb un control centralitzat (Controlador).

3.2.-JSF.-

És l'acrònim per ***JavaServer Faces***. Es defineix com un *framework* Java estàndard per la construcció de interfícies d'usuari en les aplicacions web. *JSF* (ja descrit a el capítol 2) simplifica el desenvolupament de interfícies en les següents formes:

- ✓ Proveeix d'una aproximació a un desenvolupament independent del client i centrat en el component web
- ✓ Simplifica l'accés i gestió de les dades de l'aplicació des de la interfície d'usuari
- ✓ Organitza i regula automàticament l'estat de la interfície de l'usuari en el seguit de peticions (*requests*) i diferents clients d'una manera senzilla i no obstructiva
- ✓ Proporciona facilitats i funcionalitats variades als desenvolupadors d'acord amb les habilitats dels mateixos
- ✓ I sobre tot s'aprofita de la experiència acumulada en el desenvolupament d'aplicacions web condensant-la en un API

3.2.1.-JSF i MVC.-

JSF s'adapta perfectament al patró MVC, ofereix una clara separació entre comportament (MC) i presentació (Vista). D'una manera breu destacarem els elements de JSF i com implementen aquesta separació.

Backing Beans: formen la capa de model (M), contenen les accions que són una extensió de la capa de Controlador (C) i envien les peticions a la capa de lògica de negoci (que es correspon amb model o M).

JSP: pàgines web amb etiquetes personalitzades JSF que formen la capa de presentació (V).

Faces Servlet: proporciona la funcionalitat del controlador (C).

3.2.2.-Les raons de triar JSF.-

En primer lloc introduirem **UI** com l'acrònim per *User Interface* o interfície d'usuari. Hi ha dos principals tipus de **components UI**, uns que inicialitzen una acció, com per exemple un botó, i altres que proporcionen dades com és un camp d'introducció de text. A continuació descriurem les característiques que distingeixen a JSF de la resta de *frameworks* de la capa web:

- ✓ Desenvolupament d'una aplicació web orientada a objectes a l'estil *Swing* (biblioteca gràfica per Java, conté botons, caixes de text, taules, etc)
- ✓ Control dels **backing beans** que són *JavaBeans* associats amb els components UI de la pàgina web. El control permet separar la definició dels objectes UI dels objectes que processen i guarden informació. La implementació de *JSF* emmagatzema i controla els *backing beans* en el seu apropiat context.
- ✓ Un model de components UI extensible, que vol dir que els components UI són reutilitzables i configurables, es pot crear a partir d'un component UI estàndard un altre de més complex i adaptat a les nostres necessitats.
- ✓ Un model flexible de interpretació, el intèrpret separa la vista i la funcionalitat del component UI, és una mena d'actor i aleshores podem disposar (estendre) de varis intèrprets per oferir diferents aspectes per un mateix component UI
- ✓ Un model extensible de conversió i validació de dades, a partir de convertidors i validadors estàndard es pot obtenir de nous personalitzats oferint millor protecció

Tot i semblar que JSF és prou potent i segura ens trobem que els seus components, validadors i convertidors són bàsics i per tant no del tot satisfactoris. La validació per-component no és capaç de gestionar les validacions *many-to-many* entre validadors i components. Hem d'afegir que les etiquetes personalitzades de JSF (*custom tags*) no es poden incorporar a la llibreria d'etiquetes JSP de manera simultània.

En l'apartat següent analitzarem amb detall les decisions de disseny a l'hora de implementar l'aplicació web Ateneu. Ho farem respectant en molts casos els termes en anglès per no emprar una traducció en català que no resultaria la més adient.

3.3.-Managed bean, backing bean, objecte de vista, i model d'objecte de domini.-

Un **managed bean** és essencialment un **POJO** (*plain old java object*) que guarda dades de l'aplicació i és controlat per el contenidor, està registrat en temps d'execució via fitxer XML (*faces-config.xml*) i és inicialitzat quan l'aplicació ho necessita també en temps d'execució. Un *managed bean* descriu com un *Bean* és creat i controlat.

Un **backing bean** conté una sèrie de funcionalitats per implementar seguiments d'esdeveniments o actualitzar informació per una pàgina específica:

- ✓ Configurar les propietats corresponents dels camps de text com les cadenes (*string*) **userid** o **password**
- ✓ Implementar els mètodes d'acció (*Action*) i escoltadors (*listeners*) d'esdeveniments que corresponen a components UI que inicialitzen esdeveniments d'acció com el botó 'Pàgina següent'
- ✓ Establir declaracions de instàncies de components UI que poden ser directament dirigides a un component UI de la pàgina

El *backing bean* és el intermediari entre la pàgina i la lògica de negoci. Això no significa construir un *backing bean* per cada pàgina ja que suposaria duplicar codi en una aplicació real i per tant varies pàgines es lliguen a un sol *backing bean* com es veurà en la nostra aplicació. Així com en *Struts*, *Action* i *ActionForm* es reparteixen separadament el comportament i les dades , en JSF el *backing bean* conté tant les dades com el comportament relacionat amb aquesta informació.

El *backing bean* és l'**objecte vista** (*view object*) en una aplicació basada en JSF i per tant coincideixen els conceptes per objecte vista i *backing bean*. En general un objecte vista és un objecte del model utilitzat expressament en la capa de presentació. Com vista conté la informació a mostrar i com model té la resposta a seguir, per exemple com validar, tractar esdeveniments o interactuar amb la capa de negoci. Els dos conceptes, *backing bean* i objecte vista, seran equivalents en la nostra aplicació.

Domain Object Model o model de objectes de domini es basa en el disseny orientat a objectes. Els objectes de domini són els objectes de l'aplicació que tenen incorporades les funcionalitats i característiques del sistema que es construeix, per exemple factures, usuaris, productes, etc. És necessari separar l'objecte vista del **model d'objecte de domini** quan l'aplicació és complexa com en el nostre cas. Un model d'objecte de domini, pertany a la capa de lògica de negoci, conté les dades i la lògica associada amb l'específic objecte de negoci.

En la nostra aplicació tenim com exemple de model d'objecte de domini a *AulaListBean*. Això si, cal establir un **mapeig de dades** entre els dos objectes, vista i model de domini. El fitxer jar *commons-beanutils* de API permet la implementació del mapeig de dades.

3.4.-Seguretat.-

JSF no disposa d'una característica pròpia de seguretat i volem que nostra aplicació sigui segura en l'autenticació tot i que no necessiti autorització. La diferència entre autenticació i autorització és clara, la primera exigeix identificar l'usuari per exemple per nom i contrasenya, la segona controla l'accés a certes parts del sistema o a certes funcionalitats basant-se en la identitat de l'usuari.

Tenim varies opcions per gestionar l'autenticació en JSF:

- ✓ *Backing bean*: simple però lliga el *backing bean* a una específica jerarquia hereditària
- ✓ *ViewHandler* : la lògica de seguretat està lligada a una específica tecnologia de capa web
- ✓ ***Servlet Filter****:* JSF és una aplicació web basada en Java i per tant disposa de filtres per gestionar l'autenticació i també molt important la lògica de l'autenticació es desacobla de la pròpia aplicació web

En la nostra aplicació utilitzarem un filtre que serà la classe **SecurityFilter** on en el seu codi hi haurà les localitzacions de les pàgines protegides. Una alternativa és crear un fitxer de configuració (per exemple *securityfilter-config.xml*) per guardar les normes de seguretat i els llocs de les pàgines protegides.

3.5.-Paginació.-

La paginació en una aplicació web pot ser governada en qualsevol capa. L'aplicació Acadèmia requereix paginació i serà dirigida en la capa de presentació. Com no n'hi han moltes aules la informació sobre qualsevol aula està a l'abast en una sessió d'usuari. La lògica de paginació resideix en la classe **AulaListBean**. El paràmetre relacionat amb la paginació, 'aules per pàgina' es pot configurar mitjançant una constant final en el citat *managed bean*.

3.6.-Memòria Cau.-

La tècnica de memòria cau és una de les més importants per millorar el funcionament d'una aplicació web. Consisteix en copiar dades i guardar-les quan s'accedeix a una pàgina o altra informació en la navegació web o en memòria, així en el pròxim accés a la mateixa informació (pàgina o memòria) no caldrà navegar ja que es disposa de les dades. Aquesta tècnica es pot aplicar en qualsevol capa de l'aplicació però és més recomanable quan una capa pot evitar cridar a la capa inferior. La

facilitat proporcionada per el *managed bean* de JSF permet que sigui en la capa de presentació molt més fàcil implementar la memòria cau. Si podem modificar l'espai cau aleshores el contingut en dades d'un *managed bean* pot ser tractat en diferents mides cau.

En la nostra aplicació emprem dos nivells de memòria cau. El primer dels dos resideix a la capa de lògica de negoci (directori model) en la classe **AcademiaCauServeiImpl** que manté una cau de lectura/escriptura per totes les aules i seccions de l'Acadèmia Ateneu. **Spring**, un nou framework en nostra aplicació!, governa el *managed bean* com un simple *service bean*. Per tant en l'àmbit de l'aplicació el primer nivell cau és de tipus lectura/escriptura.

Per tal de simplificar la paginació i aleshores accelerar l'aplicació, les aules també són emmagatzemades dins de la capa de presentació en l'àmbit de la sessió. Cada usuari manté el seu propi *AulaListBean* dins de la sessió. El preu a pagar per aquesta estratègia són la memòria del sistema i mantenir dades 'velles'. Al llarg d'una sessió l'usuari podrà veure dades 'velles' si l'administrador actualitza l'Acadèmia. Però tenint en compte el nombre petit de Aules i que l'Acadèmia no necessita freqüents actualitzacions no cal preocupar-se per la part negativa de l'estratègia utilitzada.

3.7.-Càrrega de fitxers d'imatge.-

La versió de *JSF* utilitzada no permet la pujada de fitxers i en canvi tenim el *framework Struts*, un dels grans per la capa de presentació, que està ben capacitat per fer-ho. Però cal la llibreria d'integració *Struts-Faces*. En la nostra aplicació Ateneu associarem una imatge per Aula. Quan l'administrador dona d'alta un Aula a continuació ha de col·locar el distintiu o imatge per l'Aula creada. Les imatges disponibles estan emmagatzemades en la memòria del servidor d'aplicacions (*Tomcat* en el nostre cas). Per tal de saber quina imatge volem pujar s'identificarà per el ID de l'Aula.

En l'aplicació Ateneu utilitzarem el servlet **FileUploadServlet** i el *API* de pujada de fitxers de *Jakarta Commons* per implementar el procés de cerca, obtenció de la imatge i inserció de la mateixa en la pàgina. Caldran dos paràmetres, directori de imatges i la pàgina resultat després de la pujada, que es poden configurar en la classe *ApplicationBean*.

3.8.-Validació.-

És una secció típica on cal prescindir (o adaptar-los) dels elements estàndard que facilita el *framework*, en aquest cas JSF. JSF porta incorporats uns validadors molt bàsics i que en la majoria dels casos no corresponen als requeriments del mon que reflecteix l'aplicació web. El

validador **SeccionsSeleccionades** ens serveix aquí per fitar el nombre mínim i màxim de seccions que s'han de seleccionar per cada aula en el procés de donar d'alta. Concretament volem que tota aula estigui associada a una sola Secció.

Cal personalitzar l'etiqueta corresponent 'academia' (*custom tag*) en el desenvolupament del validador que controla el nombre d'aules seleccionades per el component UI (*UISelectMany*):

```
<h:selectManyListbox value="#{aulaBean.idSeccioSeleccionada}"
id="idSeccioSeleccionada">
    <academia:validarNumeroSeccions minNum="1"/>
    <academia:validarNumeroSeccions maxNum="1"/>
<f:selectItems value="#{applicationBean.seccioSelectItems}"
id="seccions"/>
</h:selectManyListbox>
```

L'etiqueta personalitzada *academia* constitueix el fitxer *academia.tld* en el directori WEB-INF.

3.9.-Personalitzar el missatge d'error.-

Resulta convenient i útil per una bona interacció la estratègia d'emprar el llenguatge habitual per informar a l'usuari de quin error s'ha comés. En JSF és possible personalitzar els missatges d'error per als convertidors de dades i validadors. Cal configurar (situar) el paquet de recursos corresponent en el fitxer *faces-config.xml*:

```
<message-bundle>academia.view.bundle.Messages</message-bundle>
```

És en el fitxer Messages.properties situat a academia/vista/bundle on completem la personalització:

```
#conversion error messages
javax.faces.component.UIInput.CONVERSION=Input data is not in the correct
type.
#validation error messages
javax.faces.component.UIInput.REQUIRED=Required value is missing.
```

----------^----------

4.-Capa de lògica de negoci

Els objectes de negoci o entitats i els serveis de negoci s'ubiquen a la capa de lògica de negoci. Una entitat no només guarda dades sinó tota la lògica associada amb aquesta entitat. Tenim tres entitats o objectes de negoci en l'aplicació Ateneu: **Usuari**, **Aula** i **Seccio**.

Els serveis interactuen amb les entitats produint una lògica de negoci de més nivell. El client es trobarà amb unes interfícies de servei i per tant s'haurà de definir una capa que faci d'interfície de negoci que mostri i implementi els serveis. Nosaltres hem optat per *POJO* i el *framework Spring* per la implementació de la capa de lògica de negoci. Hi haurà dos serveis per el client, un que correspon a la mecànica de la Acadèmia (*AcademiaServei*) i un segon relacionat amb la gestió del propi usuari (*UsuariServei*).

4.1.- Spring.-

És el moment d'introduir el nou *framework* emprat en l'aplicació Ateneu: *Spring*. De fet ja hem parlat de les seves característiques més importants al Capítol 2. Recordarem que *Spring* està fonamentat en el concepte d'inversió de control (*IoC* o *Hollywood Principle*) on el codi del *framework* invoca (crida) el codi de l'aplicació i no pas a l'inrevés. Les característiques que ens interessen de *Spring* aquí són:

- ✓ ***Gestió dels beans que tenen context a l'aplicació***. *Spring* és capaç d'organitzar de manera efectiva els objectes de la capa mitja i establir les connexions entre els mateixos sense tenir que fer-ho nosaltres. Així *Spring* elimina els 'desconnectats' i manté la norma de programació orientada a objectes referida a l'ús d'interfícies
- ✓ ***Gestió de les transaccions declaratives***. És el servei llest per utilitzar-ne més important que ofereix Spring; és semblant a la proposició de valor del CMT dels EJB però amb les següents avantatges:1) es pot aplicar a qualsevol POJO; 2) no està lligada a JTA (Java Transaction API); 3) suporta semàntica addicional per tal de minimitzar la necessitat de dependència amb un API propietari de transaccions que forci una reducció
- ✓ ***Jerarquia d'excepcions en l'accés a les dades***. Spring facilita una entenedora jerarquia d'excepcions en comptes de la proporcionada per BD, *SQLException*. Cal definir **el intèrpret d'excepcions** d'accés de dades de Spring en el seu fitxer de configuració, en el nostre cas *applicationContext.xml*:

```
<!-- Spring Data Access Exception Translator Definition -->
<bean id="jdbcExceptionTranslator"
class="org.springframework.jdbc.support.SQLErrorCodeSQLExceptionTranslator">
<property name="dataSource"><ref bean="dataSource"/></property>
```

```
</bean>
```

En l'aplicació Ateneu si l'administrador dona d'alta un Aula amb un identificador ja propietat d'un altre Aula aleshores una excepció és llançada, *DataIntegrityViolationException.* Aquí intervé el 'intèrpret', es captura l'excepció i se rellança com ***DuplicatIdAulaExcepcio*** que pot ser controlada de manera diferent des de altres excepcions d'accés de dades.

✓ ***Integració amb Hibernate***. *Spring* no ens obliga a emprar la seva forta característica d'abstracció *JDBC* i tanmateix s'integra molt bé amb els *frameworks* de la capa de persistència tipus mapeig O/R com **Hibernate**. *Spring* ofereix un eficaç i segur control de les sessions en *Hibernate*, gestiona la configuració dels *SessionFactory* i les fonts de dades *JBDC* en els àmbits de l'aplicació, i sobre tot fa que l'aplicació resulti molt més fàcil de provar.

----------^----------

5.-Capa de persistència.-

Per implementar aquesta capa utilitzarem el *framework* ***Hibernate*** (ja descrit a la Part 1) que correspon a un tipus de mapeig O/R i a més suporta la majoria de sistemes de *BD SQL*. El *HQL* (*Hibernate Query Language*) és un dialecte orientat a objectes i dissenyat com una extensió de *SQL* que guarda semblances amb *EJB-QL* però més potent i elegant. *HQL* resulta molt fàcil d'aprendre a partir de conceptes bàsics de *SQL*.

Hibernate ofereix funcionalitats per recuperació i actualització de dades, control de transaccions, bancs de connexions a BD (*connection pool*), consultes tant programàtiques com declaratives, i el control del mapeig de les associacions entre entitats.

5.1.-Perquè Hibernate?

Les raons de triar *Hibernate* s'han de sumar a les mostrades a l'inici del capítol 6. *Hibernate* és menys invasor que altres *frameworks* tipus mapeig O/R. Utilitza la generació de *bytecodes* en temps d'execució i també *Reflection*. Java proveeix *java.lang.reflect.Proxy* per *JDK 1.3* i superiors. Una instància del Proxy pot fer-se en temps d'execució, implementant una donada llista d'interfícies. Aquesta solució per *Reflection* és bona quan els objectes persistents són accedits via interfície.

Reflection i generació de *bytecodes* en temps d'execució així com la generació SQL ens permeten desenvolupar objectes de la capa de persistència amb el llenguatge de Java i amb la inclusió d'associació, composició, herència, polimorfisme i el *framework* Java Collections. En el nostre cas els objectes de negoci són POJO i no necessiten implementar cap interfície *Hibernate*.

5.2.-DAO.-

En l'aplicació Acadèmia utilitzarem el patró **DAO**. Ens permet canviar els mecanismes d'accés a les dades independentment del codi utilitzat per les dades. En concret:

- ✓ Separació de la interfície del client base de dades dels seus mecanismes d'accés a les dades
- ✓ Adaptació d'un específic API d'accés a dades a una interfície d'un client genèric

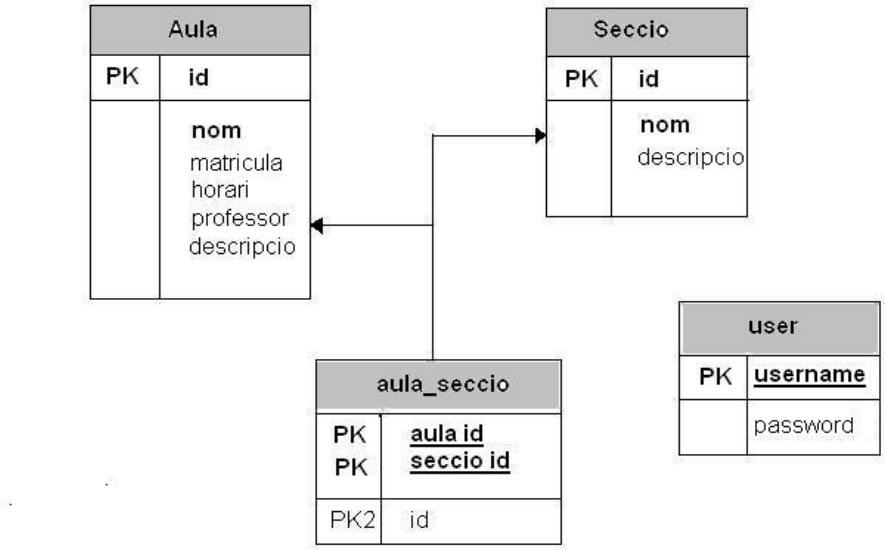
Una classe *DAO* pot proveir accés a una particular base de dades sense acoblar el *AP*I a la lògica de negoci. Per exemple, les classes d'Acadèmia accedeixen a les seccions i aules utilitzant l'interfície *AcademiaDAO*. La segona interfície *DAO* d'Acadèmia és *UsuariDAO*. Ambdues classes són implementades respectivament per *HibernateAcademiaDaoImpl* i *HibernateUsuariDaoImpl* que guarden la lògica de *Hibernate* per gestionar la persistència de les dades.

6.-Disseny de la Implementació.-

És el moment d'implementar tot el que s'ha detallat a un nivell teòric sobre les capes de l'aplicació Acadèmia i sempre d'acord amb els requeriments del client. Començarem per el disseny de la BD.

6.1.-Disseny de la Base de Dades.-

Crearem un esquema anomenat Academia (serà també el nom de la BD):



--- Figura 13:esquema de BD Academia ---

En total consta de quatre taules, Aula, Seccio, aula_seccio i user (respectem en aquesta última taula la terminologia anglesa per raons d'universalitat).

6.2.-Disseny de Classes.-

Com s'ha explicat en els anteriors apartats l'aplicació Ateneu està formada per tres capes, cadascuna de les quals queda implementada per una sèrie de classes que detallarem capa per capa.

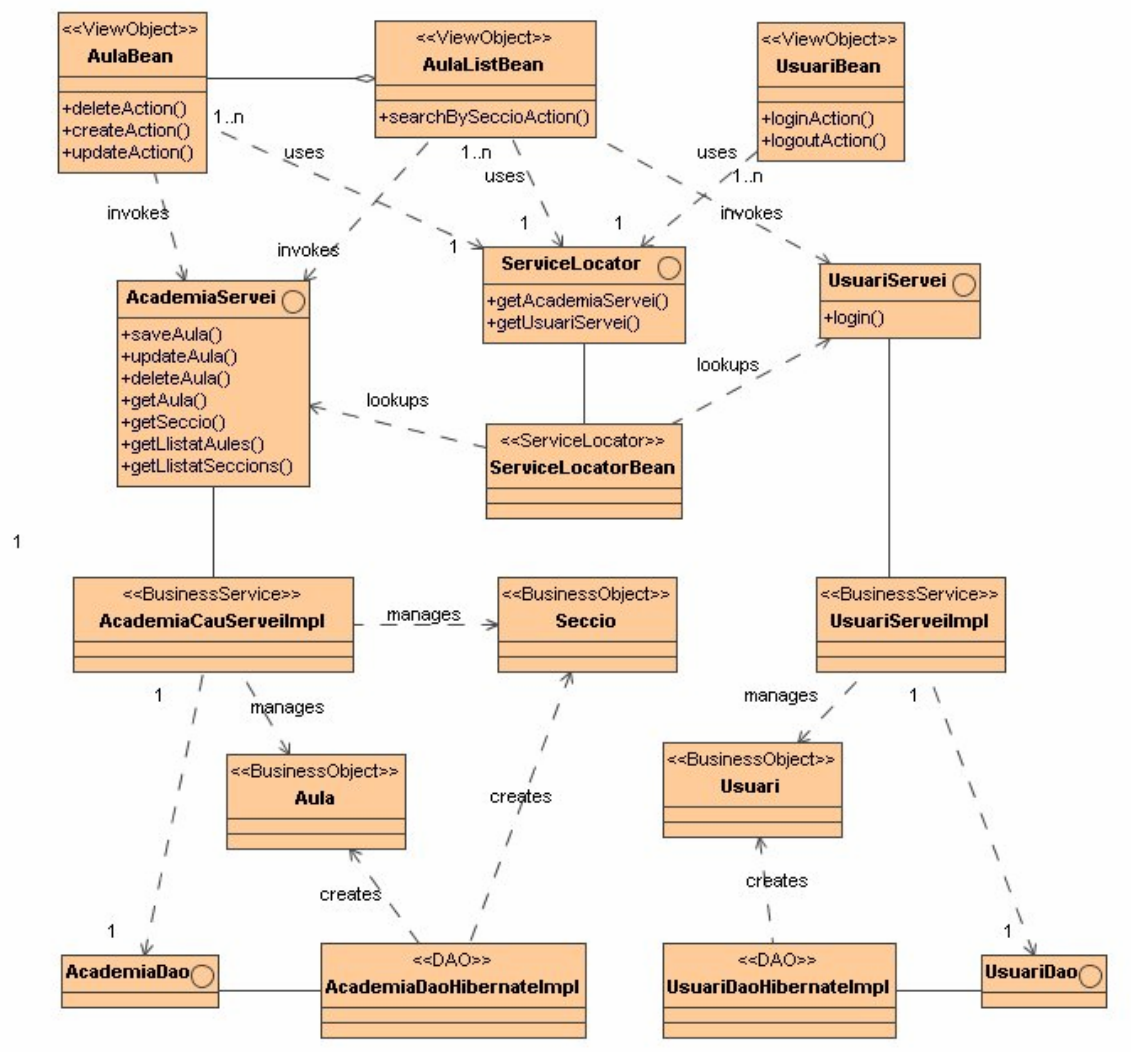
Capa de presentació: tenim tres *backing beans*, ***AulaBean***, ***AulaListBean*** i ***UsuariBean***.

Capa de lògica de negoci: hi ha dos serveis de negoci, ***AcademiaServei*** i ***UsuariServei*** i tres entitats de domini, ***Aula***, ***Seccio*** i ***Usuari***.

Capa de persistència: tenim les dos interfícies *DAO*, ***AcademiaDAO*** i ***UsuariDAO***, i les seves implementacions, ***AcademiaDaoHibernateImpl*** i ***UsuariDaoHibernateImpl***.

Spring gestiona les associacions entre els beans dins de les capes de lògica de negoci i persistència. El ***ServiceLocator*** integra *JSF* (capa de presentació) amb la capa de lògica de negoci.

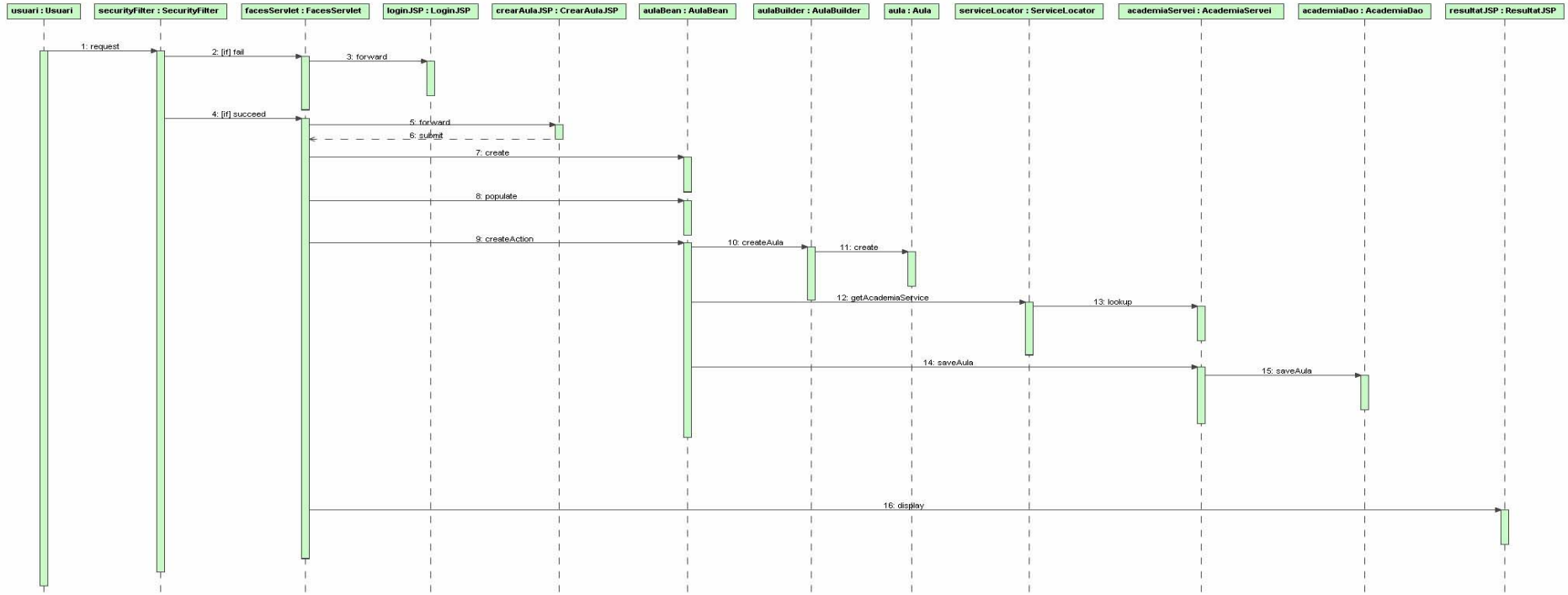
A continuació es mostra el diagrama de classes de l'aplicació Ateneu:



--- Figura 14: diagrama de classes ---

6.3.-El cas d'ús Alta_Aula (createAula).-

Mitjançant el cas d'ús mostrarem les connexions entre classes i com es construirà l'aplicació. Començarem per el diagrama de seqüència:



--- Figura 15: diagrama de seqüència cas d'ús Alta_Aula ---

6.3.1.-Capa de presentació Ateneu.-

La implementació d'aquesta capa de l'aplicació Ateneu suposa

- ✓ Crear les pàgines JSP
- ✓ Definir la navegació per les pàgines JSP
- ✓ Crear i configurar els *backing beans*
- ✓ Integrar JSF amb la capa de lògica de negoci

1) En el cas d'ús *Alta_Aula* la pàgina és *crearAula.jsp* i correspon a donar d'alta una nova aula a la Acadèmia. Ha de contenir els components UI i connectar-los a la classe *AulaBean*. L'etiqueta personalitzada ***validarNumeroSeccions*** (*academia.tld)* confirma el nombre obligat de seccions que l'usuari privilegiat (administrador) ha de seleccionar. Per cada aula és necessari seleccionar un mínim de seccions i no passar d'un màxim:

```
<tag>
    <name>validarNumeroSeccions</name>
   <tag-class>academia.vista.validator.SelectedItemsRangeValidatorTag</tag-class>
        <attribute>
              <name>minNum</name>
        </attribute>
        <attribute>
              <name>maxNum</name>
        </attribute>
  </tag>
```

2) La definició i configuració de la navegació per les pàgines JSP s'estableix en el fitxer de configuració de JSF, *faces-config.xml*. A continuació es mostren les normes de navegació per *Alta_Aula*:

```
<navigation-rule>
<navigation-case>
      <description>
      </description>
     <from-outcome>crearAula</from-outcome>
     <to-view-id>/crearAula.jsp</to-view-id>
</navigation-case>
 </navigation-rule>
-------
<navigation-rule>
   <from-view-id>/crearAula.jsp</from-view-id>
   <navigation-case>
      <description>
      </description>
     <from-outcome>success</from-outcome>
```

```
<to-view-id>/carregarImatge.jsp</to-view-id>
 </navigation-case>
 <navigation-case>
    <description>
    </description>
  <from-outcome>retry</from-outcome>
  <to-view-id>/crearAula.jsp</to-view-id>
 </navigation-case>
 <navigation-case>
    <description>
    </description>
  <from-outcome>cancel</from-outcome>
  <to-view-id>/llistatAules.jsp</to-view-id>
 </navigation-case>
</navigation-rule>
```

3) *AulaBean* conté els mapeigs de propietats a les dades que corresponen a tots els components UI de la pàgina JSP i a més conté els mètodes de les accions *createAction*, *editAction* i *deleteAction*. Mostrarem el codi del mètode lligat al cas d'ús Alta_Aula (*createAula)*.

```
/**
 * Accio del Backing bean per crear una nova aula.
 * @return resultat
 */
public String createAction() {
        this.logger.debug("createAction cridada");

        try {
                Aula aula = AulaBuilder.createAula(this);
                        this.serviceLocator.getAcademiaServei().saveAula(aula);

                //guarda l'actual id d'aula id en el bean de sessio.
                //per l'us del carregador d'imatge.
                FacesUtils.getSessionBean().setCurrentAulaId(this.id);

                //elimina la llista d'aules del cache
                this.logger.debug("borra AulaListBean del cache");
                FacesUtils.resetManagedBean(BeanNames.AULA_LIST_BEAN);
        } catch (DuplicatIdAulaExcepcio de) {
                String msg = "El identificador ja existeix";
                this.logger.info(msg);
                FacesUtils.addErrorMessage(msg);

                return NavigationResults.RETRY;
        } catch (Exception e) {
                String msg = "Impossible guardar aquesta aula";
                this.logger.error(msg, e);
```

```
            FacesUtils.addErrorMessage(msg + ": Internal Error");

                    return NavigationResults.FAILURE;
                }
                String msg = "Aula amb id " + this.id + " s'ha afegit correctament.";

                this.logger.debug(msg);
                FacesUtils.addInfoMessage(msg);

                return NavigationResults.SUCCESS;
        }
```

Explicarem que passa fonamentalment dins d'aquest mètode. Primer de tot es crea una entitat *Aula* basada en les propietats de *AulaBean*. Després *ServiceLocator* cerca *AcademiaServei*. Finalment la petició *createAula* és delegada a *AcademiaServei* de la capa de lògica de negoci.

La configuració de *AulaBean* és al fitxer de JSF *faces-config.xml*:

```
<managed-bean>
      <description>
       Backing bean amb informació de la Aula.
      </description>
      <managed-bean-name>aulaBean</managed-bean-name>
     <managed-bean-class>academia.vista.bean.AulaBean</managed-bean-
   class>
      <managed-bean-scope>request</managed-bean-scope>
      <managed-property>
       <property-name>id</property-name>
       <value>#{param.idAula}</value>
      </managed-property>
      <managed-property>
       <property-name>serviceLocator</property-name>
       <value>#{serviceLocatorBean}</value>
      </managed-property>
     </managed-bean>
```

AulaBean se configura per tenir un àmbit de petició, que vol dir que la implementació de JSF crearà una nova instancia de *AulaBean* per cada petició si *AulaBean* té una referència en la pàgina JSP (àmbit). La propietat de nom *id* s'omple amb el paràmetre de la petició *idAula*. La implementació de JSF aconsegueix el paràmetre de la petició i configura la propietat del *managed-bean*.

4) *ServiceLocator* fa una abstracció de la lògica per tal de localitzar els serveis. En la nostra aplicació està definida com una interfície implementada com *ServiceLocatorBean* que cerca els serveis dins del context *Spring*:

```
/**
* Constructor.
 */
public ServiceLocatorBean() {
		ServletContext context = FacesUtils.getServletContext();
		this.appContext =
WebApplicationContextUtils.getRequiredWebApplicationContext(context);
		this.academiaService =
(AcademiaService)this.lookupService(ACADEMIA_SERVEI_BEAN_NAME);
		this.usuariService =
(UsuariService)this.lookupService(USUARI_SERVEI_BEAN_NAME);

		this.logger.info("El bean localitzador del Servei inicialitzat");
	}
```

En *BaseBean* es defineix *ServiceLocator* com una propietat

```
protected ServiceLocator serviceLocator;
```

Aquí torna a aparèixer la inversió de control, característica bàsica de *Spring*, la facilitat del *managed bean* connecta la implementació de *ServiceLocator* amb els *managed beans* que han de connectar-se amb ell.

6.3.2.-Capa de lògica de negoci Ateneu.-

La implementació en aquesta capa suposa:

- ✓ Definir els objectes de negoci o entitats de domini
- ✓ Crear les interfícies de servei amb les seves implementacions
- ✓ Enllaçar els objectes amb *Spring*

1) El fet que *Hibernate* proveeix la persistència, els objectes Aula i Seccio necessiten definir el mètodes *set* i *get* per a tots els camps que contenen. Per exemple:

```
/**
 * Constructor per defecte.
 */
    public Aula() {
		this.seccioId = new HashSet();
	}
    public String getId() {
		return this.id;
	}
    public void setId(String nouId) {
		this.id = nouId;
	}
```

```
public String getNom() {
        return this.nom;
    }
    public void setNom(String nouNom) {
        this.nom = nouNom;
    }
    public String getDescripcio() {
        return this.descripcio;
    }
    public void setDescripcio(String novaDescripcio) {
        this.descripcio = novaDescripcio;
    }
    public String getHorari() {
        return this.horari;
    }
    public void setHorari(String nouHorari) {
        this.horari = nouHorari;
    }
    public String getProfessor() {
        return this.professor;
    }
    public void setProfessor(String nouProfessor) {
        this.professor = nouProfessor;
    }
    public double getMatricula() {
        return this.matricula;
    }
    public void setMatricula(double novaMatricula) {
        this.matricula = novaMatricula;
    }
```

2) La interfície *AcademiaServei* defineix tots els serveis (funcionalitats) relacionats amb la gestió de l'acadèmia:

```
public interface AcademiaServei {
        public Aula saveAula(Aula aula) throws AcademiaExcepcio;
public void updateAula(Aula aula) throws AcademiaExcepcio;
            public void deleteAula(Aula aula) throws AcademiaExcepcio;
           public Aula getAula(String aulaId) throws AcademiaExcepcio;
public Seccio getSeccio(String seccioId) throws AcademiaExcepcio;
         public List getLlistatAules() throws AcademiaExcepcio;
          public List getLlistatSeccions() throws AcademiaExcepcio;
 }
```

La implementació (d'acord amb la memòria cau) de la interfície *AcademiaServei* correspon a *AcademiaCauServeiImpl* i conté un mètode set per un objecte *AcademiaDao*.

```
/**
 * Entrar el codi de AcademiaDao.
 * pot ser emprat per el contenidor Spring IoC.
 * param nouAcademiaDao el AcademiaDao a entrar
 */
        public void setAcademiaDao(AcademiaDao nouAcademiaDao) {
                this.academiaDao = nouAcademiaDao;
        }
```

D'aquesta forma *Spring* enllaça la implementació amb l'objecte *Dao*.

3) La configuració *Spring* per la interfície *AcademiaServei* a *applicationContext.xml*:

```
<!-- Hibernate Transaction Manager Definition -->
        <bean id="transactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
                <property name="sessionFactory"><ref
local="sessionFactory"/></property>
        </bean>

<!-- Cached Academia Servei Definition -->
        <bean id="academiaServiceTarget" class="academia.model.service.impl.
AcademiaCauServeiImpl" init-method="init">
        <property name="academiaDao"><ref  local="academiaDao"/></property>
        </bean>

<!-- Transactional proxy for the Academia Servei -->
<bean id="academiaServei"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager"><ref
local="transactionManager"/></property>
    <property name="target"><ref local="academiaServeiTarget"/></property>
      <property name="transactionAttributes">
      <props>
        <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
        <prop key="save*">PROPAGATION_REQUIRED</prop>
        <prop key="update*">PROPAGATION_REQUIRED</prop>
```

```
    <prop key="delete*">PROPAGATION_REQUIRED</prop>
        </props>
      </property>
  </bean>
```

El control de la transacció declarativa queda configurat per a la interfície *AcademiaServei,* que es pot connectar a una implementació diferent que no sigui *AcademiaDao*. Acabem de veure com *Spring* crea i gestiona un solitari objecte *AcademiaServei* sense necessitat d'una Factoria (*bean factory*).

Finalment resta connectar la capa de lògica de negoci amb la de persistència. És a dir, integrar els *frameworks* ***Spring*** i ***Hibernate***.

Tornem al fitxer de configuració de *Spring applicationContext.xml*. Primer s'ha de configurar HibernateSessionFactory:

```
<!-- Hibernate SessionFactory Definition -->
        <bean id="sessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingResources">
        <list>
            <value>academia/model/entitats/Aula.hbm.xml</value>
            <value>academia/model/entitatst/Seccio.hbm.xml</value>
            <value>academia/model/entitats/Usuari.hbm.xml</value>
        </list>
    </property>

   <property name="hibernateProperties">
        <props>
            <prop
key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.cglib.use_reflection_optimizer">true</prop>
                <prop
key="hibernate.cache.provider_class">net.sf.hibernate.cache.HashtableCacheProvider
</prop>
        </props>
  </property>

  <property name="dataSource">
        <ref bean="dataSource"/>

</property>
        </bean>
```

En segon lloc configurarem HibernateTemplate utilitzat per l'objecte AcademiaDao en la integració Spring-Hibernate:

```
<!-- Hibernate Template Definition -->
        <bean id="hibernateTemplate"
class="org.springframework.orm.hibernate.HibernateTemplate">
                <property name="sessionFactory"><ref
bean="sessionFactory"/></property>
                <property name="jdbcExceptionTranslator"><ref
bean="jdbcExceptionTranslator"/></property>
        </bean>
```

6.3.3.-Capa de persistència Ateneu.-

És la capa corresponent al *framework Hibernate* en l'aplicació Ateneu i aleshores mostrarem com *Hibernate* la implementa. En la BD relacional que utilitzarem, *MySQL*, tenim dos taules on emmagatzemarem les dades de les aules i les seccions. *Hibernate* mapeija els objectes a la BD relacional mitjançant fitxers de configuració xml. El mapeig Objecte Aula-Taula Aula es farà utilitzant el fitxer *Aula.hbm.xml* i Objecte Seccio- Taula Seccio amb el fitxer *Seccio.hbm.xml*, d'aquest últim mostrem el contingut:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping package="academia.model.entitats">
        <class name="Seccio"
                table="SECCIO">

                <id name="id" column="ID"
                        unsaved-value="null">
                        <generator class="increment"/>
                </id>

                <property name="nom"
                                column="NOM"
                                not-null="true"/>

                <property name="descripcio"
                                column="DESCRIPCIO"/>
        </class>
</hibernate-mapping>
```

En el fitxer de configuració de *Spring applicationContext.xml* tenim l'enllaç entre *HibernateTemplate* i *Spring*:

```
<!—Academia DAO Definition: Hibernate implementation -->
<bean id="academiaDao"
class="academia.model.dao.hibernate.AcademiaDaoHibernateImpl">
   <property name="hibernateTemplate"><ref
bean="hibernateTemplate"/></property>
</bean>
```

----------^----------

Capítol 4: Implementació d'una funcionalitat en un *framework*

1.- Introducció.-

És el punt de destinació del nostre PFC. Ha calgut un llarg aprenentatge sobre els frameworks i en concret sobre els fitxers bàsics que proporcionen. En el present capítol només parlarem del *framework JavaServer Faces* dedicat a implementar la capa de presentació.

Crearem unes pàgines *JSP* mitjançant la tecnologia *JSF*. Com ja es va explicar al llarg del capítol 4 l'aplicació *J2EE* creada tracta d'una web on l'usuari Gestor pot crear i actualitzar aules.

JSF disposa d'una llibreria estàndard d'etiquetes per les pàgines *JSP* però no posa cap inconvenient en que nosaltres creem ***etiquetes personalitzades*** i les incorporem a la pàgina *JSP* corresponent.

De la mateixa manera *JSF* ofereix ***classes*** que nosaltres podem implementar i així aconseguir funcionalitats per les aplicacions. Justament en l'aplicació Acadèmia del capítol 4 volíem controlar que l'associació d'un aula fos amb una sola secció. La nova funcionalitat requerida consistirà en un ***validador***.

Per tal de que JSF incorpori el validador en el seu comportament haurem de tenir-ho com element en el ***fitxer de configuració***.

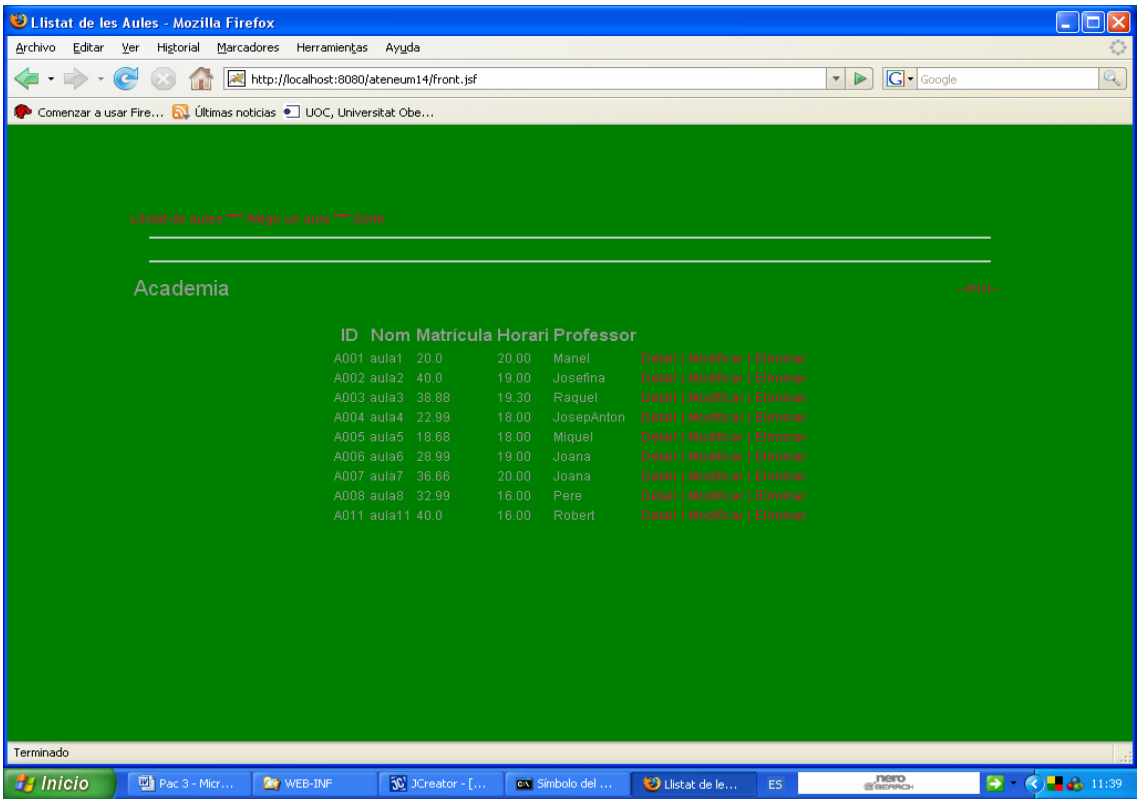
----------^----------

2.- Implementació d'una funcionalitat en JSF.-

Tot i haver un grapat de modificacions (mitjançant canvis en els fitxers estàndards de configuració) en frameworks en el present projecte i que s'han mostrat al llarg del Capítol 4 ara volem tractar un en concret i de forma exhaustiva. De fet hem triat el cas de l'etiqueta personalitzada com el millor exemple a nivell pedagògic per mostrar un canvi en un *framework* per tal d'afegir-li una funcionalitat o adaptar una de establerta. També hem seleccionat el *framework JSF* per el cas ja que està estretament relacionat amb l'entorn de comunicació de l'usuari, interacció i navegació web.

2.1.- Funcionalitat requerida.-

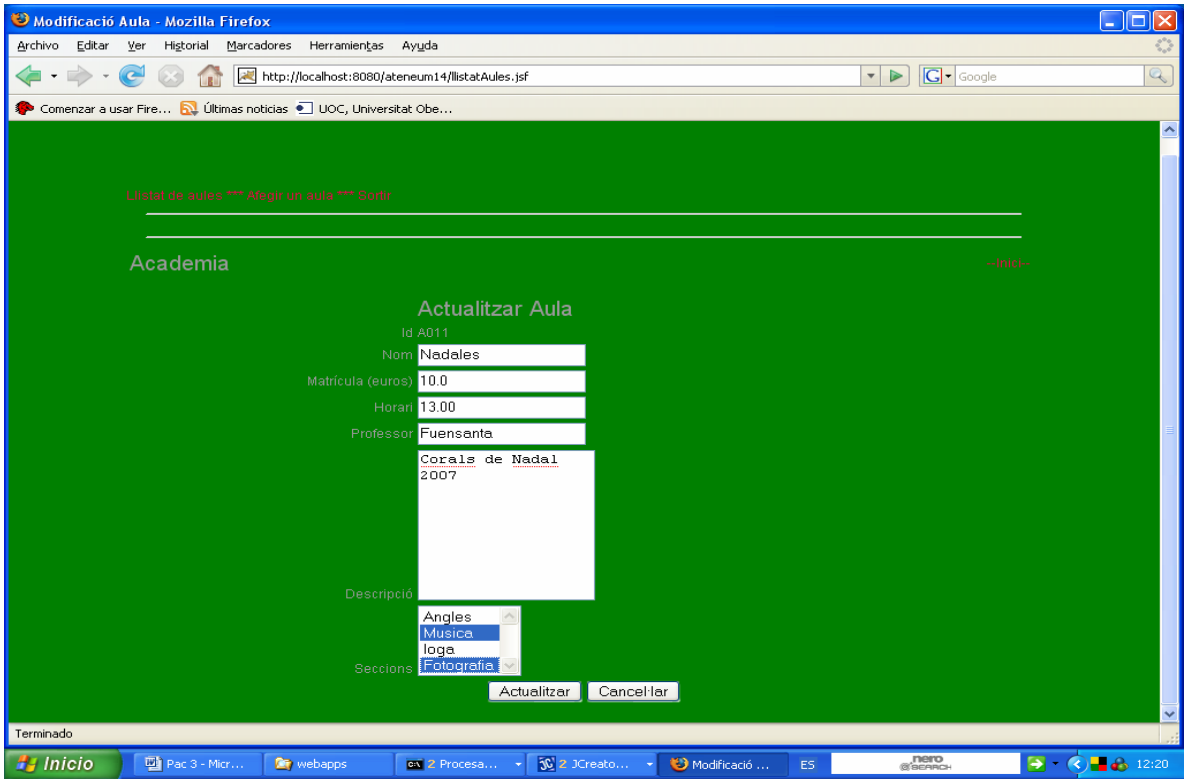
En l'aplicació J2EE que es vol implementar tenim l'usuari Gestor o Administrador amb una sèrie de funcionalitats restringides al seu rol: llistar totes les Aules, donar d'alta o crear una nova Aula, actualitzar un Aula o modificar alguna de les seves característiques i poder eliminar una del Llistat d'Aules. En la figura 8 es veu la pàgina d'accés a les funcionalitats de Gestor.



--- Figura 16: pàgina de llistat de les aules ---

Justament en 'Afegir un Aula' i en 'Modificar' hem dissenyat la relació d'un Aula amb una i només una Secció. Aleshores necessitem que en el moment de 'Afegir un Aula' i arribat a l'instant de seleccionar la Secció a la que ha de pertànyer, l'aplicació controlarà el número de seccions seleccionades de tal manera que no acceptarà que estigui per sota de 1 o per damunt de 1, és a dir Num_seccions_selec ha de ser exactament 1. És obvi que un aula de curs de ioga no pot estar també a fotografia i també és raonable que tota aula pertanyi a una secció.

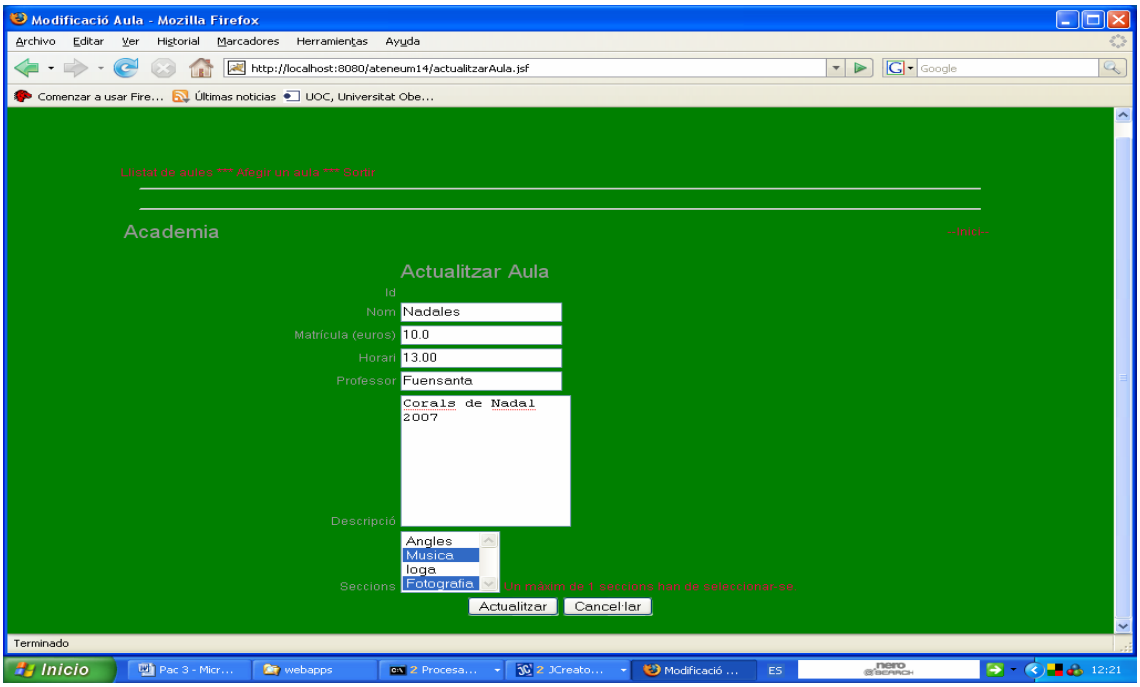
Igualment en el procediment de 'Modificar' quan el Gestor ha de seleccionar la secció de la nova configuració de l'Aula l'aplicació no permeti que el número de seccions seleccionades sigui diferent de 1. En la figura 9 es veu la pàgina per actualitzar l'aula on destaquen dos seccions en blau:



--- Figura 17: pàgina per actualitzar un aula ---

Ara disposem de les dues opcions: Actualitzar l'aula o Cancel·lar l'operació. Si triem Cancel·lar aleshores tornarem a la pàgina de Llistat d'aules (veure *figura 8*) sense haver acabat el procés d'actualització.

Si premem el botó 'Actualitzar' es passarà a la següent pàgina amb la recomanació destacada (en vermell) i sense haver-ne produït l'actualització. En la figura 10 es veu la pàgina on destaca la línia en vermell al costat de la finestra de selecció de seccions:



--- Figura 18: pàgina per actualitzar un aula amb l'advertència ---

2.2.- Implementació.-

Si partim de la capa de presentació des de la qual es mostren els resultats de la funcionalitat requerida haurem de preparar les pàgines JSP relacionades amb les tasques de crear i modificar un Aula. Les dos pàgines JSP són: *crearAula.jsp* i *actualitzarAula.jsp*. Estan guardades en el directori web amb la resta de pàgines JSP de l'aplicació.

Igualment i de manera més profunda hem de construir codi que permeti controlar el número de seccions seleccionades. En el directori *academia/vista/validator* guardarem els dos fitxers Java que implementaran el codi (veure Annexos per veure el codi dels fitxers):

ValidadorSeccionsSeleccionades i *ValidadorSeccionsSeleccionadesEtiqueta*

Precisament els fitxers de *validator* estan relacionats amb l'etiqueta personalitzada que haurem de crear per establir els valors mínim i màxim: *academia.tld*. El fitxer *ValidadorSeccionsSeleccionades* es registra en el fitxer de configuració de JSF anomenat *faces-config.xml*. Ambdós fitxers, *academia.tld* i *faces-config.xml* són en el directori *web/WEB-INF*.

Podríem resumir les tasques anteriors com **la implementació d'un validador**.

2.2.1.- L'etiqueta personalitzada.-

En JSF ja es disposa d'una llibreria d'etiquetes per les pàgines JSP, en el nostre projecte Ateneu està localitzada junt amb els fitxers de configuració i la mateixa etiqueta personalitzada, en el directori web/WEB-INF: ***c.tld***. Tot i això necessitem fabricar una de pròpia per no disposar-la a la llibreria estàndard.

La nostra etiqueta personalitzada ***academia.tld*** conté bàsicament informació sobre el nom *tag* que figura en les pàgines jsp detallades anteriorment , lloc del fitxer del validador i les dos variables que fiten el valor a validar:

```
<tag>
        <name>validarNumeroSeccions</name>
<tag-class>academia.vista.validator.ValidadorSeccionsSeleccionadesEtiqueta</tag-
class>
        <attribute>
                <name>minNum</name>
        </attribute>
        <attribute>
                <name>maxNum</name>
        </attribute>
  </tag>
```

Veure ***Annexos*** on és el fitxer *academia.tld*

2.2.2.- Pàgines *crearAula.jsp* i *actualitzarAula.jsp*.-

La implementació del validador que detecti i controli el número de seccions seleccionades en les pàgines JSP consisteix en incorporar unes poques línies de codi. Primer de tot aquesta:

```
<%@ taglib prefix="academia" uri="academia.vista.validator" %>
```

En les nostres dos pàgines és la línia 3. El nom de l'etiqueta personalitzada i el directori on són els fitxers Java. El cos del validador en ambdues pàgines és el mateix:

```
<td align="left" width="400">
<h:selectManyListbox value="#{aulaBean.idSeccioSeleccionada}" id="idSeccioSeleccionada">
<academia:validarNumeroSeccions minNum="1" />
<academia:validarNumeroSeccions maxNum="1" />
<f:selectItems value="#{applicationBean.seccioSelectItems}" id="seccions"/>
      </h:selectManyListbox>
       <h:message for="idSeccioSeleccionada" styleClass="errorMessage"/>
 </td>
```

Corresponen a les línies 82-83 en la pàgina *crearAula.jsp* i 83-84 en la *actualitzarAula.jsp*. (***veure Annexos***)

2.2.3.- Fitxers Java.-

Hem aprofitat el codi Java de validadors estàndard. En el fitxer *ValidadorSeccionsSeleccionadesEtiqueta* s'estableixen les variables que fiten el número permès de seccions seleccionades i en especial es crea el validador associat a l'etiqueta academia.

```
/**
 * Crear el validator associat amb l'etiqueta.
 * @return el validator associat amb l'etiqueta
 */
public Validator createValidator() throws JspException {
            ValidadorSeccionsSeleccionades validator =
(ValidadorSeccionsSeleccionades)super.createValidator();

            Integer minValue = FacesUtils.evalInt(this.minNum);
            if (minValue != null) {
                  validator.setMinNum(minValue.intValue());
            }

            Integer maxValue = FacesUtils.evalInt(this.maxNum);
            if (maxValue != null) {
                  validator.setMaxNum(maxValue.intValue());
            }

      return validator;
      }
```

En el fitxer *ValidadorSeccionsSeleccionades.java* que implementa la classe Validator s'estableix l'interval i les condicions de validació:

```
if (valorList.size() < this.minNum) {

                //advertiment si no s'han seleccionat un minim de seccions
        msg = "Un mínim de " + this.minNum + " seccions han de seleccionar-se.";
                }
else if (valorList.size() > this.maxNum) {

                //advertiment si s'han seleccionat mes d'un maxim de seccions
        msg = "Un màxim de " + this.maxNum + " seccions han de seleccionar-se.";
                }
```

Els missatges d'advertiment són els que es mostraran en les pàgines jsp.

2.2.4.- El fitxer de configuració faces-config.xml.-

En el fitxer de configuració de JSF anomenat *faces-config.xml* serà on registrarem el validador:

```
<validator>
        <description>
        </description>
        <display-name>
        </display-name>
        <icon>
        </icon>
        <validator-id>academia.vista.validator.SeccionsSeleccionades</validator-id>
<validator-class>academia.vista.validator.ValidadorSeccionsSeleccionades</validator-class>
        <attribute>
        </attribute>
        <property>
        </property>
 </validator>
```

Sempre que necessitem el *validator* en una aplicació web haurem de incorporar les línies anteriors del *id* i *classe* en el fitxer de configuració, els valors s'hauran de configurar d'acord amb el directori i nom de la classe. La resta de paràmetres no cal que es posin.

Veure ***Annexos*** on hem creat un fitxer de configuració propi per el validador:
validator-config.xml

2.2.5.- Fitxer web.xml.-

En el fitxer anomenat **descriptor de desplegament** *web.xml* quedarà registrat el fitxer de configuració *faces-config.xml* :

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

El fitxer *web.xml* es guarda en el directori web/WEB-INF. En aquest fitxer es defineixen les regles de seguretat, la informació per a cada *servlet* i sobre tot per el que farà de controlador, així mateix es descriu el component i quins *mappings* té. En resum, detalla com es desplegarà l'aplicació en el contenidor web. (**Capítol 3**, **p. 6**)

NOTA: si creem un fitxer de configuració *validator-config.xml* aleshores la línia de <param-value> serà:

```
      <param-value>/WEB-INF/faces-config.xml,/WEB-INF/validator-config.xml
</param-value>
```

2.3.- Resum: què hem fet i què hem aconseguit.-

Per tal d'obtenir la validació del número de seccions seleccionades hem anat al lloc on es produeix la selecció: en dos pàgines jsp. Incorporar el detector del valor fitat suposa tenir una etiqueta que no està a l'abast en la llibreria estàndard de *JSF* i aleshores hem creat abans una etiqueta personalitzada, la qual ha d'enllaçar amb el codi Java de fitxers que implementin les classes de JSF *Validator i ValidatorTag*.

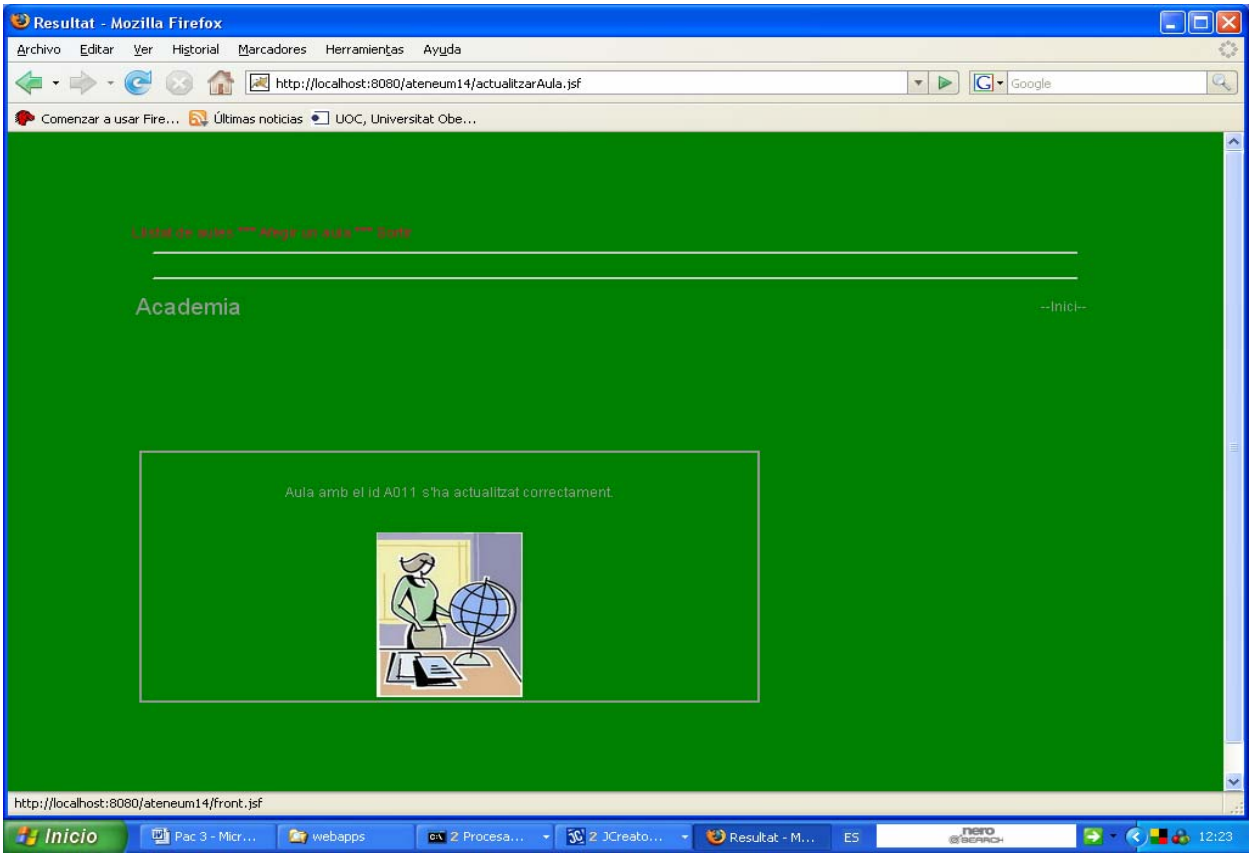
Fins aquí tindríem tota la tasca dura de edició i per completar el procediment necessitem registrar el validador en el fitxer de configuració de *JSF*, punt on considerem que es produeix realment la modificació del comportament del *framework JSF*.

Finalment l'aplicació web que es creï i pugui validar el número de seccions seleccionades haurà d'embalar el fitxer de configuració JSF en el descriptor de desplegament.

En resum **hem fet** els següents procediments:

- ✓ Crear dos classes que implementin *Validator i ValidatorTag*
- ✓ Crear una etiqueta personalitzada
- ✓ Incorporar l'etiqueta del validador a les pàgines JSP
- ✓ Registrar el validador a *faces-config.xml*
- ✓ Embalar el fitxer *faces-config.xml* en el *web.xml*

A continuació es mostra la pàgina resultat un cop el Gestor ha actualitzat correctament un Aula:



--- Figura 19: pàgina resultat actualitzar un aula ---

En el llista d'aules podríem comprovar que l'aula actualitzada ha quedat associada amb una secció, la que nosaltres jem seleccionat.

Hem aconseguit tenir en el *framework* JSF i disposar, amb les adaptacions oportunes!, per altres aplicacions web on es necessiti validar la selecció de ítems un ***validador***:

- ✓ Dos classes Java que implementen *Validator* i *ValidatorTag*
- ✓ Una etiqueta personalitzada
- ✓ Mostrar com s'incorpora l'etiqueta a les pàgines JSP
- ✓ Tenir preparat l'element <validator></validator> a *faces-config.xml*

El procediment seguit no només serveix per l'aplicació en concret Ateneu sinó per altres de similars. Nos podem trobar en el transcurs de la creació d'una aplicació web amb el requeriment d'associar d'acord amb unes condicions(que no té perquè ser sempre 1-1) un element amb un altre element o elements de categoria superior de l'estil *producte-categoria*, *llibre-biblioteca*, *assignatura-estudis*, *equips-competicions*, *tenistes-torneigs*, etc. i que aleshores pot resultar útil implementar-la mitjançant JSF i utilitzar els components i metodologia ensenyada anteriorment.

----------^----------

3.- Exemple d'utilització de la implementació del validador

Primer de tot i per si no havia quedat clar, disposem d'un validador que ***controlarà el nombre de elements que seleccionarem d'un grup X per associar-los a un element d'un altre grup Y***. Aleshores ja hem de tenir els beans o classes que implementen la associació i selecció. Nosaltres ara volem o necessitem restringir aquesta selecció. Més encara, nos agradaria gestionar-la segons uns criteris de mercat, competició, administració, etc.

Considerarem una aplicació J2EE genèrica de nom PFCAssoc on tenim en la BD dos conjunts M i N. El conjunt M conté elements A i el conjunt N elements B. Els elements B han d'estar associats a algun element A. El Gestor de l'aplicació PFCAssoc crea, actualitza i elimina elements B. Quan crea o actualitza un element B, en el procediment s'inclou l'associació de pertinència a un o més d'un element A.

L'associació és del tipus 1 (també seria possible 0) a molts però en 'molts' serà on el Gestor posarà les condicions. El Gestor l'encarrega al desenvolupador que l'associació sigui d'un mínim de 1 i d'un màxim de 3.

El desenvolupador té guardada la implementació del validador (veure fitxers en ***Annexos***) i es disposa a seguir els següents passos que no tenen perquè ser en el mateix ordre:

Operació 1:

- ✓ Obrir l'etiqueta personalitzada academia.tld i conservar/modificar els valors que conformaran els de les pàgines JSP i classes Validator
- ✓ Obrir els fitxers ValidadorSeccionsSeleccionades.java i ValidadorSeccionsSeleccionadesEtiqueta.java i conservar/modificar els valors que quedaran afectats

Els possibles canvis de noms dels valors estaran subjectes a la terminologia de l'aplicació i permetran entendre la seva relació amb els objectes de domini.

Operació 2:

Precisament és en les pàgines JSP relatives a la creació i actualització d'elements B on el desenvolupador ha de registrar l'etiqueta (taglib) a l'inici i on ha d'indicar els valors mínim i màxim de l'associació (1 i 3).

Operació 3:

Un cop el desenvolupador ha implementat l'etiqueta i les classes corresponents els valors corresponents (conservats o modificats) anem al fitxer de configuració de JSF i descriptor de desplegament:

- ✓ Obrir *faces-config.xml* i editar l'element (o incloure'l en cas de que hi sigui) validator; els valors han de correspondre als establerts en l'operació primera

- ✓ Finalment s'ha de confirmar l'existència del fitxer *faces-config.xml* en el registre de *web.xml* (dins de l'element *context-param*)

NOTA: JavaServer Faces té per defecte el fitxer de configuració *faces-config.xml* però no és necessari que sigui per aquest nom. Nosaltres podem triar un altre. Igualment podem afegir més fitxers de configuració. Així en el cas del validador si volem disposar d'un de propi per guardar les dades del validador podríem crear un anomenat *validator-config.xml* (veure ***Annexos***)

Si que després hauríem de registrar-ho junt amb *faces-config.xml* en el <context-param> del fitxer de desplegament *web.xml*. Justament en el registrament s'estableix el lloc on es guarda els fitxers de configuració que per convecció és en el directori WEB-INF que tampoc té que sé sempre aquest.

---------^----------

Capítol 6: Conclusions

Conclusions

La primera conclusió important que hem tret ha resultat l'eficàcia dels *frameworks* però també la seva especificitat, *Hibernat*e implementa la cap de persistència i *JSF* la de presentació. Així mateix hem observat en la integració de *frameworks* en una aplicació la influència que adquireix algun d'ells envers als altres i com aconsegueix millorar-los. Nos ha interessat comprovar el fet que *Spring* compensa certes mancances de *Hibernate* i *JSF*. Per exemple *Spring* ofereix un eficaç i segur control de les sessions en *Hibernate,* i en la integració de JSF amb la lògica de negoci en el cas de *ServiceLocator*.

La segona conclusió fonamental és disposar dels fitxers de configuració on podrem escriure els paràmetres que conformaran la implementació del *framework* en la aplicació. Per exemple en el fitxer *applicationContext.xml* de *Spring* es determinaran les sessions de Hibernate.

La penúltima conclusió és la de constatar que un *framework* no 've' amb totes les funcionalitats llestes per ser utilitzades en l'aplicació. Tanmateix hem comprovat que porta unes classes, eines i els fitxers de configuració que nos facilitaran la labor de la implementació de la funcionalitat requerida. Creiem que és lògic que sigui el desenvolupador qui completi la part del framework no porta a punt. És el desenvolupador qui ha de personalitzar l'aplicació.

Finalment hem tret la conclusió per el que aquest PFC tenia raó d'existència: la implementació d'una funcionalitat en un framework en una aplicació concreta pot servir-nos per altres aplicacions. El validador implementat servirà per aplicacions on s'ha de fitar el nombre mínim i màxim de elements A que s'han de seleccionar per cada element B en el procés de crear o actualitzar un element B.

----------^----------

Glossari

applicationContext.xml : Fitxer de configuració de ***Spring***

Base de Dades (BD): Una BD pot ser definida com una col·lecció estructurada de registres o dades guardades en un computador per tal que un programa pugui consultar-les en les peticions o cerques (*queries*) que se li fan.

capa de lògica de negoci: És una de les habituals capes en una arquitectura multi-capa. Separa la lògica de negoci dels altres mòduls com és la capa d'accés a les dades i de la interfície d'usuari. Aquesta separació li permet no quedar afectada per els canvis efectuats en les altres dos capes.

capa de persistència: On es guarden les dades persistents, és a dir que sobreviuen l'execució del programa o les sessions d'usuari. El format d'emmagatzematge no volàtil pot ser un sistema de BD relacional.

capa de presentació : També anomenada ***capa web*** en *J2EE*, té com objectiu fonamental facilitar l'accés de la lògica de negoci a clients mitjançant el protocol *HTTP* (per exemple un navegador).

descriptor de desplegament (DD): És un component en aplicacions J2EE que descriu com una aplicació web ha de desplegar-se. Dirigeix una eina de desplegament de mòduls o de tota l'aplicació amb opcions per un contenidor específic i descriu requeriments concrets de configuració que el desplegador ha de resoldre. En J2EE s'ha de dir ***web.xml*** i guardat en el directori *WEB-INF* sota l'arrel de l'aplicació.

etiqueta: Fa referència als elements HTML i en molts casos només a l'inici i final de l'element. En el PFC l'etiqueta és el fitxer (*academia.tld*) que guarda l'element validador per les JSP.

faces-config.xml : Fitxer de configuració de ***JavaServer Faces***

fitxer hbm.xml : Fitxer de ***mapeig*** que utilitza *Hibernate* en llenguatge *XML*. Per convenció és disposar d'un fitxer de mapeig per cada classe. Així la classe *Aula.java* tindrà el fitxer de mapeig *Aula.hbm.xml*.

fitxer de configuració: Són utilitzats per configurar els valors establerts (*settings)* inicials en determinats programes informàtics. Contenen els *settings* de les aplicacions d'usuari, processos del servidor i sistemes operatius.

Actualment XML és el llenguatge més emprat en els fitxers de configuració. En el present PFC hem vist uns quants: ***struts-config.xml***, ***faces-config.xml*** o ***validator-config.xml***.

framework : És un disseny reutilitzable per un sistema de programari. Pot incloure llibreries de codi, llenguatge de scripting, o fins i tot altre programari per ajudar a desenvolupar i integrar tot plegat els diferents components d'un projecte de programari. A destacar que les parts d'un framework són exposades en APIs.

framework d'aplicació web: Dissenyat per donar suport al desenvolupament de webs dinàmiques, aplicacions i serveis web. L'objectiu fonamental és alleugerar la sobrecàrrega associada a activitats comunes i repetitives habituals en el desenvolupament d'una aplicació web com l'accés a una BD o la gestió de les sessions.

Hibernate : *Framework* d'aplicació web. És una solució en llenguatge *Java* per el mapeig O/R i proveeix d'una facilitat en la utilització del *framework* en el mapeig del model de domini orientat a objectes a una tradicional BD relacional. El propòsit de *Hibernate* és lliurar al desenvolupador de programar tasques relacionades amb la persistència de dades. És de codi obert desenvolupat per *Red Hat*.

JavaServer Faces(JSF) : *Framework* d'aplicació web basat en llenguatge *Java*. Simplifica el desenvolupament de interfícies d'usuari per aplicacions J2EE. En la seva tecnologia per mostrar la web utilitza les pàgines *JSP*.

JSP : És una tecnologia Java que permet als desenvolupadors la generació dinàmica de documents ***HTML***, ***XML*** o d'altres tipus en resposta a una petició del client web. JSP facilita que el codi Java i certes accions predefinides estiguin incrustades en el contingut estàtic. A destacar que JSP també facilita la creació de llibreries d'etiquetes JSP que actuen com extensió de les etiquetes estàndard ***HTML*** o ***XML***.

MVC: Acrònim per ***Model-Vista-Controlador***, patró d'arquitectura en enginyeria de programari. En una aplicació complexa que presenta una gran quantitat de dades a l'usuari, és desitjable separar les dades (Model) i els problemes de la interfície d'usuari (Vista), de tal manera que els canvis a la interfície no afectin el maneig de les dades, i que les dades puguin ser organitzades sense canviar la interfície d'usuari. MVC soluciona aquest problema desacoblant l'accés a les dades i la lògica de negoci de la presentació de les dades i interacció de l'usuari, mitjançant un component intermediari: el Controlador.

Spring : És un *framework* d'aplicació que suporta moltes tecnologies de generació de vistes com les JSP i s'integra perfectament amb *Hibernate*. *Spring* és de codi obert per la plataforma *Java*, aplica els principis de Inversió de Control utilitzant la tècnica de Injecció de dependències.

Struts: *Apache Struts* és un *framework* d'aplicació web de codi obert per desenvolupar aplicacions J2EE. Utilitza i estén Java Servlet API per permet als desenvolupadors adoptar el patró MVC (model-vista-controlador). La principal avantatge de *Struts* és que separa el Model (la lògica de negoci que interactua amb la BD) de la Vista (pàgines HTML mostrades al client) i el Controlador (instància que passa informació entre Vista i Model).

validador : Utilitzat en un programa per comprovar la correcció sintàctica d'un fragment de codi o document. El terme s'empra en el context de validació de documents en *HTML*, *CSS* o *XML*, però també es fa servir per comprovar valors, elements seleccionats, etc en pàgines *JSP* entrats per l'usuari.

----------^----------

Bibliografia

Base de Dades I (UOC)
---------- *Jaume Sistac Planas*
Enginyeria de Programari I (UOC)
----- *Benet Campderrich Falgueras*
Enginyeria de Programari II (UOC)
----- *Ma.Jesus Marco Galindo*
Enginyeria de Programari orientat a l'objecte (UOC)
------- *Jordi Cabot Sagrera*
------- *Isabel Guitart Hormigo*
Enginyeria del programari de components i sistemes distribuïts (UOC)
------ *Jordi Cabot Sagrera*
Java 2 Curso de Programación (Ra-Ma)
----- *Fco. Javier Ceballos*
Java 2 Interfaces gráficas y aplicaciones para internet (Ra-Ma)
----- *Fco. Javier Ceballos*
Anàlisis y Diseño orientado a objetos con UML y el proceso unificado (McGraw-Hill) ----- *Stephen R. Schach*
J2EE Desarrollo de aplicaciones Web (eni ediciones)
---- *Benjamín Aumaille*
Jakarta Struts – Pocket Reference(O'Reilly)
----- *Chuck Cavaness/Brian Keeton*
JavaServer Faces – The Complete Reference (McGraw-Hill)
----- *Chris Schalk/Ed Burns*
Java Development with the Spring Framework (Wiley)
----- *Rod Johnson/Juergen Hoeller*
Hibernate in Action (Hanning)
-------- *Christian Bauer/Gavin King*

----------^----------

Internet

Tema: Construcció d'Aplicacions de Programari Lliure en J2EE/Tècniques avançades: mapatge objecte-relacional, test unitaris i inversió de control
Web: ----------*http://ca.wikibooks.org/*
Tema: Tutorials per Hibernate, Struts i JavaServerfaces (JSF)
Web:---------- *http://www.lalilluna.de/first-java-server-faces-tutorial.html*
Tema: Integració de frameworks (JSF, Spring, Hibernate)
Web:---------- *http://www.javaworld.com/*
Tema: Tutorial sobre JSF
Web:---------- *http://www.jsftutorials.net/*
Tema: Mapeig O/R a Hibernate i instal·lació driver MySQL
Web:*http://www.regdeveloper.co.uk/2005/12/03/hibernate_object_relational_mapping/page2.html*
Tema: Integració de Spring amb Hibernate
Web:----------*http://www.springframework.org/docs/reference/orm.html*
Tema: Integració de Spring amb JSF
Web:--- *http://www.springframework.org/docs/reference/webintegration.html*
Tema: Configuració de securityfilter-config.xml
Web:
http://servlets.com/archive/servlet/ReadMsg?msgId=481192&listName=struts-user
Tema: Tutorial sobre J2EE (Sun Microsystems)
Web:-------- *http://java.sun.com/j2ee/1.4/docs/tutorial/doc/*
Tema: Cicle de vida de JSF
Web: http://www.ibm.com/developerworks/library/j-jsf2/
Tema: Introducció a Spring
Web:*http://www.myeclipseide.com/images/tutorials/quickstarts/springintroduction/tutorial.html*
Tema: Patrons de presentació
Web:
http://www.solomanuales.org/frame.cfm?url_frame=http://www.programacion.com/java/tutorial/patrones
Tema: Terminologia, definicions sobre els frameworks
Web: *http://es.wikipedia.org/wiki/*
Web: *http://ca.wikipedia.org/wiki/*
Web: *http://en.wikipedia.org/wiki/*
Tema: Tecnologies
Web:
http://oness.sourceforge.net/proyecto/html/ch06s03.html#spring_overview

Annexos

Fitxer ***ValidadorSeccionsSeleccionades.java***:

```
/*
 * Projecte Ateneu
 */
package academia.vista.validator;

import java.util.List;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
import javax.faces.FacesException;

/**
 * Classe validator JSF per establir l'interval de validacio
 * Valida el numero de seccions associades a una aula
 * Consta de dos parametres: maxim i minim
 * <li>minNum: el minim de seccions
 * <li>maxNum: el maxim de seccions
 * @author Luis Gallego Ruestes
 */
public class ValidadorSeccionsSeleccionades implements Validator {
//minima fita
private int minNum = Integer.MIN_VALUE;
//maxima fita
private int maxNum = Integer.MAX_VALUE;
/**
 * Metode principal
 */
public void validate(FacesContext context, UIComponent component, Object valor) {
if (valor == null) {
return;
}
List valorList = null;
try {
valorList = (List)valor;
}
catch (Exception e) {
throw new FacesException("UISelectManyValidator només es relaciona amb el component
UISelectMany.");
}

String msg = null;
if (valorList.size() < this.minNum) {
```

```
//advertiment si no s'han seleccionat un minim de seccions
msg = "Un mínim de " + this.minNum + " seccions han de seleccionar-se.";
}
else if (valorList.size() > this.maxNum) {
//advertiment si s'han seleccionat mes d'un maxim de seccions
msg = "Un màxim de " + this.maxNum + " seccions han de seleccionar-se.";
}

if (msg != null) {
FacesMessage message = new FacesMessage(FacesMessage.SEVERITY_ERROR, msg, msg);
throw new ValidatorException(message);
}
}
public void setMinNum(int nouMinNum) {
this.minNum = nouMinNum;
}
public void setMaxNum(int nouMaxNum) {
this.maxNum = nouMaxNum;
}
}
```

----------^----------

Fitxer ***ValidadorSeccionsSeleccionadesEtiqueta.java***:

```
/*
 * Projecte Ateneu
 */
package academia.vista.validator;

import javax.faces.validator.Validator;
import javax.faces.webapp.ValidatorTag;
import javax.servlet.jsp.JspException;
import academia.vista.util.FacesUtils;

/**
 * Personalitzacio de l'etiqueta per ValidadorSeccionsSeleccionades.
 * @author Luis Gallego Ruestes
 * @see ValidadorSeccionsSeleccionades
 */
public class ValidadorSeccionsSeleccionadesEtiqueta extends ValidatorTag {
//el identificador de validator registrat a JSF, veure faces-config.xml
private static String ID_VALIDATOR = "academia.vista.validator.SeccionsSeleccionades";


//fita minima
private String minNum;
```

```
//fita maxima
private String maxNum;

/**
 * Constructor per defecte.
 */
public ValidadorSeccionsSeleccionadesEtiqueta() {
this.setValidatorId(ID_VALIDATOR);
}

public void setMinNum(String nouMinNum) {

this.minNum = nouMinNum;
}

public void setMaxNum(String nouMaxNum) {

this.maxNum = nouMaxNum;
}

/**
 * Crear el validator associat amb l'etiqueta.
 * @return el validator associat amb l'etiqueta
 */
public Validator createValidator() throws JspException {

ValidadorSeccionsSeleccionades validator =
(ValidadorSeccionsSeleccionades)super.createValidator();

Integer minimValor = FacesUtils.evalInt(this.minNum);
if (minimValor != null) {
validator.setMinNum(minimValor.intValue());
}

Integer maximValor = FacesUtils.evalInt(this.maxNum);
if (maximValor != null) {
validator.setMaxNum(maximValor.intValue());
}

return validator;
}

public void release() {
this.minNum = null;
this.maxNum = null;
}
}
```

----------^----------

Fitxer ***academia.tld:***

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>academia</short-name>
  <uri>academia.vista.validator</uri>
  <display-name>Academia Custom Tag</display-name>
  <description>Etiqueta personalitzada per Academia</description>
  <tag>
  <name>validarNumeroSeccions</name>
<tag-class>academia.vista.validator.ValidadorSeccionsSeleccionadesEtiqueta</tag-class>
<attribute>
<name>minNum</name>
</attribute>
<attribute>
<name>maxNum</name>
</attribute>
  </tag>
</taglib>
```

Fitxer ***validator-config.xml:***

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config
1.0//EN"
        "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
<validator>
<validator-id>academia.vista.validator.SeccionsSeleccionades</validator-id>
<validator-class>academia.vista.validator.ValidadorSeccionsSeleccionades</validator-class>
  </validator>
</faces-config>
```

Fitxer ***crearAula.jsp***:

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@ taglib prefix="academia" uri="academia.vista.validator" %>
<html>
<head>
<title>Alta nova Aula</title>
<link rel="stylesheet" type="text/css" href="stylesheet.css"/>
<f:loadBundle basename="academia.vista.bundle.Messages" var="msgs"/>
   </head>
   <body>
<f:view>
<%@ include file="header.jsp" %>
         <h:form id="createAulaForm">
<table align="center" width="500">
 <tr>
   <td align="left">
<h:outputText value="Donar d'alta una nova aula" styleClass="headerText"/>
 </td>
     </tr>
         <tr>
          <td align="right" width="100">
         <h:outputText value="Id"/>
         </td>
         <td align="left" width="400">
                  <h:inputText value="#{aulaBean.id}" id="id" required="true"/>
              <h:message for="id" styleClass="errorMessage"/>
         </td>
        </tr>
           <tr>
         <td align="right" width="100">
         <h:outputText value="Nom"/>
         </td>
         <td align="left" width="400">
          <h:inputText value="#{aulaBean.nom}" id="nom" required="true"/>
              <h:message for="nom" styleClass="errorMessage"/>
         </td>
           </tr>
           <tr>
            <td align="right" width="100">
         <h:outputText value="Matrícula (euros)"/>
           </td>
         <td align="left" width="400">
              <h:inputText value="#{aulaBean.matricula}" id="matricula"/>
              <h:message for="matricula" styleClass="errorMessage"/>
          </td>
           </tr>
            <tr>
```

```
        <td align="right" width="100">
         <h:outputText value="Horari"/>
           </td>
         <td align="left" width="400">
             <h:inputText value="#{aulaBean.horari}" id="horari"/>
              <h:message for="horari" styleClass="errorMessage"/>
             </td>
              </tr>
            <tr>
            <td align="right" width="100">
           <h:outputText value="Professor"/>
            </td>
         <td align="left" width="400">
            <h:inputText value="#{aulaBean.professor}" id="professor"/>
           <h:message for="professor" styleClass="errorMessage"/>
          </td>
           </tr>
            <tr>
            <td align="right" width="100" valign="bottom">
             <h:outputText value="Descripció"/>
             </td>
         <td align="left" width="400">
       <h:inputTextarea value="#{aulaBean.descripcio}" id="descripcio" rows="8" cols="16"/>
              <h:message for="descripcio" styleClass="errorMessage"/>
              </td>
          </tr>
            <tr>
              <td align="right" width="100" valign="bottom">
          <h:outputText value="Seccions"/>
             </td>
          <td align="left" width="400">
<h:selectManyListbox value="#{aulaBean.idSeccioSeleccionada}"
id="idSeccioSeleccionada">
<academia:validarNumeroSeccions minNum="1"/>
<academia:validarNumeroSeccions maxNum="1"/>
<f:selectItems value="#{applicationBean.seccioSelectItems}" id="seccions"/>
</h:selectManyListbox>
 <h:message for="idSeccioSeleccionada" styleClass="errorMessage"/>
           </td>
          </tr>
           <tr>
          <td align="center" colspan="2">
           <h:commandButton value="Crear" action="#{aulaBean.createAction}"/>
              <h:commandButton value="Cancel·lar" action="cancel" immediate="true"/>
           </td>
          </tr>
             <tr >
          <td align="left" colspan="2">
               <h:messages styleClass="errorMessage" globalOnly="true"/>
              </td>
            </tr>
        </table>
          </h:form>
          </f:view>
  </body>
</html>
```

Fitxer ***actualitzarAula.jsp***:

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@ taglib prefix="academia" uri="academia.vista.validator" %>

<html>
<head>
   <title>Modificació Aula</title>
     <link rel="stylesheet" type="text/css" href="stylesheet.css"/>
   </head>
   <body>
<f:view>
      <%@ include file="header.jsp" %>
         <h:form id="editAulaForm">
         <table align="center" width="500">
          <tr>
         <td align="left">
              <h:outputText value="Actualitzar Aula" styleClass="headerText"/>
          </td>
          </tr>
              <tr>
         <td align="right" width="100">
            <h:outputText value="Id"/>
         </td>
         <td align="left" width="400">
      <h:outputText value="#{aulaBean.id}" id="id"/>
              <h:inputHidden value="#{aulaBean.id}" id="hiddenId"/>
          </td>
              </tr>
               </tr>
             <tr>
             <td align="right" width="100">
         <h:outputText value="Nom"/>
         </td>
         <td align="left" width="400">
             <h:inputText value="#{aulaBean.nom}" id="nom" required="true"/>
               <h:message for="nom" styleClass="errorMessage"/>
            </td>
           </tr>
            <tr>
              <td align="right" width="100">
         <h:outputText value="Matrícula (euros)"/>
         </td>
        <td align="left" width="400">
           <h:inputText value="#{aulaBean.matricula}" id="matricula"/>
              <h:message for="matricula" styleClass="errorMessage"/>
        </td>
          </tr>
           <tr>
            <td align="right" width="100">
         <h:outputText value="Horari"/>
        </td>
```

```
  <td align="left" width="400">
              <h:inputText value="#{aulaBean.horari}" id="horari"/>
               <h:message for="horari" styleClass="errorMessage"/>
          </td>
             </tr>
             <tr>
             <td align="right" width="100">
          <h:outputText value="Professor"/>
            </td>
             <td align="left" width="400">
               <h:inputText value="#{aulaBean.professor}" id="professor"/>
               <h:message for="professor" styleClass="errorMessage"/>
             </td>
           </tr>
             <tr>
              <td align="right" width="100" valign="bottom">
          <h:outputText value="Descripció"/>
         </td>
          <td align="left" width="400">
       <h:inputTextarea value="#{aulaBean.descripcio}" id="descripcio" rows="8" cols="16"/>
               <h:message for="descripcio" styleClass="errorMessage"/>
          </td>
            </tr>
             <tr>
               <td align="right" width="100" valign="bottom">
                 <h:outputText value="Seccions"/>
              </td>
               <td align="left" width="400">
 <h:selectManyListbox value="#{aulaBean.idSeccioSeleccionada}"
id="idSeccioSeleccionada">
   <academia:validarNumeroSeccions minNum="1"/>
<academia:validarNumeroSeccions maxNum="1"/>
           <f:selectItems value="#{applicationBean.seccioSelectItems}"
id="seccions"/>
             </h:selectManyListbox>
              <h:message for="idSeccioSeleccionada" styleClass="errorMessage"/>
             </td>
           </tr>
           <tr>
            <td align="center" colspan="2">
               <h:commandButton value="Actualitzar" action="#{aulaBean.updateAction}"/>
                <h:commandButton value="Cancel·lar" action="cancel" immediate="true"/>
            </td>
           </tr>
           <tr >
           <td align="left">
                <h:messages errorStyle="errorMessage" globalOnly="true"/>
           </td>
             </tr>
        </table>
          </h:form>
           </f:view>
  </body>
</html>
```

----------^----------