PFC

Disseny i implementació d'un marc de treball de presentació per aplicacions J2EE

Alumne: Pasqual Castellón Gadea
Tutor: Josep Maria Camps Riba
Curs: 2n cicle enginyeria Informàtica 2012/2013

Dedicatòries

Aquest treball el dedico a totes aquelles persones que en algun moment de la meva formació i vida professional m'han aportat els seus coneixements i m'han animat a continuar endavant.

Així mateix i de forma molt especial, agraeixo a la meva família el suport incondicional i confiança que sempre han demostrat en mi.

Per últim i sobretot li vull dedicar aquest treball a la meva dona, que ha hagut de patir les llargues hores de soledat que m'han exigit els estudis.

Àrea PFC

Disseny i implementació d'un marc de treball (framework) de presentació per aplicacions J2EE

Paraules clau

MVC, Spring, presentation framework java.

Resum

El patró Model-Vista-Controlador és actualment el més utilitzat per els desenvolupadors en la creació pàgines web complexes i dinàmiques. J2EE és un estàndar de la indústria de desenvolupament composat per un gran nombre de mòduls i llibreries multipropòsit que conformen una API complerta. Amb l'ús d'aquestes dues tecnologies, el projecte proposa dissenyar un framework de presentació que incorpori alguna millora respecte les actuals solucions existents en el mercat.

Existeixen un àmpli nombre de frameworks de presentació actualment: Spring, Struts2, JSF, Seam, Tapestry o Wicket serien alguns exemples. Tots ells conformen eines prou evolucionades i desenvolupades per crear aplicacions web, però tenen entre si notables diferències, relacionades amb la capacitat de configuració, el funcionament intern, el fluxe de dades entre la vista i el model, o l'accesibilitat a l'usuari.

Un framework de presentació ha d'incloure una sèrie de capacitats per tal de poder ser considerat adequat per el desenvolupament d'aplicacions web, com serien: l'ús de manejadors, l'ús d'interceptors, validació de formularis, validació de credencials, gestió de la internacionalització, etc ...

Proposarem crear un nou framework de presentació sota el nom de Winter MVC que, prenent com a punt de partida el complert framework de presentació Spring MVC, incorporarà una classe POJO de configuració que permetrà eliminar els arxius de configuració XML que incorporen una sintaxi estricta i difícil de recordar, reduint així la seva corba d'aprenentatge.

Índex de continguts

Índex de figures

Pla de treball

1. Introducció

El desenvolupament de *frameworks* és potser una de les tasques més agraïdes per un programador donat que un cop finalitzats resulten útils tant al propi desenvolupador com a tercers en la creació de noves aplicacions, la qual cosa sempre és reconfortant. Els *frameworks*, son un conjunt d'eines creades per facilitar el desenvolupament d'aplicacions dins d'un àmbit concret. En el projecte que es presenta, ens centrarem en els *frameworks* de la capa de presentació per aplicacions *J2EE*.

2. Objectius del projecte

Els objectius a assolir en aquest projecte és, com indica el seu títol, desenvolupar un *framework* destinat a facilitar la implementació de la capa de presentació web en aplicacions que treballin amb la plataforma *J2EE*. Com a objectiu secundari, em proposo entendre i dominar el lèxic i comportament relatiu a les aplicacions *J2EE* , comprendre la funció de cadascun dels elements que la conformen i adquirir suficient agilitat en el seu desenvolupament. També aspiro a millorar algun dels productes existents en el mercat en un o més àmbits concrets.

3. Planificació del projecte

La planificació proposada per desenvolupar el projecte, inclou les fites corresponents a cadascuna de les entregues, dividides cada una d'elles en una sèrie de subtasques:

Tasca	Inici	Final
PAC2		
Estudi comparatiu	11/oct/2012	20/oct/2012
Anàlisi i abast	21/oct/2012	29/oct/2012
Disseny	30/oct/2012	08/nov/2012
PAC3		
Implementació	09/nov/2012	09/dec/2012
Proves	01/dec/2012	17/dec/2012
App d'exemple	10/dec/2012	17/dec/2012
LLIURAMENT		
Memòria	18/dec/2012	10/gen/2013
Presentació	11/gen/2013	14/gen/2013

Taula de tasques i terminis 1

Estudi Comparatiu

Entendre la complexitat del projecte és sens dubte , el primer dels problemes que haurem d'afrontar. Potser per la poca experiència del autor en el desenvolupament de aplicacions *J2EE* i en concret en l'ús de *frameworks* de presentació, aquesta tasca servirà per definir el context i familiaritzar-se amb la terminologia i eines utilitzades actualment. Durant l'estudi comparatiu s'hauran de crear petites aplicacions utilitzant diferents *frameworks* existents per copsar les diferències més rellevants i estudiar les possibles millores. Aprofundirem en l'estudi de dos dels *frameworks* més utilitzats avui dia: *Struts2* i *Spring*, sense deixar d'esmentar altres que han adquirit un elevat nivell de popularitat en els darrers anys degut sobretot al seu alt nivell de productivitat.

Anàlisi i abast

Es requereix realitzar un bon anàlisi del problema per definir les possibles solucions. Per la naturalesa del projecte, aquesta és potser una de les tasques a realitzar més importants. De la mateixa forma, és el moment ideal per delimitar els objectius del projecte i especificar el seu abast.

Disseny

El disseny del *framework* ens servirà de punt de partida per la implementació i en gran mesura l'utilitzarem posteriorment per seguir l'estructura determinada.

Implementació

La implementació traslladarà el disseny definit al codi necessari per desenvolupar el nostre *framework* de presentació.

Proves

Aquest apartat es cabdal per garantir la funcionalitat de la implementació i detectar possibles problemes que es podrien corregir en la fase de implementació o inclús de disseny, per la qual cosa ens obligarà a retrocedir sovint en la línia de projecte.

Aplicació d'exemple

El desenvolupament de l'aplicació d'exemple, complementarà la fase de proves, i ens servirà per acabar de perfilar els detalls d'ús del *framework* en entorns reals.

Memòria

El desenvolupament de la memòria constituirà la redacció del projecte, la definició del seu abast, el desenvolupament les possibles ampliacions amb les conclusions.

Presentació

Finalment, realitzarem la presentació formal utilitzant un conjunt de diapositives i un resum de la memòria.

Producte lliurable

El producte final ha estat desenvolupat amb el IDE Spring Tool Suite i es lliura en forma de projecte eclipse amb la seguent estructura:

/src/main/java/org/uoc/winter/anotacions
/src/main/java/org/uoc/winter/enumeracions
/src/main/java/org/uoc/winter/listeners
/src/main/java/org/uoc/winter/mvc
/src/main/java/org/uoc/winter/renderers
/src/main/java/org/uoc/winter/utils
/src/main/java/org/uoc/winter/validacions
/src/main/java/webapp/views
/src/main/java/webapp/web.xml

Les llibreries necessàries per executar el programa son:

Servlet-api-2.5.jar
Jsp-api-2.1.jar
Jstl-1.2.jar
Validation-api-1.0.0.GA.jar
Log4j-1.2.17.jar
Javassist-3.12.1.GA.jar

4. Diagrama de Gantt

El diagrama de Gantt mostra gràficament la planificació del projecte segons els terminis especificats a la taula de tasques i terminis.

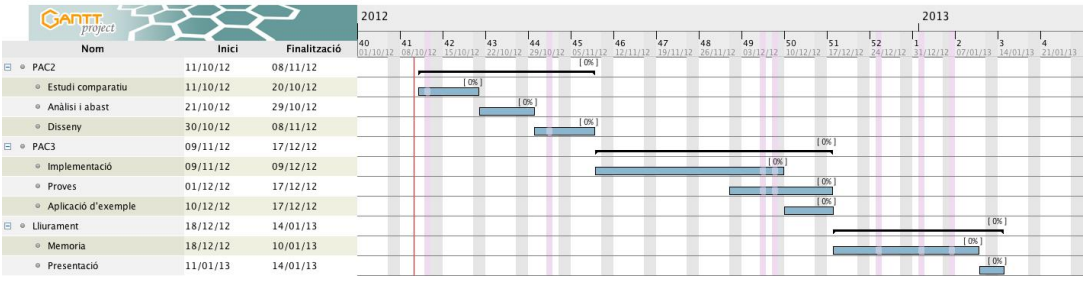


Diagrama de Gantt 1

Estudi comparatiu

1. Introducció

Actualment existeixen en el mercat una gran quantitat de marcs de treball destinats al desenvolupament d'aplicacions *J2EE*, degut a la creixent popularitat i creació de noves aplicacions web dinàmiques amb un gran component de interacció amb l'usuari. Producte d'això és que en els darrers anys han aparegut infinitat de noves tecnologies orientades a facilitar aquesta tasca. En aquest capítol ens centrarem principalment en l'anàlisi i estudi comparatiu d'aquelles dirigides a facilitar la presentació de les aplicacions, és a dir, d'aquells *frameworks* que ajudaran a processar, validar i presentar les dades necessàries per treballar amb l'aplicació. El projecte es centra en aquests tres àmbits tot i que resulta igualment necessari enllaçar aquestes eines amb una capa de dades que ens permetin un cert grau de persistència.

En línies generals, els *frameworks* de presentació segueixen el patró MVC (Model-Vista-Controlador) que s'ha imposat com la solució més eficaç. Aquest patró consisteix en separar clarament tres aspectes de l'aplicació: Les vistes (el que es mostra a l'usuari), el model (on es desenvolupa la capa de negoci de l'aplicació) i els controladors (components que actuen d'enllaç entre les vistes i el model). Aquest esquema proporciona nombroses avantatges sobretot en l'assignació de rols previ al desenvolupament de l'aplicació, però també facilita el manteniment i l'escalabilitat de l'aplicació, desacoblant al màxim els diferents components que la conformen. Aquesta seria per tant la idea principal en la que es fonamenten actualment els *frameworks* de presentació i les aplicacions J2EE en general.

2. Java Enterprise Edition

La plataforma coneguda com *Java Enterprise Edition* (*J2EE*) va ser creada originàriament per *Sun Microsystems* i adoptada en versions posteriors per *Oracle*. Fins a la versió 1.4, va conservar la terminologia *Java 2 Platform (J2EE)* per posteriorment convertir-se en *Java EE 5* i l'actual *Java EE 6*. Es tracta d'una plataforma de programació per desenvolupar aplicacions *Java* utilitzant arquitectures de múltiples nivells i es basa en components que s'executen sobre un servidor d'aplicacions. Està considerada com un estàndard de la indústria informàtica composada per un gran nombre d'especificacions amb funcionalitats diverses (*JDBC*, *RMI*, *e-mail*, *Serveis WEB*, *XML*, *servlets*, validació, decoració, *javaBeans*, etc.). Totes elles conformen una API molt complerta, perfectament modularitzada i ben documentada i que amplia les funcionalitats de la API de *Java Standard Edition*. Tot i que es tracta d'una plataforma multipropòsit per desenvolupar i executar aplicacions *Java*, s'ha convertit en un dels estàndards més utilitzats en el desenvolupament d'aplicacions WEB sobretot quan requereixen d'un accés a base de dades, però també per motius de distribució de components, manteniment, escalabilitat i accés remot.

El conjunt d'eines que ofereix la plataforma *Java EE* és de per sí suficient per cobrir totes les necessitats de programació *Java* actuals. Tot i això, diverses companyies i grups de desenvolupament han creat *frameworks* i entorns de desenvolupament que faciliten l'ús de l'API automatitzant processos, agilitzant el desenvolupament, millorant la llegibilitat del codi o inclús estalviant la necessitat de reproduir reiteradament peces de codi en diferents parts de l'aplicació. Aquests marcs de treball aconsegueixen millorar l'experiència del desenvolupador oferint un grau d'accessibilitat més confortable i s'haurien de veure com una API de nivell superior a la API que ofereix *Java EE*, però també com un paradigma de programació que incorpora patrons de disseny ben definits.

3. La capa de presentació

Quan parlem del disseny d'una aplicació amb una arquitectura per capes, observem principalment tres capes que conformen la presentació (visualització), el negoci (lògica de l'aplicació) i la persistència (base de dades) de l'aplicació. Aquest esquema permet separar perfectament els tres components evitant el seu acoblament el que facilitarà el manteniment i l'escalabilitat. De totes tres, la capa de presentació és la que segurament conté potencialment més riscos en forma de errors o problemes de seguretat degut a la necessària interactuació amb l'usuari. En una aplicació WEB la capa de presentació té com a principal objectiu fer accessible la lògica de negoci al client a través de peticions realitzades per el protocol HTTP. De forma més concreta, les funcionalitats que hauria de realitzar serien:

1) Permetre l'accés a la capa de negoci recollint les peticions, cridant al mètode de negoci que correspongui i presentant els resultats.
2) Generar contingut web dinàmic.
3) Gestionar el flux de presentació i decidir la següent pantalla a mostrar en funció de les entrades del client i l'execució realitzada a la capa de negoci.
4) Control de validació de les entrades del client.
5) Mantenir l'estat entre diferents peticions al protocol HTTP.
6) Oferir compatibilitat sobre diferents plataformes. (navegador, dispositius mòbils…)

Històricament, el disseny de la capa de presentació ha passat per diferents etapes:

- Capa de presentació sense MVC
  Generació de contingut estàtic o dinàmic amb l'ajuda de *CGI*'s que representaven la capa de negoci, però a més generaven la sortida directament i controlaven el flux de presentació.

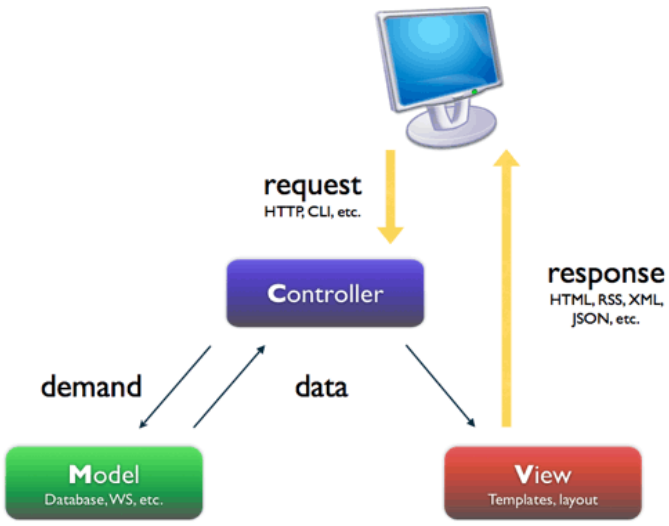
- MVC amb Model-1
  Estructuració del model utilitzant tecnologia *JSP* que permet integrar crides *java* al servidor dins d'un format de pàgina web amb la utilització de *JavaBeans*.

- MVC amb Model-2
  Separació de la capa de presentació i la capa de negoci amb la utilització de *servlets* i pàgines *JSP*. Apareix un element, el 'Controlador' que centralitzarà el flux de presentació i realitzarà les crides necessàries a la lògica de negoci.

- *Framework* basat en Model-2
  L'evolució de la tecnologia permet utilitzar *frameworks* que ofereixen un ampli nombre d'avantatges:

  - Permet separar perfectament la capa de presentació, de les de negoci i persistència.
  - Simplifica i estandaritza la validació de dades d'entrada.
  - Simplifica la gestió del flux de navegació.
  - Proporciona un punt central de control (Controlador).
  - Reutilització de les funcions més habituals.
  - Estandaritza l'arquitectura entre diferents desenvolupaments.
  - Simplificació de tasques repetitives.

La següent imatge descriu la relació entre els diferents elements que conformen el patró *MVC*.



Patró MVC

Observem la participació dels tres elements principals que conformen aquest patró. El client envia una petició HTTP al controlador. Aquest rep la petició i utilitza les dades de la petició per treballar amb el Model, on es situa la lògica de negoci i la persistència. Un cop finalitzat i segons el resultat de les operacions, el Controlador pot decidir la següent vista a mostrar, que a més retornarà una possible resposta en el client i el flux de funcionament tornarà a començar.

Aquest esquema compleix el paradigma del patró MVC aïllant perfectament la presentació de l'aplicació formada per les vistes de les capes de negoci i persistència. El nexe d'unió entre aquests dos nivells el realitza el Controlador, que processa les peticions HTTP i actua en conseqüència traslladant les dades a les capes de negoci i persistència.

Tot i què actualment l'estàndard de desenvolupament recomana utilitzar un sistema de *framework* en Model 2 (MVC) la idoneïtat d'aquesta alternativa s'ha de valorar en cada cas i en funció de la complexitat de l'aplicació que es desitja implementar.

4. Solucions actuals

Com ja s'ha comentat anteriorment, existeixen a l'actualitat un ampli ventall de *frameworks* dirigits al desenvolupament d'aplicacions *J2EE* i en concret centrats en la capa de presentació que segueixen el Model-2 *MVC*. A continuació explicarem els detalls dels més importants:

- Spring MVC

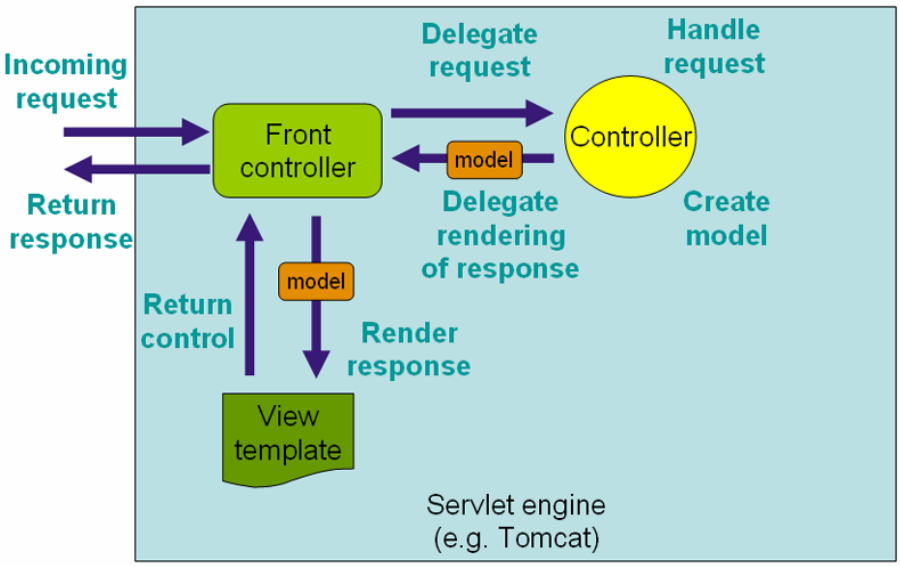
Spring és un complert producte basat en desenvolupament de codi obert (tot i que el manteniment del *framework* està centralitzat en una única comunitat), que inclou un gran nombre de mòduls que ofereixen diverses funcionalitats i que cobreixen tots els processos de desenvolupament de l'aplicació. Cal indicar que és tracta en realitat d'un *framework* de caràcter general per aplicacions *J2EE* i per tant engloba funcionalitats destinades també a la resta d'aspectes de l'aplicació, com són les capes de negoci i persistència. La part d'*Spring* dedicada a la capa de presentació, s'anomena *Spring MVC* i és en si, un mòdul més del *framework*. De fet l'ús de mòduls independents n'és una de les característiques principals però també destaca a seva potència, versatilitat i flexibilitat. En aquest sentit, el desenvolupador pot crear mòduls i incorporar-los al sistema si té la necessitat, o be reutilitzar els que venen amb el producte, però el conjunt d'eines incloses en aquest marc de treball és suficient en la majoria de les ocasions. En contrapartida, aquest marc de treball necessita d'un temps d'aprenentatge previ força elevat.

El funcionament es basa en l´ús d'un *servlet* central , el '*DispatcherServet*' que hereda directament de *HttpRequest*, i que s'encarrega de gestionar les peticions de l'usuari. En aquest sentit, el funcionament de *Spring* segueix el flux següent: El client envia la petició (*request*) al *DispatcherServlet* que la delega al controlador corresponent. Aquest rep la petició, la processa a la capa de negoci i persistència i retorna la resposta al *DispatcherServlet* perquè pugui retornar la nova pàgina a mostrar a l'usuari. *Spring MVC* també permet utilitzar controladors que generin codi *HTML* per ser incrustat a la pàgina directament. No obstant, aquesta possibilitat de retornar codi *HTML* s'ha d'utilitzar amb cura, per evitar que part del disseny de les pàgines que es mostren estigués codificat en la classe *Java*, la qual cosa dificultaria la separació de rols de desenvolupament i ens allunyaria del paradigma que pretén seguir el patró *MVC*.

Spring està molt ben documentat i tot i això no és fàcil iniciar-se amb ell degut a la gran quantitat de configuracions necessària per què funcioni. A part del imprescindible arxiu *web.xml*, s'han de configurar d'altres per definir els contexts per cada un dels controladors que utilitzem (a més del *DispatcherServlet)* però a més, els contexts d'aplicació (que és una característica del contenidor d'aplicacions, i no pas del *framework*), poden ser heretats i redissenyats segons convingui.

Durant el cicle de vida de l'aplicació, el desenvolupador haurà de ajustar el comportament dels controladors utilitzant una sèrie d'etiquetes definides o anotacions definides per el *framework*. Les anotacions ens permeten per exemple, validar l'entrada de l'usuari amb facilitat, mapejar els controls del formulari a variables del model, i parametritzar els manejadors de cada controlador. Aquest fet, ens obliga a l'aprenentatge i comprensió de funcionament d'un nou lèxic basat en etiquetes i anotacions la qual cosa requereix un esforç addicional.

El següent diagrama extret de la web oficial, representa el flux de funcionament del *framework Spring*:



Els avantatges que es destaca la mateixa pàgina son:

Flexibilitat a l'hora de mapejar els manejadors o *handlers*. De fet, cadascun dels participants (controladors, validadors, handlers, interceptors) poder ser objectes *Java* normals, sense necessitat d'heredar d'una classe abstracta o *interface* com passa amb *Struts* i *Struts2.*

Potent i senzilla configuració del marc de treball de les classes com JavaBeans.

Adaptable i no intrusiva creació dels controladors, que poden derivar de diferents classes en funció del rol que han de tenir a l'aplicació.

Codi de negoci reutilitzable.

Vincles i validació personalitzables. En quant a la validació, *Spring MVC* proporciona el seu propi mòdul de validació que pot ser recodificat per ajustar-se a les necessitats de l'aplicació.

Manejadors i resolució de vistes personalitzables.

Ús d'interceptors que permeten personalitzar les accions a executar abans i després de processar la petició utilitzant un mètode per cada situació, que funcionen com a resposta a cada un dels events que es produeixen durant el processament de la petició.

Senzilla implementació de la internacionalització (utilització de missatges en diferents idiomes) gràcies a les etiquetes *JSP*.

Temes personalitzats per assignar un determinat estil gràfic a les pàgines.

Autenticació de credencials d'usuari a partir de configuració *XML*, que permet especificar inclús la comanda SQL que s'executarà en el Model per validar l'accés, sense implementar pràcticament cap línia de codi *Java*.

Extensió de les etiquetes estàndard que ofereix *JSP*.

Diferents tecnologies de renderització aplicables a les vistes: Velocity, JSF, JSP... sense necessitat d'utilitzar *plugins* addicionals.

En resum, *Spring MVC* ofereix una potent eina per el desenvolupament d'aplicacions *J2EE*, però la seva flexibilitat i configurabilitat el fan poc accessible als desenvolupadors que no estiguin familiaritzats amb ell, i la seva productivitat en aquest sentit es veu perjudicada. Tot i això, un cop es dominen els nous conceptes, fitxers de configuració i sintaxi, es tracta d'un producte recomanable sobretot per què és l'únic que permet treballar per totes les fases del desenvolupament, incloent la capa de negoci i persistència, i per tant, és l'únic que permet utilitzar un únic *framework* per tota l'aplicació. A més, respecte a l´ús de *Spring MVC*, resulta interessant per la seva capacitat de modificar el comportament estàndard dels elements que el conformen.

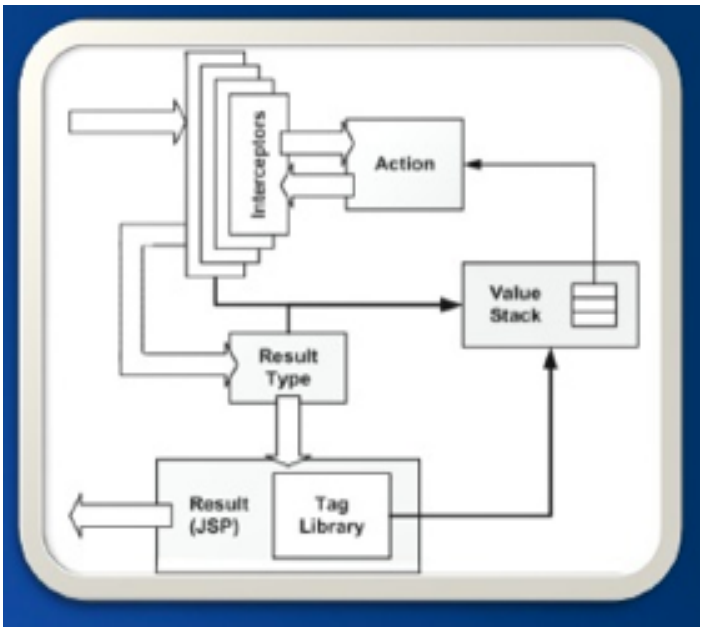
- Struts

Va ser dissenyat dintre del projecte *Jakarta d'Apache Foundation*, tot i que posteriorment va constituir-se com un projecte propi. Degut a que de fet, va ser un dels primers *frameworks* existents, avui en dia es considera que no té les prestacions de que disposen altres alternatives disponibles i no es recomana el seu ús. Aquest factor va fer que el projecte quedés paralitzat al desembre de 2008 en favor de *Webwork/Struts2*.

- Struts2

També basat en codi obert, *Struts2* representa un re-disseny a partir de dos projectes previs: *Struts* i *WebWork*. De fet, tothom coincideix a afirmar que *Struts2* es basa principalment en aquest últim, tot i que per motius de màrqueting es va escollir el nom actual. L'experiència assolida amb *Struts* va permetre crear un nou *framework* més actual i que complia amb els requeriments del moment. La seva principal característica és la necessitat de definir accions que actuen en el paper de controlador. La seva corba d'aprenentatge es força més baixa que en el cas de *Spring MVC*, la qual cosa el fa convertir-se en un producte més assequible per al principiant.

El següent diagrama resumeix el flux de funcionament de *Struts2*:



Struts2

Podem distingir una sèrie d'elements diferenciats que participen en el funcionament del marc de treball: Les accions (Controladors), els resultats (Vistes) i el 'Value Stack' que s'encarrega de mantenir la integritat amb el Model (la capa de negoci i persistència). L'ús d'interceptors és també un dels punts clau del seu funcionament, ja que permeten inserir codi *java* per ser executat abans i després de la invocació de l'acció corresponent. L'eix central d'*Struts2* és la

utilització d'objectes simples de Java per especificar les diferents accions que el *framework* executarà en resposta de les peticions provinents de la pàgina web. Aquestes accions, i la corresponent pagina a mostrar en cada cas han de quedar reflectides al fitxer de configuració *struts.xml.*

Struts2 inclou tot un seguit de característiques que el fan molt popular a l'hora de ser escollit com a opció preferida per desenvolupar el model *MVC* a la capa de presentació:

Ús d'accions en la forma d'objectes *Java* simples, com a implementació d'una *interface* o com a subclasse de la classe *ActionSupport* que a la seva vegada inclou diverses implementacions de *interface*.

El *framework* ofereix seguretat entre *threads* ja que cada crida al controlador provoca una instanciació de la classe, per la qual cosa, la memòria no és compartida per diferents instàncies o peticions.

Ús d'interceptors que permeten executar codi abans i després de processar la petició a través d'un sol mètode. (Al contrari que passava amb *Spring MVC*, el programador necessita implementar la invocació de la petició manualment, considerant cadascun dels moments en els que vol intervenir).

Ofereix seguretat de credencials d'usuari a través d'un plugin adicional.

Suport per *ONGL*, un potent llenguatge d'expressions que permet referenciar les propietats dels objectes *Java*.

Ús d'anotacions que simplifiquen la configuració a través de *XML*.

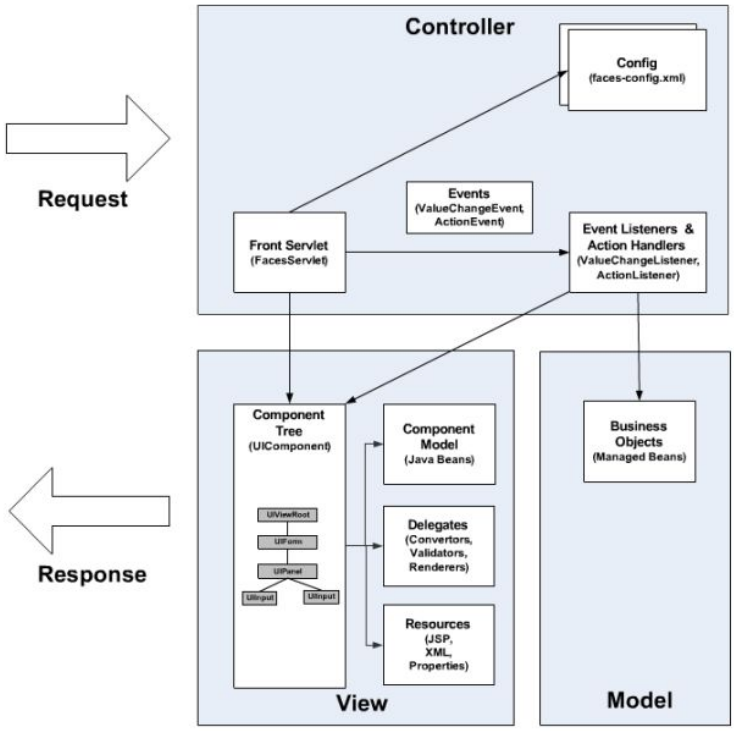
Struts2 és sovint utilitzat en combinació a *Spring* (reemplaçant l'ús de *Spring MVC*) per el desenvolupament d'aplicacions web, degut a que resulta més fàcil de implementar. Tot i això, la documentació no és tan extensa com en el cas de *Spring*.

- JSF (Java Servlet Faces)

Principalment orientat a facilitar el disseny de les vistes, *JSF* centra la seva atenció en el disseny i control dels components gràfics de les pàgines i control d'aquestes. Forma part, de fet, de l'especificació *J2EE* la qual cosa garanteix el seu manteniment i estabilitat. *JSF* està orientat a events, i defineix dos tipus de *Beans* diferenciats: Els *beans* de control que faran el paper de Controlador, i els *beans* administrats que permetran accedir al Model. El funcionament d'aquest *framework* és el següent: La petició és rebuda per el controlador central, que en aquest cas s'anomena *FacesServlet*, configurat amb l'ajuda d'un arxiu *XML* (*faces-config.xml*). El controlador escolta una sèrie d'events i executa alguna de les accions associades, que a la vegada en la seva execució treballen amb el Model, a través dels beans administrats (*ManagedBeans*), que també es configuren al mateix arxiu *faces-config.xml*. Novament a través d'events s'accedeix a la capa de les Vistes, que estan dividides en dos parts principalment: La vista arrel, que

inclou referències als controls gràfics i mantenen control sobre el seu estat en cada moment a través de l'element *UIComponent*, i les diferents vistes, dissenyades en *JSP*, *HTML*, etc... El primer cop que es mostra la pàgina en pantalla, JSF genera a la vista arrel la relació de tots el components gràfics existents i en preserva l'estat.

El model *MVC* que implementa el *framework* és una versió evolucionada del model clàssic que permet integrar en certa manera, la capa de Vistes amb el Model. El propi *framework* incentiva aquesta pràctica que tot i que hi ha veus que asseguren que s'allunya del disseny estricte MVC, altres en canvi ofereixen punts de vista més positius, assegurant que es tracta d'un model perfeccionat.



JSF

- Seam

Framework dissenyat per la comunitat *JBoss*. En la seva estructura, *Seam* integra la capa de vistes amb la capa del Model, i potser sigui aquest l'aspecte més interessant i diferenciador d'aquest marc de treball. Tot i això, el rendiment no resulta especialment beneficiat per la qual cosa, les aplicacions desenvolupades amb aquest marc de treball solen llançar-se (paradoxalment) en servidors de *servlets* més lleugers, com *Tomcat*. El seu funcionament està basat en *JSF* (tecnologia de servidor) i aconsegueix evitar la necessitat d'utilitzar un controlador per la qual cosa no podem dir que segueixi estrictament el paradigma de *framework MVC*.

- Tapestry

Projecte desenvolupat per *Apache* que és molt lleuger i per tant ofereix un bon rendiment. Es tracta d'un *framework* basat en components i orientat a events. Aconsegueix eliminar pràcticament la configuració *XML*, el que el fa accessible i fàcil d'utilitzar en general, estalviant al programador la necessitat de comprendre la sintaxi dels fitxers de configuració i el seu comportament. Com veurem també amb el següent *framework* de la llista, *Tapestry* utilitza *namespaces* per definir una nomenclatura *XHTML* estàndard que permet editar els fitxers des de un programa de disseny. Les pàgines utiltzen aquestes etiquetes especials per enllaçar-se amb objectes *Java* simples el que facilita un accés a les dades dels controls natural i intuïtiva.

- Wicket

Es tracta de un *framework* també desenvolupat per *Apache* i basat en components que fa de la seva simplicitat, el seu punt fort. De fet no es pot considerar un entorn que implementi el paradigma *MVC*, sinó que com succeeix amb *JSF* o *Tapestry*, està molt orientat a establir un vincle natural i fàcil de desenvolupar entre pàgines *HTML* i objectes java. Defineix un *namespace* especial per incloure les etiquetes necessàries dins el codi *HTML* però al contrari que succeeix amb les pàgines *JSP*, *Wicket* no hi permet incrustar codi *java* sinó que al definir etiquetes que conformen l'estàndard *XHTML*, les pàgines poden ser editades sense problemes per un dissenyador des d'un programa editor de pàgines web professional. El *servlet* principal es diu *WicketFilter*, però apart d'aquest punt de partida, que és necessari definir a l'arxiu *web.xml*, la resta del funcionament intern queda totalment transparent al desenvolupador. Aquest *framework* és apreciat per la seva productivitat i està especialment orientat a aquells programadors experts en *java* que no desitgin haver de configurar els diferents aspectes que altres *frameworks* com *Struts2* o *Spring* permeten. Per tant, aspectes com la validació, els interceptadors, els controladors, han de ser implementats purament en *java*.

5. Anàlisi comparatiu

Un cop definits els principals marcs de treball existents, el fet d'escollir un o altre *framework* a l'hora d'iniciar una aplicació pot esdevenir una decisió complicada ja que tots ells persegueixen un mateix objectiu i sovint no competeixen entre ells, sinó que es complementen. Per exemple, tant *Spring* com *Struts2* disposen de mòduls que permeten utilitzar capacitats de l'altre *framework* i durant el desenvolupament, les vistes poden estar implementades en *JSF*. A més, en els

darrers anys la majoria dels *frameworks* descrits anteriorment han evolucionat de forma destacada fent que cada cop les diferències entre ells siguin menors.

L'aspecte més important i diferenciador a destacar és la manera com cadascun dels *frameworks* implementen el patró *MVC* i en particular com gestionen les peticions que arriben al *servlet* principal a través dels manejadors. Després hi ha característiques que s'han considerat molt interessants per el desenvolupament d'aplicacions de J2EE i que cadascú implementa de forma particular, com és la gestió de interceptors, la injecció de dependències, la resolució de les vistes, la internacionalització, la validació de formularis o l'autenticació de credencials. A continuació estudiarem cada apartat per separat, fixant-nos en els avantatges i desavantatges de cada *framework* subjecte d'estudi.

- Manejadors (Handlers)

Hi ha diferents tipus de manejadors en els entorns *MVC* . Els interceptors son un tipus particular de manejadors i els estudiarem en un següent apartat. En aquest punt, principalment ens interessen els manejadors que relacionen les peticions provinents del client, amb el controlador (i mètodes) que han d'executar. Ens fixarem en especial en quin sistema fan servir els diferents *frameworks* per cridar els mètodes dels controladors en cada cas.

Spring

Spring inclou diferents tipus de manejadors que son configurables per la via del fitxer *web.xml* o amb l'ús d'anotacions. El sistema de mapeig de manejadors amb spring es basa en l'ús d'anotacions, la qual cosa proporciona al desenvolupador una gran flexibilitat per crear els controladors que gestionaran la lògica del negoci. En primer lloc amb l'anotació '@controller' identifica el controlador, que com ja s'ha comentat, és una classe java simple, sense necessitat d'herència ni abstracció. A Spring existeixen dos tipus de controladors: Els 'CommandController' i els 'FormController'. Els primers son el més semblant a les accions d'Struts, (els controladors, executen una acció determinada i retornen una vista com a resultat). mentre que els segons, permeten enllaçar còmodament les dades que conté un formulari en una classe java, realitzar tècniques de validació i persistència de dades. Tant si és a nivell de classe, com si és a nivell de mètode, l'anotació '@RequestMapping' permet establir criteris de coincidència amb el camí que arriba al controlador , el mètode *http* utilitzat (GET, POST ...) o fins i tot l'estat de variables de sessió per determinar quins mètodes s'executaran quan arribi una determinada petició. Un exemple de configuració de manejadors utilitzant anotacions amb Spring seria:

```
@Controller
public class MyController {

      @RequestMapping(value = "/hola",method = RequestMethod.GET)

      public String hello(HttpServletRequest req,
                  HttpServletResponse answ {
```

```
            return “hello”;
        }
}
```

Aquest es un cas de controlador molt simple, que respon a la petició ‘/hola’ amb el protocol ‘GET’ per retornar el nom de la vista ‘hello’. Més endavant , la resolució de vistes haurà de transformar aquest nom en un arxiu vàlid per ser renderitzat.

Struts2

Struts2 basa l’ús de controladors en les denominades ‘actions’ o objectes simples de *java* que s’executen en resposta de determinades peticions. La configuració d’aquestes accions es realitza al fitxer *struts.xml* que és on quedarà reflectit quina acció s’executa en resposta a quina petició. Un exemple de configuració seria el següent:

```
<action name="Logon" class="tutorial.Logon">
   <result type="redirectAction">Menu</result>
   <result name="input">/Logon.jsp</result>
</action>
```

En aquest petit exemple l’acció denominada ‘Logon’ s’executa a la classe ‘tutorial.Logon’ i en funció del valor retornar per aquesta, passarà al següent pas (redirecció a ‘Menu’ o mostrar la vista ‘Logon.jsp’. Això significa que Struts necessita indicar tant la classe a executar com les vistes derivades del processament de la classe.

JSF

En aquest cas, no s’utilitza un sistema de mapeig similar als que hem vist anteriorment, sinó que el *framework* permet cridar directament des de la pàgina .jsf al controlador i mètode que necessitem en cada cas. Per tant la crida es realitza directament des de la pròpia pàgina. En alguns dels casos que veurem a continuació (*Tapestry, Wicket*) , l’aproximació és força similar.

Seam

En *Seam*, els controladors de l’aplicació hereten d’una classe abstracta ‘*Controller*’. El *framework* incorpora dues subclasses, ‘*BusinessProcessController*’ i ‘*PersistenceController*’ per implementar controladors de negoci o persistència respectivament. Els controladors s’accedeixen a través de la pàgina (jsf) mitjançant un expressio EL amb la denotació ‘#classe.metode’. Així , un exemple podria ser:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html> <head> <title>Login</title> </head>
<body> <h1>Login</h1>
<f:view>
    <h:form>
      <div>
```

```
          <h:inputText value="#{login.user}"/>
          <h:commandButton value="Login" action="#{login.login}"/>
       </div>
      </h:form>
</f:view>
</body>
</html>
```

Tapestry

Amb tapestry, la implementació dels manejadors és radicalment diferent. En aquest cas, s'anomenen 'component events' i estan implementats a la pròpia pàgina JSP, que estableix un conjunt extés d'etiquetes, per associar els enllaços amb l'acció a executar en cada cas. Per exemple (extret de la pàgina oficial de Tapestry http://tapestry.apache.org/component-events.html:

```
        <p> Escull un numero del 1 al 10:
      <t:count end="10" value="index">
         <a t:id="select" t:type="actionlink"
                                    context="index">${index}</t:comp>
       </p>
```

Això permet cridar al servidor com a:

http://localhost:8080/chooser.select/3

éssent 'chooser' el nom del template (chooser.tml) , 'select' el identificador del control, i '3' el paràmetre que volem passar. En codi java, la classe que haurà de rebre la petició s'indentificarà amb l'anotació '@OnEvent(component="select") i el framework l'associarà amb el control de la vista per executar el negoci corresponent. Per tant en aquest cas no trobem una relació de requests => controladors/accions si no que la referència a l'execució de l'acció s'implementa directament a la pàgina de la vista.

Wicket

En aquest cas, la crida al controlador es fa mitjançant un event associat al control desde la pàgina html. El controlador, que està implementat en la classe associada amb el model, inclou un mètode que rep el event indicat. Per exemple, el següent codi:

```
      parent.add(new Button("neteja") {
            @Override
            public void onSubmit() {
                BusquedaCelularesPage.this.buscador.clear();
            }
        });
```

Aquesta seria la implementació de l'event *onSubmit* que executaria el negoci. El control cridarà aquest event en quan correspongui (p.e. event onclick en un butó).

En els casos en que la pàgina *html* conté la referència al controlador que s'ha d'executar, el *framework* no incorpora un sistema de mapeig de manejadors, ja que la crida es produeix des de la pròpia pàgina.

- Gestió de les vistes

La Gestió de vistes recull la capacitat de trobar els fitxer que es renderitzaran en el navegador web en funció del camí proporcionat i també s'ocupa de la pròpia renderització que permetrà visualitzar la pàgina de forma dinàmica, amb els atributs que hagi establert el desenvolupador prèviament.

Spring

Per la gestió de les vistes, Spring disposa per una banda d'una classe *InternalViewResolver* que resol els noms de les vistes i obté el camí del fitxer que ha de renderitzar, i després, de diverses classes que s'encarreguen de renderitzar i mostrar els continguts dels fitxers sel·leccionats en funció de la tecnologia utilitzada en cadascun d'ells. La correspondència entre els fitxers a renderitzar i la classe de renderització associada es realitza seguint la configuració *xml* en el fitxer de contexte corresponent. Spring permet processar vistes de una àmplia varietat de tecnologies: Jsp, Html, Velocity/ FreeMaker, Jsf , Xml, Pdf, Excel ... Es poden configurar més de una classe de renderització a la vegada, per cada tipus de fitxer pugui ser processat adequadament.

Struts2

Struts2 implementa un sistema de crida a accions que son configurats en el fitxer *struts.xml*. Aquí es representen les accions tal com s'havia explicat en el punt anterior i per tant, el *framework* sap on trobar la classe associada amb l'acció quan aquesta es crida des de la pàgina web.

JSF

Seam

Tapestry

Wicket

Aquests *frameworks* centren la seva gestió de vistes en les pàgines *JSF*. Per tant, integren la renderització de vistes de forma transparent al desenvolupador i no necessiten de una configuració addicional.

- Interceptors

Els interceptors proveeixen de mètodes per executar abans o després de la crida a les peticions.

Spring

A Spring, Aquests interceptors han de ser configurats al fitxer de configuració de contexte de Spring. Un dels objectius de la pràctica serà eliminar aquesta necessitat. Un exemple de configuració actual és com segueix:

```
<!-- configura Interceptors -->
<mvc:interceptors>
<!-- Aquesta configuració intercepta totes les URL's -->

<bean class="marin.interceptor.RequestInitializeInterceptor" />

</mvc:interceptors>
```

Les classes que tenen la funció d'interceptors implementen la interface *HandlerInterceptor* inclouen tres mètodes, un per cada situació:

. prehandle. Per ser executat abans de la petició

. posthandle. Per ser executat després de la petició

. afterCompletion. Per ser executat un cop la petició hagi finalitzat.

Struts2

En aquest cas, els interceptors s'utilitzen per ser executats abans o després d'una determinada acció. La classe ha d'implementar una interface *Interceptor* i a través d'ella disposa de tres mètodes:

. init Inicialització de l'interceptor

. destroy. Destrucció de l'interceptor

. intercept. Mètode que intercepta la crida.

El procés que segueix Struts2 amb els interceptors és el següent:

1. En primer lloc, es genera la petició per el usuari i s'envia al contenidor d'aplicacions.

2. Seguidament, el *FilterDispatcher* de Struts2 esbrina quina acció correspon a la petició demanada.

3. A continuació s'executen els interceptors abans de cridar a l'acció.

4. Després s'executa l'acció i es recull el resultat.

5. Finalment, es renderitza la vista obtinguda.

Val a dir que una de les diferències principals amb Spring és que en aquest cas, l'acció s'executa dins el codi de l'interceptor utilitzant l'objecte *ActionInvocation*,

per la qual cosa, només és necessari un mètode per inserta codi abans o després de l'execució.

Els interceptors s'han de configurar al fitxer struts.xml abans de ser utilitzats. Per exemple:

```
<interceptors>
    <interceptor name="mylogging"
        class="net.viralpatel.struts2.interceptor.MyLoggingIntercepto
r">
    </interceptor>
    <interceptor-stack name="loggingStack">
        <interceptor-ref name="mylogging" />
        <interceptor-ref name="defaultStack" />
    </interceptor-stack>
</interceptors>
```

JSF

Amb JSF, l'ús d'interceptors es pot implementar amb el *PhaseListener*. Aquesta interfície permet identificar el tipus de interceptor i després implementar manejadors a events que s'executaran abans o després d'aquesta acció determinada. En aquest cas, els interceptors no es limiten a la crida de peticions web, si no que poden ser implementats en una gama més àmplia de contextes, per exemple, en el moment de renderitzar una pàgina. Aquest seria un exemple d'implementació:

```
public class RequestInterceptor implements PhaseListener {
   @Override     public PhaseId getPhaseId() {
         return PhaseId.RENDER_RESPONSE;

     }
   @Override     public void beforePhase(PhaseEvent event) {
         // Codi que s'executa abans del render.
     }
   @Override     public void afterPhase(PhaseEvent event) {
         // Codi que s'executa despres del render.
     }
  }
```

En aquest cas, l'interceptor respon a un event RENDER_RESPONSE, però es podrien utilitzar altres, com per exemple ANY_PHASE, per controlar qualsevol moment de l'execució.

Seam

Amb *Seam*, els interceptors s'utilitzen amb una anotació especial @AroundInvoke que identifica l'interceptor. Per ser utilitzat, caldrà crear una anotació nova que referenciarà la classe i utilitzar-la en el mètode que volguem controlar. Val a dir que com en el cas de JSF o els que venen a continuació, Tapestry o Wicket, que son frameworks orientats a events, el desenvolupador disposa dels events com a eines per controlar l'execució de determinats events del sistema. En aquest cas, els interceptors s'utilitzen més aviat en el cas que

necessitem crear situacions noves a controlar (Per exemple, esbrinar si un l'usuari ha entrat a la sessió abans de processar una determinada pàgina).

Tapestry

Wicket

Aquests *frameworks* no suporten interceptors en el sentit que els estem tractant.

- Injecció de dependències

La injecció de dependències es una característica comuna a tots els *frameworks* MVC que permet desacoblar la dependència entre classes, seguint el principi que si una classe necessita d'una altra, li hauria d'arribar des de fora, mitjançant el constructor o un mètode '*setter*'.

Spring

La injecció de dependències és una de les principals característiques de *Spring*. Aquestes poden ser configurades des del fitxer XML de configuració de contexte o be utilitzant l'anotació '@AutoWired'. La injecció de dependències, es pot realitzar a nivell de constructor, o a nivell de mètode setter. L'objectiu és aquestes classes que siguin dependència de la que volem implementar, arribin des de fora i d'aquesta manera desacoblar les dependències entre classes. Per exemple:

```
package testbean;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import writer.IWriter;
  @Service public class MySpringBeanWithDependency {
      private IWriter writer;

      @Autowired
      public void setWriter(IWriter writer) {
          this.writer = writer;
      }
      public void run() {
         String s = "Aixo es una prova";
         writer.writer(s);
      }
   }
```

La implementació de la interface *IWriter* seria con segueix:

```
import org.springframework.stereotype.Service;
@Service public class NiceWriter implements IWriter {
   public void writer(String s) {
     System.out.println("la cadena es " + s);
   }
 }
```

En aquest cas, el *framework* buscarà entre els seus beans aquell que implementa la interfície *IWriter* i la injectarà al *setter*. Tot i que Spring recomana l'ús d'anotacions per les noves aplicacions encara manté la possibilitat d'utilitzar el fitxer de configuració XML:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
2.5.xsd">
<bean id="writer" class="writer.NiceWriter" />
      <bean id="mySpringBeanWithDependency"
                          class="testbean.MySpringBeanWithDependency">
      <property name="writer" ref="writer" />
   </bean>
</beans>
```

Struts2

Struts2 inclou un sistema d'injecció de dependències natiu, i també permet realitzar-ho des de un *plugin* per *Spring*. El sistema natiu també incorpora anotacions, en aquest cas l'anotació '@Inject' com es pot veure en l'exemple següent:

```
@Inject("man")
private ProductManager manager;
// Aquest mètode s'utilitza per injectar l'objecte
public void setManager(ProductManager manager) {
    this.manager = manager;
}
```

En aquest cas, també serà necessari configurar el bean al fitxer *struts.xml*

```
<!DOCTYPE struts PUBLIC "- //Apache Software Foundation //DTD Struts
Configuration 2.1//EN""http://struts.apache.org/dtds/struts-2.1.dtd">
<struts>
   <bean class="com.mahendra.ProductManager name="man" />
</struts>
```

JSF

La injecció de dependències a *JSF* es realitza a través del fitxer de configuració XML *faces-config.xml* de la següent manera:

```
<faces-config>
   ..
    <managed-bean>
        <managed-bean-name>task</managed-bean-name>
        <managed-bean-class>sernet.verinice.web.TaskBean</managed-
bean-class>
```

```
        <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>editBean</property-name>
            <property-class>sernet.verinice.web.EditBean</property-
class>
            <value>#{edit}</value>
        </managed-property>
    </managed-bean>
    <managed-bean>
        <managed-bean-name>edit</managed-bean-name>
        <managed-bean-class>sernet.verinice.web.EditBean</managed-
bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
   ..
</faces-config>
```

En aquest cas, l'objectiu es injectar el *bean* controlat 'edit' en el *bean* 'task' . Per que això sigui efectiu, la classe 'task' ha de incloure el getter i setters corresponents, i d'aquesta manera, el framework instanciarà les dues classes 'edit' i 'task' i serà capaç d'injectar la dependència 'edit' a la classe 'task' adequadament:

```
public class TaskBean {
    private EditBean editBean;
    public EditBean getEditBean() {
        return editBean;
    }

    public void setEditBean(EditBean editBean) {
        this.editBean = editBean;
    }
}
```

Seam

Seam disposa d'injecció de dependències en tots dos sentits. Aquesta característica s'anomenta '*Bijection*' i s'utilitza amb dos anotacions diferenciades '@Out' i '@In'. Per una banda, el desenvolupador ha d'anotar amb '@Out' l'objecte que ha de actualitzar els components gestionats des de l'acció. En canvi ha d'anotar amb '@In' l'objecte de la classe destinació que rebrà la dependència provinent dels components d'usuari provinents en alguns casos d'un formulari implementat a la pàgina *JSF*. S'ha de indicar que amb *Seam* , la injecció de dependències es produeix cada vegada que s'invoca el mètode, al contrari del que passa per exemple amb *Spring*, on es produix un cop quan s'instància la

classe. Això vol dir, per exemple, que a *Seam*, la classe que rebi la dependència ha de ser configurada dins d'un *scope* el més restringit possible, per evitar problemes de concurrència. Un exemple d'implementació seria el següent:

```
@Name("userDao")
@AutoCreate
@Scope(ScopeType.APPLICATION)
public class UserDaoBean implements UserDao {

    @In
    private EntityManager entityManager;

    public User findByEmail(String email) {
        return (User)
entityManager.createNamedQuery("findUserByEmail").setParameter("email
", email).getSingleResult();
    }
}
```

En aquest exemple, la classe *UserDaoBean* rep la dependència de *EntityManager*. En aquest cas, no es necessita crear un setter específicament. En canvi, el problema ve quan Seam finalitza la invocació i reinicialitza el valor de *entityManager* a null, la qual cosa fa que un altre *thread* que es pugui estar executant de forma concurrent, trobi que el valor ha canviat inesperadament.

Tapestry

La injecció de dependències amb *Tapestry* s'implementa utilitzant el que s'anomena '*Tapestry IoC's Registry*' (Registre d'inversió de control de *Tapestry*), que és el contenidor que instància les classes. Val a dir que la injecció de dependències en aquest cas es realitza a nivell de constructor. *Tapestry* sap trobar el constructor de la classe i analitza els paràmetres per esbrinar en quins d'ells ha d'efectuar la injecció. Aquest *framework* no necessita de cap configuració addicional XML per efectuar el procés. Aquest tros de codi implementa la injecció de dependències amb Tapestry. Com es pot veure, s'utilitza el mètode bind per informar el Registre d'inversió de control de quins objectes ha de considerar injectables.

```
public class MonitorModule {
   public static void bind(ServiceBinder binder)   {
      binder.bind(QueueWriter.class, QueueWriterImpl.class);
      binder.bind(MetricScheduler.class, MetricSchedulerImpl.class);
   }
   public void
            contributeMetricScheduler(Configuration<MetricProducer>
                 configuration, QueueWriter queueWriter, . . .)   {
      configuration.add(new TableMetricProducer(queueWriter, . . .))
   }
}
```

Wicket

Wicket utilitza la injecció de dependències a través de *setters* i *getters* amb l'ajuda de l'anotació '@Inject'. El següent és un exemple d'injectació de dependències amb *Wicket*:

```
@ApplicationScoped
public class Clock {
   public String getTime() {
      return Time.now().toString("HH:mm:ss");
 }
}

public class ClockModel extends AbstractReadOnlyModel {
   @Inject Clock clock;

   @Override public String getObject() {
       return clock.getTime();
   }
}
```

El *framework* no necessita que s'especifiqui un setter sinó que en crea un per defecte automàticament. Com es pot veure, al contrari del que passava amb *Seam*, la injecció de dependències amb *Wicket* si que és segura entre *threads* per la qual cosa el *scope* de la classe pot ser establert a nivell d'aplicació. Wicket no necessita de cap configuració addicional.

- Internacionalització

La internacionalització és la característica que permet desenvolupar aplicacions web amb suport per diferents idiomes, sense necessitat de crear una vista diferent per cadascun.

Spring

A través d'una configuració al arxiu de contexte XML, *Spring* permet definir un '*localeResolver*' per realitzar la resolució dels missatges. De fet, no hi ha una única manera d'implementar la internacionalització amb Spring, però utilitzar el mètode del '*localeResolver*' permet assignar, per exemple un '*CookieLocaleResolver*' que ens mantindrà automàticament la configuració del idioma actual en *cookies*. En qualsevol cas, haurem d'especificar també un fitxer de recursos a través d'un *bean* '*ReloadableResourceBundleMessageSource*' que especifica el directori i nom base sota el que es trobaran les diferents traduccions dels textos que volguem utilitzar. Aquest fitxers (un per cada idioma), permeten definir els missatges de la següent manera:

profile.F_CREATEPROFILE=Crear perfil de un usuario

Per utilitzar-ho des de la pàgina necessitarem crear el missatge amb l'identificador (part esquerra de l'igual) amb un *namespace* propi de *Spring*, de la seguent manera:

<spring:message code="profile.F_CREATEPROFILE" var='L_PROFILE'/>

Per accedir al valor de la variable establerta, podrem fer-ho amb ${L_PROFILE}.

Aquest sistema requereix de cert grau de configuració en un fitxer de contexte XML i a més, l'ús de *namespaces* propis dins la vista.

Struts2

La implementació de la internacionalització des de *Struts2*, s'efectua a través d'un interceptor especial 'i18n' afegit a la pila de interceptors d'*Struts2* i configurat a l'arxiu *struts.xml*:

```
<interceptor-ref name="i18n">
   <param name="parameterName">lang</param>
   <param name="attributeName">MY_LANG</param>
</interceptor-ref>
```

El *framework* també utilitza fitxers de recursos d'idiomes, amb la mateixa sintàxi que *Spring*, però en aquest cas, la seva localització es configura en un altre arxiu de propietats, struts.properties on s'indicaran els noms dels fitxers que s'utilitzaran per les traduccions.

L'accés al text traduït es fa a través d'una etiqueta especial d'Struts2 a la vista:

```
<s:textfield key="profile.F_CREATEPROFILE"  size="25"
                                            maxLength="64"/>
```

JSF

Per implementar la internacionalització amb *JSF*, primer s'ha de carregar l'arxiu de recursos a la vista, utilitzant l'etiqueta '*loadBundle*' on s'especifica el nom base dels fitxers d'idiomes (aquell sense l'extensió corresponent a l'idioma) i la variable que s'utilitzarà per traduir els textos. Una alternativa que ofereix *JSF* és a través de la configuració del seu arxiu *faces-config.xml* on caldria referenciar l'arxiu de recursos igualment, amb una configuració com la següent:

```
<application>
<message-bundle>com.sun.bookstore6.resources.ApplicationMessages
                                                    </message-bundle>
     <locale-config>
         <default-locale>en</default-locale>
         <supported-locale>es</supported-locale>
         <supported-locale>de</supported-locale>
         <supported-locale>fr</supported-locale>
     </locale-config>
     <resource-bundle>
        <base-name>com.example.faces.i18n.Text</base-name>
        <var>msg</var>
```

```
    <resource-bundle>
</application>
```

La referència al text traduït es farà segons la variable especificada, en aquest cas 'msg' amb el format que s'ha explicat en els casos anteriors.

Seam

Seam utilitza bàsicament el mateix mètode de internacionalització heretat de *JSF*.

Tapestry

Per activar el suport multi-idioma a tapestry, en primer lloc cal informar al framework de quins idiomes tenim disponibles. Això ho podem fer utilitzant el mètode 'contributeApplicationDefaults' de la classe 'AppModule'. La sintaxis és la següent:

```
public static void contributeApplicationDefaults(
                                    MappedConfiguration configuration){

      configuration.add("tapestry.supported-locales", "en,fr,de");
   }
```

A continuació és necessari crear els arxius de propietats corresponents a cada idioma. Els fitxers de propietats poden existir per cada pàgina template creada i si tenen el mateix nom base que aquesta, el framework les utilitzarà directament en cada control requerit dintre de la vista. Per crear la referència al text que serà traduït, s'utilitzarà l'expressió:

`$(message:key.in.properties.file}`, per exemple, si al fitxer de propietats tenim:

```
confirm.delete.key=You are about to delete entity '%s'. Are you sure
?
```

La referència a la pàgina haurà de ser:

```
${format:confirm.delete.key=deleteEntityName}
```

D'aquesta manera, *Tapestry* actualitzarà el valor del missatge segons el locale definit a la sessió de l'aplicació en cada moment.

Wicket

Wicket segueix una aproximació similar del problema. En primer lloc, els fitxers de propietats al igual que passa amb *Tapestry*, estan orientats als components o la vista i per tant es poden crear fitxers de propietats a nivell d'aplicació, de pàgina, o de component. Per accedir al text traduït, *Wicket* proveeix un tag especial, *wicket:message* amb un a propietat *key* que identifica el text dins el fitxer de propietats. Així, per exemple:

```
<wicket:message key="helloworld">Hello</wicket:message>
```

Busca el text identificat amb el key 'helloworld' dins el fitxer de propietats corresponent, i substitueix el text dins el node del tag per la traducció que correspongui.

La següent taula, mostra un resum de les característiques descrites en l'apartat anterior:

Framework	MVC	Ajax	Seguretat	I18n	Push/pull	Validació	Proves
Spring MVC	Si	Si	Spring security	Si	Push	Commons validator, Bean Validation	Objectes Mock,test unitaris
Struts	Si	Si	No	Si	Push/pull	Si	Tests unitaris
Struts2	Si	Si	No	Si	Push/pull	Si	Tests unitaris
JSF	Si	Si	Si	Si	Pull	Validadors natius Bean Validation	JUnit
Seam	Si	Si	JAAS, Drools, OpenID, CAPTCHA	Si	Pull	Validador hibernate	JUnit, TestNG
Tapestry	Si	Si	Tapestry-security	Si	Pull	Sistema de validació integrat	Selenium, JUnit, TestNG
Wicket	No	Si amb extensions	Si	Si	pull	Si	Objectes Mock, Integració amb extenisions

Resum frameworks MVC

Disseny i implementació

1. Anàlisi i abast

L'objectiu del present projecte és desenvolupar un *framework* per la capa de presentació que compleixi amb les especificacions *J2EE*. En aquest sentit, i després de analitzar les solucions existents en el mercat avui dia, es pot afirmar que resulta difícil proposar un producte millor en rendiment i prestacions donat el grau d'evolució que en aquest moment tenen cadascun dels productes analitzats anteriorment. De tots els marcs de treball existents, en podem destacar dos, que per la seva popularitat han esdevingut molt populars a l'hora de desenvolupar aplicacions web. Tant *Spring MVC* com *Struts2* proporcionen les eines necessàries per satisfer qualsevol necessitat de forma satisfactòria. *Seam*, *JSF*, *Wicket* i *Tapestry* no proporcionen la flexibilitat dels anteriors i ho compensen amb un procés de desenvolupament força més simplificat.

De tots ells el que m'ha cridat més l'atenció és *Spring MVC*, per tractar-se d'un producte complert, extensible i altament flexible i personalitzable. Tot i això, *Spring MVC* té un cert marge de millora ja que el seu punt feble és sens dubte la seva corba d'aprenentatge, i en particular l'accessibilitat a l'hora d'implementar les diferents característiques que ofereix.

És per aquest motiu que la proposta presentada en aquest document serà presentar un nou *framework* de presentació per plataforma *J2EE* prenent com a punt de partida la tecnologia descrita per *Spring MVC*, amb l'objectiu concret de simplificar el màxim el procés de iniciació i funcionament, estalviant en la mesura de lo possible, les configuracions XML que en aquest moment el fan perdre adeptes en favor d'altres marcs de treball més confortables. Concretament, intentarem minimitzar / eliminar els fitxers de context i traslladar part de les configuracions referents a *JavaBeans* a un fitxer de configuració implementat amb una classe java Standard utilitzant noves anotacions que permetin obtenir una codificació molt més compacte. A aquest nou *framework* el batejarem amb l'ocurrent nom de *Winter MVC*.

Partint de la idea de la modularitat proposada per el propi *Spring MVC*, desenvoluparem els mòduls necessaris que ens permetin assolir l'objectiu. Per aquest propòsit, caldrà fixar-se en les diferents funcionalitats del marc de treball, analitzar-les en detall una a una per estudiar les diferents alternatives d'implementació disponibles i proposar un model que les millori en algun aspecte.

Característiques a analitzar:

Internacionalització.

Per treballar amb la internacionalització i localització de les aplicacions, *Spring MVC* utilitza tres classes que han de quedar configurades en elements *bean* d'un arxiu *XML*.

```
org.springframework.context.support.ReloadableResourceBundleMessageSource
```

Permet fer servir arxius de missatge per assignar el valor correcte en els controls del formulari

`org.springframework.web.servlet.i18n.LocaleChangeInterceptor`

Permet canviar el llenguatge de referència

`org.springframework.web.servlet.i18n.CookieLocaleResolver`

Permet guardar en una *cookie* el llenguatge actual.

Aquest sistema ofereix utilitzar el *tag spring:message* per establir els valors de text corresponents segons el idioma utilitzat, i a més canviar el idioma cridant a un enllaç a la pàgina per canviar el valor de la variable establerta (p.e. 'lang'). *Spring* no necessita que sigui creada una classe *java* addicional que canvii el valor de la variable amb *getters* i *setters*. Aquest mètode és prou flexible i fàcil d'implementar i no sembla que sigui viable realitzar cap millora sensible.

Validació de dades de formulari.

Spring proposa dos implementacions diferents segons les necessitats. En primer lloc, hi ha la possibilitat de definir els paramètres d'acceptació de les diferents variables en la pròpia classe del model, i utilitzar el tag *@Valid* per comprovar els valors corresponents en el pròpi controlador, que serà cridat un cop s'hagi acceptat el formulari.

La segona implementació permet personalitzar el comportament del validador, i per aquest objectiu, utilitza una *interface Validador* que el programador pot implementar. Utilitzant els mètodes *support* i *validate*. Els errors de la validació es mostren a la pàgina *JSP* utilitzant el tag *spring:errors*, que a més, permet mostrar missatges d'internacionalització.

En quant a la validació, el *framework* ens deixa pot espai de millora ja que les dos alternatives semblen prou intuïtives.

Implementació de seguretat de credencials.

La seguretat de credencials té un fort component de configuració *XML*, ja que pràcticament permet realitzar l'autenticació sense escriure codi *java*. En contrapartida, la configuració és més difícil d'implementar. El sistema permet autenticar a través de base de dades o parells de noms d'usuari , *passwords* i *roles* en el propi fitxer XML, i especificar les pàgines que es mostraran a continuació en cas d'èxit o error. També permet protegir les vistes segons el rol de l'usuari. La configuració principal recau en el *bean authentication-manager*, que ja bé implementat de forma automàtica en el *framework*.

Tractament d'interceptors.

Els interceptors son una característica comuna a alguns dels *frameworks* estudiats en aquest document, i donen la possibilitat de tractar les peticions abans que aquestes s'executin. Per exemple, la internacionalització fa servir interceptor per canviar el valor de missatge abans que es mostri per pantalla i la

implementació de la seguretat també és una característica tractada amb interceptors *http*. A *Spring MVC*, els interceptors s'implementen a través d'una *interface HandleInterceptor* i una sèrie de manejadors (*handlers*) per executar codi java en diferents estadis de processament de la petició (*preHandle*, *postHandle* i *afterCompletion*). També es permet implementar una classe heredada d'una classe abstracta *HandlerInterceptorAdapter* que proveeix de útils comportaments per defecte. L'ús d'interceptors requereix de la configuració XML apropiada, a través de la propietat *interceptors* a les classes utilitzades, per exemple:

```
org.springframework.web.servlet.handler.SimpleUrlHandlerMapping
o
org.springframework.web.servlet.mvc.support.
                                    ControllerClassNameHandlerMapping
```

Injecció de dependències.

La injecció de dependències permet aïllar les dependències d'una classe el màxim possible. D'aquesta manera, si una classe A depèn d'una segona classe B, es pot proveir de la informació necessària de B a través del constructor de A o utilitzant un mètode *setter*. Aquest sistema augmenta la re-usabilitat de les classes i minimitza la dependència externa i per tant, el manteniment. La injecció de dependències és un dels eixos centrals de *Spring MVC*, a través del *Spring Core Container*, que permet relacionar les classes a través del fitxer de configuració *XML* (*XMLBeanFactory*) o anotacions *(@AutoWired*), que permet a *Spring* buscar en la configuració el *bean* que implementa la *interface* requerida i inserta el *setter* de forma automàtica. Aquest mateix objectiu es pot aconseguir amb l'apropiada configuració *XML* a través de l'ús de propietats amb el nom de la variable de referència. *Spring* permet per tant fer ús de la injecció de dependències a través d'anotacions que resulta més intuïtiu i fàcil d'implementar.

Tractament dels manejadors.

Hi ha diversos tipus de manejadors en Spring MVC. Un d'ells, els interceptors ja han estat analitzats prèviament. Existeixen, també manejadors que responen a determinats events del sistema, com *cStartEventHandler* o *cStopEventHandler* que s'implementen a través de les interfases integrades en el sistema. Aquests manejadors permeten detectar quan un determinat context entra en funcionament o s'atura. Els events també es poder personalitzar, implementant una classe que hereti de *ApplicationEvent*. Tots ells s'han de configurar apropiadament a l'arxiu *XML* usant els *beans* corresponents.

Un altre tipus de manejadors molt utilitzat son els Handler Mappings o mapejadors, que permeten seleccionar els mètodes del controlador a executar en funció de diversos criteris de la petició. Per exemple, si es POST o GET, el nom relatiu de la petició. Existeix una *interface* que implementa el mapeig de la petició (*HandlerMapping*). Aquest tipus de manejadors també poder ser utilitzats amb l'ajuda d'anotacions (@*RequestMapping*)

Resolució de vistes.

Les vistes que actualment estan codificades en *JSP*, permeten utilitzar una tecnologia molt estàndard i versàtil, però en contrapartida, el fet de codificar *Java* dins de la pròpia pàgina dificulta l'edició i el tractament per part d'un dissenyador gràfic. Intentarem utilitzar anotacions amb *namespaces* que segueixin l'estàndard *XHTML* per redissenyar pàgines purament HTML que siguin perfectament compatibles i fàcils d'editar amb qualsevol programa de disseny web.

2. Disseny i implementació

El *framework* que pretenem desenvolupar té l'objectiu principal d'eliminar en la mesura del possible, els fitxers de configuració de context *xml*. La raó és que resulta difícil familiaritzar-se amb els fitxers de configuració *xml*, donada la gran quantitat d'opcions disponibles i la seva particular sintaxi. Restringirem en certa mesura a flexibilitat del *framework*, però en canvi, el farem més accessible a aquells que es vulguin introduir en el desenvolupament d'aplicacions web.

El punt inicial del projecte es el fitxer de configuració. Aquest es tracta d'una classe *java* que el desenvolupador de l'aplicació haurà d'anotar adequadament per donar valors als diferents atributs que utilitzarem en la implementació. Aquesta classe facilitarà la configuració dels diferents elements del *framework* de forma considerable.

Aquest seria un exemple de classe de configuració:

```
public class Contexte
{

    @ContextAttr(group="CONTROLLERS",tag="SCAN")
    static final String controllers ="org.uoc.apptest.mvc;"+
                                     "org.uoc.apptest.main";

    @ContextAttr(group="VIEWS",tag="ANY")
    static final String views_jsp  ="/WEB-INF/views/apptest"

    @ContextAttr(group="VIEWS",tag="JSP")
    static final String views_jsp  ="/WEB-INF/views/apptest/jsp";

    @ContextAttr(group="VIEWS",tag="HTML")
    static final String views_html ="/WEB-INF/views/apptest/html;"+
                                    "/WEB-INF/views/apptest/error-pages";

    @ContextAttr(group="VIEWS",tag="NAMES")
    static final String view_names ="error->error.html;" +
                                     "newprofile->ProfileController:newprofile;" +
                                     "addprofile->ProfileController:addprofile;" +
                                     "principal->MainController"

    @ContextAttr(group="INTERCEPTOR",tag="PRE")
    static final String interceptor_pre ="org.uoc.apptest.Interceptor:preexecute";

    @ContextAttr(group="INTERCEPTOR",tag="POST")
    static final String interceptor_post ="org.uoc.apptest.Interceptor:postexecute";

}
```

Com podem observar, aquesta classe no té cap mètode i en canvi utilitza les constants per assignar valors a cadascun dels atributs. Alguns d'ells, que requereixen algun paràmetre addicional, poden necessitar l'ús d'un separador que s'especificarà en cada cas. D'aquesta manera podrem minimitzar la varietat d'atributs a utilitzar el que facilitarà la utilització.

La localització de la classe de configuració d'especifica de la següent manera:

```
  <!--  la configuracio del framework -->
<context-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>org.uoc.apptest.config.Contexte</param-value>
   </context-param>

<!--  el context listener del framework -->
    <listener>
   <listener-class>org.uoc.winter.listeners.ContextListener</listener-class>
</listener>
```

Això permetrà trobar la classe de configuració i carregar els valors dels atributs quan s'inicii el contexte del servlet. Aquests paràmetres es llegiran i s'assignaran a atributs del tipus 'group@tag = valor' per què puguin ser recollits per el controlador principal en el moment de la inicialització.

Els atributs vàlids son els següents:

- Grup: CONTROLLERS Tag: SCAN

Aquest atribut permet establir els paquets en els que es buscaran els controladors de l'aplicació. Permet establir més d'un valor, separat per ';'.

El següent grup d'atributs ens permet classificar els arxius segons el seu tipus en directoris diferents i establir la localització de cadascun d'ells.

- Grup: VIEWS Tag: ANY

Especifica el directori on hauríem de trobar qualsevol vista, que no coincideixi amb els criteris dels tags següents que son més específics. Podem especificar més d'un directori ,separat per ';'.

- Grup: VIEWS Tag: JSP

Especifica el directori on hauríem de trobar les vistes *jsp* (amb extensió .jsp o .jstl). Podem especificar més d'un directori ,separat per ';'.

- Grup: VIEWS Tag: HTML

Especifica el directori on hauríem de trobar les vistes *html* (amb extensió .html o .htm). Podem especificar més d'un directori ,separat per ';'.

- Grup: VIEWS Tag: NAMES

Ens permet associar 'noms de vista' amb arxius (*html* o *jsp*) , o bé amb controladors que seran executats per el Controlador central. Els noms de vista poden ser 'absoluts' (relatius respecte al servidor d'aplicacions) o noms de fitxer. En el primer cas, l'arxiu es buscarà allà on s'indiqui exactament: p.e. /WEB-INF/views/index.html. En el segon cas, es faran servir els tags anteriors per buscar la localització de l'arxiu més idònia.

Quan s'indica un nom de controlador a ser executar, es pot triar especificar el nom del mètode. Si no es fa, el *framework* agafarà el nom relacionat de la vista i buscarà un mètode dintre de la classe que coincideixi.

¡ - Grup: INTERCEPTOR Tag: PRE

Ens permetrà especificar una classe i mètode que actuarà com a interceptor, executant codi java abans de l'execució del controlador.

- Grup: INTERCEPTOR Tag: POST

Ens permetrà especificar una classe i mètode que actuarà com a interceptor, executant codi java després de l'execució del controlador.

Com es pot apreciar, aquesta configuració que utilitza noms de grup i tags i sintaxi senzilla ens permet definir els aspectes bàsics de la nostra aplicació. Un petit fitxer de configuració d'exemple, permetria al desenvolupador entendre de seguida les possibilitats de configuració i la seva utilització.

Els atributs quedaran recollits en forma de Mapa amb un parell (clau,valor) per ser 'exportats' posteriorment al Controlador central en forma d'atributs.

L'anàlisi de les anotacions el farem utilitzant una llibreria estàndard (*javassist*) que permet tenir un control força potent del arxius de classes instal·lats a l'aplicació.

El procés de funcionament del *servlet* central (*DispatcherServlet*) serà el següent:

. El servidor d'aplicacions inicia el *framework* executant el *ContextListener*, el qual crea una instància de la classe Configuracio i processa les anotacions amb l'ajuda de la llibreria *javassist*. Posteriorment, converteix els parells (clau=group@tag-valor) trobats en atributs que seran accessibles des del Dispatcher Servlet.

. El Dispatcher Servlet, en el moment d'iniciar-se, executa el mètode 'init', el qual llegeix els atributs creats anteriorment des del ContextListener i els processa, segons la seva natura. En quant al tag CONTROLLERS@SCAN, pren els noms dels paquets del valor i amb l'ajuda de la llibreria *javassist* realitza una búsqueda de

les classes contingudes en els paquets especificats, que continguin l'anotació '@Controller'. Llavors crea un mapa de manejadors segons el nom del mètode i el tipus de petició. Aquest mapa, servirà posteriorment per trobar les vistes especificades al fitxer de configuració amb el tag VIEWS@NAMES que especifiquin noms de controladors en comptes de pàgines (vistes) directament. Això difereix del comportament stàndard de Spring en què aquest es fixa en el valor donat a les anotacions '@Controller' i '@RequestMapping' per establir el camí de la petició, però penso que representa una característica que clarifica el comportament del *framework* i facilita la seva comprensió. D'aquesta manera, les peticions als controladors s'hauran de realitzar en funció del nom de la vista, entesa des d'un punt de vista més àmpli (fluxe de l'aplicació)

. Un cop la petició es rebuda per el *Dispatcher Servlet,* aquest separa els diferents components (contextpath, servletpath i infopath) per obtenir el camí que caldrà processar. En aquest punt, vull fer notar que al fitxer de configuració web.xml és on s'indica quines peticions rebrà el Dispatcher Servlet, definir un camí com '/' produeix un efecte de retroalimentació infinit en el *framework*, ja que aquest no renderitza les pàgines, sinó que les propaga per què sigui el propi contenidor d'aplicacions el que realitzi aquesta feina. La configuració esmentada abans, fa que la petició torni a entrar al *Dispatcher Servlet* i torni a executar-se de nou. He necessitat introduir un paràmetre més restrictiu '/mvc/*' per evitar aquest efecte, la qual cosa fa que l'aplicació s'executi amb la seguent *URL*:

http://localhost:8080/winter/mvc/<viewname>

Aquesta adreça executa el *Dispatcher Servlet* i realitza la cerca de la vista demanada com s'explicarà a continuació:

. Un cop esbrinat el camí de la petició, passarem a obtenir quina de les vistes o controladors associats s'ha de processar. Buscarem en el mapa generat anteriorment quin fitxer web (html o jsp) procedeix a renderitzar o en el cas que sigui un controlador, l'executarem i recuperarem el seu resultat per iniciar la búsqueda de nou. L'ordre de cerca de les vistes, correspondrà en primer lloc als directoris especificats a VIEWS@ANY, i posteriorment a VIEWS@HTML, VIEW@JSP en funció del tipus de vista requerida. En quant als controladors, aquests es trobaran als mapes obtinguts a arrel del tag CONTROLLERS@SCAN. Això ens permetrà obtenir la referència a la classe del controlador i mètode que cal executar, per obtenir un altre identificador (resultat de l'execució) que ens permetrà reiniciar el procés de resolució. D'aquesta manera, un controlador podria enllaçar amb un altre, o amb una vista resultat.

. Quan obtenim el fitxer a renderitzar crearem la vista i cridarem el mètode render de la classe UrlView, que farà un *forward* del fitxer per què es pugui mostrar en pantalla, delegant la part corresponent a la renderització al servidor d'aplicacions on estarà instal·lada l'aplicació web.

. Quan obtenim el nom del controlador, instanciarem la classe i executarem el mètode utilitzant també un mètode 'render', corresponent a la classe ControllerView.

. Afegim una tercera classe ‘ContentView’ en cas que la renderització correspongués a un fitxer que cal escriure de forma directa a través de la classe IO.PrintWriter.

. Tindrem en compte els tags INTERCEPTOR@PRE i INTERCEPTOR@POST per instanciar les classes i executar els mètodes corresponents abans i després de processar la crida als controladors. Donat que especifiquem classe:mètode, no serà necessari que els mètodes corresponents tinguin un nom determinat.

Conclusions

1. Conclusions

Els *frameworks* de presentació s'han establert com una eina de disseny útil en el desenvolupament de pàgines web dinàmiques. La relació que estableixen entre les vistes (pàgines web) i el model (classes java) faciliten al programador la manipulació dels controls i la interactuació amb l'usuari.

El patró de disseny MVC és la opció que permet una perfecte diferenciació entre els diferents estadis de l'aplicació:

. El disseny de vistes, part gràfica de l'aplicació habitualment desenvolupades per un dissenyador gràfic.

. El model , negoci de l'aplicació, que implementa la lògica de l'aplicació.

. La persistència, o la gestió de les dades de l'aplicació.

Aquest disseny per capes, permet diferenciar els rols de desenvolupament adequats i desacobla els diferents aspectes de l'aplicació web.

Les solucions actuals existents son força nombroses, variades, i evolucionades degut a l'interés que desperta la programació web. Així mateix, el coneixement d'aquests entorns de treball constitueixen un valor afegit al currículum a la hora de trobar feina com a desenvolupador.

Resulta certament difícil proposar solucions més complertes que les que existeixen actualment, però tot i així , s'ha proposat implementar un nou marc de treball basat en Spring que faciliti la configuració mitjançant l'ús d'una classe Java POJO. S'ha implementat un *framework* que inclou la gestió de controladors, manejadors, localització de vistes, i preveu l'ús d'interceptors i internacionalització.

Mitjançant aquesta classe, i seguint una senzilla sintaxi, el desenvolupador pot configurar els paràmetres sense utilitzar els complexes fitxers XML que en el marc de treball original Spring s'utilitzen habitualment.

El projecte recupera la informació de la classe en temps d'execució i assigna els atributs corresponents per ser utilitzats segons convingui. També deixa oberta la implementació de la resta de capacitats que composaran el producte final.

Aquest projecte m'ha aportat un coneixement profund de funcionament intern dels f*rameworks* de presentació existents, amb els que fins ara no hi havia treballat mai, la qual cosa em permetrà aprofundir en el disseny i implementació d'aplicacions web, un àrea de treball que m'interessa especialment a nivell professional.

Glossari

Framework. Entorn de treball multi-propòsit que facilita el desenvolupament d'aplicacions.

Handlers (Manejadors). Mètodes java que s'executen segons uns criteris definits per l'aplicació.

HTML (HyperText Markup Language). Llenguatge de marques (veure XML) destinat a la creació de documents web.

HTTP (Hypertext Transfer Protocol). Protocol d'intercanvi d'informació per internet.

Interceptors. Mètodes que permeten executar codi java en diferentes fases de l'aplicació, habitualment abans o després d'algún succés determinat.

Internacionalització. Capacitat de les aplicacions per gestionar els missatges a l'usuari en diferents idiomes.

J2EE. Estàrdard de la indústria del desenvolupament, creat per Sun Microsystems i molt utilitzat en la creació d'aplicacions web.

JSP (Java servlet pages). Tecnologia que permet generar respostes dinàmiques a peticions HTTP. Permet incrustar codi java en pàgines HTML.

MVC (Model-Vista-Controlador). Patró de disseny utilitzat en el desenvolupament d'aplicacions que defineix una estructura en capes desacoblades que permeten augmentar el manteniment, la sostenibilitat i l'actualització de l'aplicació.

POJO. Objecte simple de java.

Servlet. Mini aplicació java preparada per ser executat en un servidor d'aplicacions i que permeten recollir les entrades en forma de peticions HTTP i generar sortides, com per exemple, pàgines web.

XML (*e**X**tensible **M**arkup **L**anguage)*. Meta-llenguatge desenvolupat per el World Wide Web Consortium , basat en etiquetes , organitzat en forma d'arbre i de propòsit general.

Referències

. *Apunts UOC – Enginyeria del programari de components i sistemes distribuits.*
. *Apunts UOC – Enginyeria del programari orientat a l'objecte.*
. *http://static.springsource.org/spring/docs/current/spring-framework-. reference/html/mvc.html*
. *http://www.librosweb.es/jobeet/capitulo4/la_arquitectura_mvc.html*
. *http://www.dzone.com/tutorials/java/struts-2/struts-2-example/struts-2-interceptors-example-1.html*
. *http://www.dzone.com/tutorials/java/spring/spring-interceptor.html*
. *http://www.slideshare.net/Markox/introduccion-struts2*
. *http://www.arumeinformatica.es/blog/introduccion-a-apache-tapestry/*
. *http://stackoverflow.com/questions/2668328/spring-mvc-vs-seam*
. *http://www.andygibson.net/articles/seam_spring_comparison/html_single/*
. *http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks*
. http://static.springsource.org/spring/docs/3.0.0.M3/reference/html/ch16s04.html
. http://www.jpalace.org/docs/mvc/mvc.html