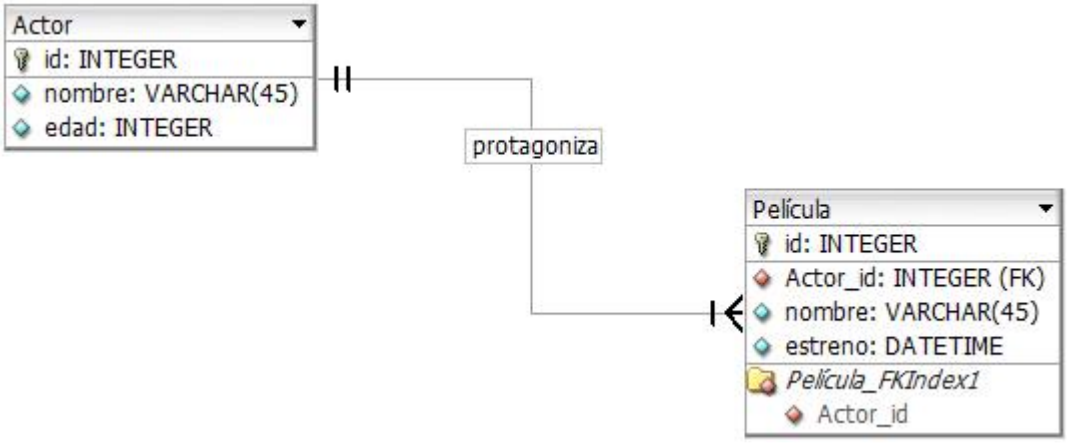
INTEGRACIÓN SPRING + HIBERNATE

OBJETIVO

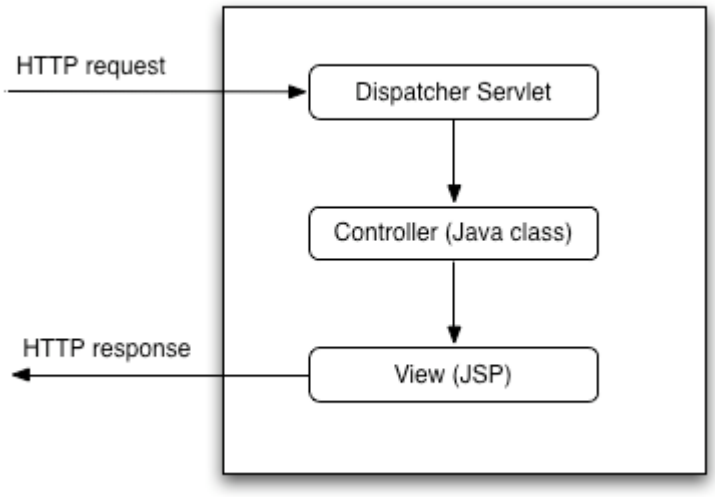
- ✓ *Creación de un proyecto de integración de SPRING HIBERNATE*

PROBLEMA



CONCEPTOS PREVIOS

Esquema de trabajo de SPRING MVC

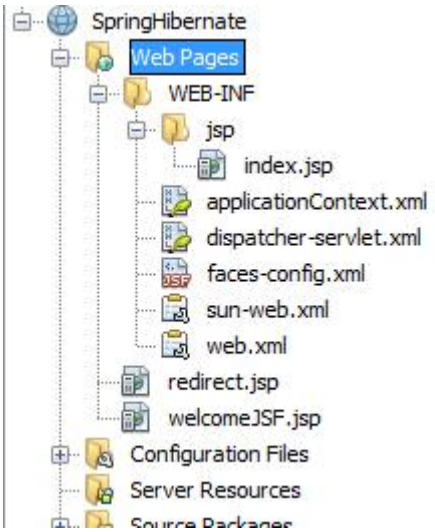


CREACIÓN DE PROYECTO

Creamos un proyecto Web indicando que los frameworks que se utilizarán serán: Spring MVC, Hibernate y Java Server Faces. Recordar que al seleccionar Hibernate se debe seleccionar la conexión a la base de datos que contiene las tablas del tutorial.

En mi caso el proyecto lo he llamado SpringHibernate lo que deja la distribución de los archivos del proyecto como se indica en la siguiente

figura. Algunos de estos archivos serán modificados a lo largo del tutorial, así como se agregaran otros.



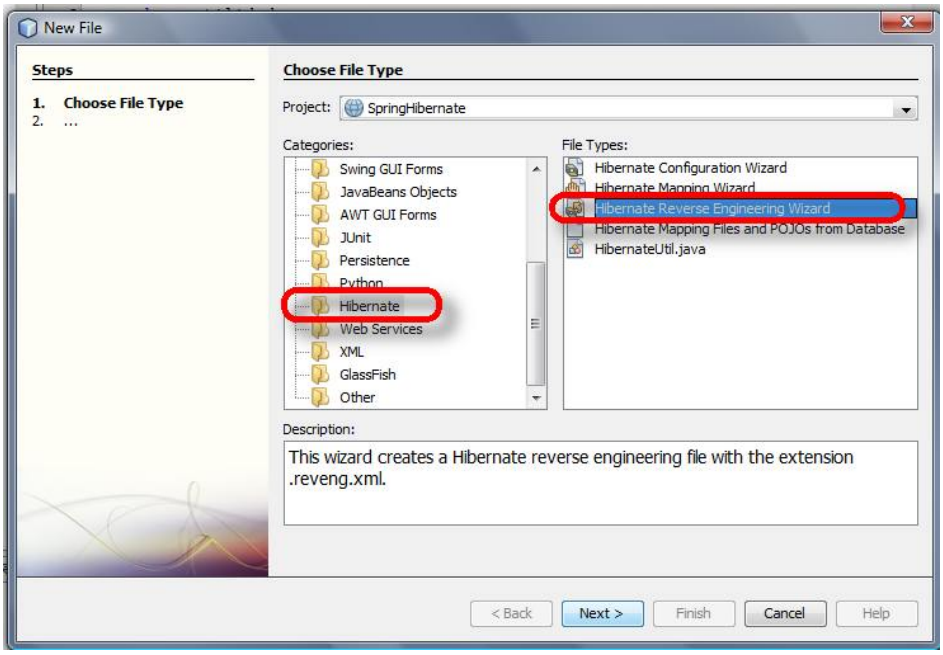
CREACIÓN DE PAQUETES – ORGANIZACIÓN DE CLASES

Se van a crear los siguientes paquetes: bean, bo, dao, negocio.

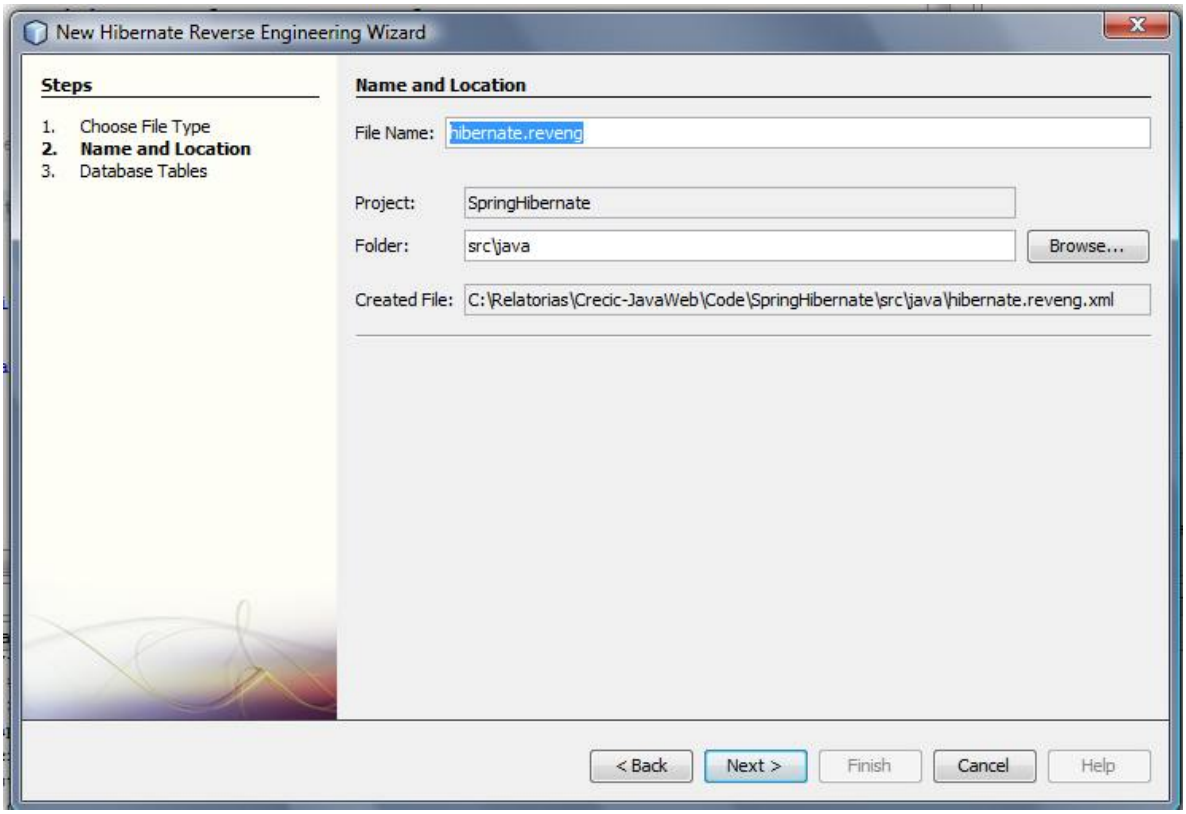
CREACIÓN DE ARCHIVOS POJO

Se van a crear las clases a partir del mapeo de las tablas que contiene el modelo de datos con el cual se va a trabajar. Las clases POJO deberán quedar en el paquete negocio.

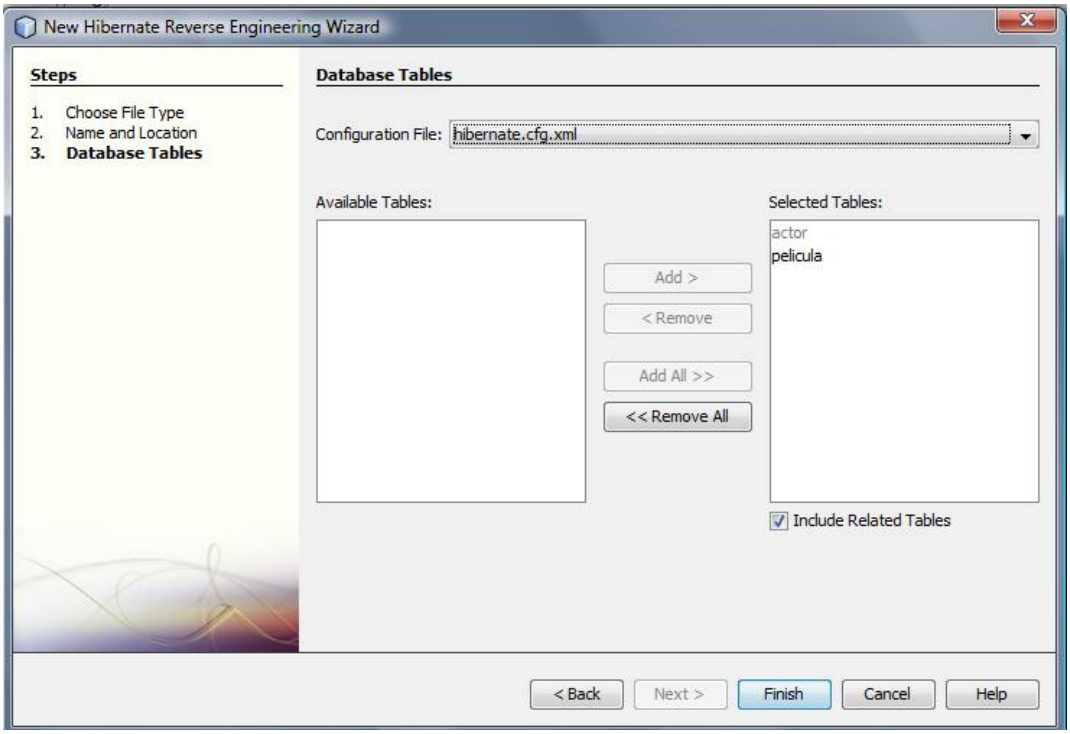
Ahora creamos el asistente para la ingeniería inversa:



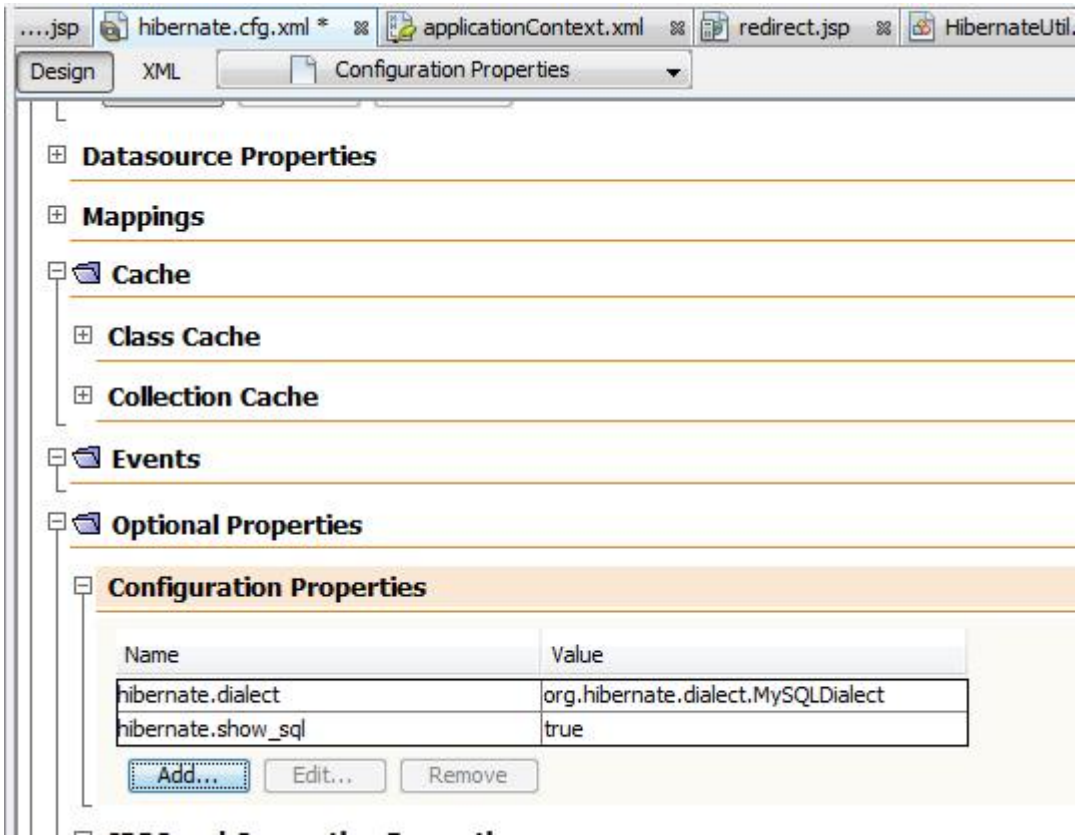
No cambiamos ningún valor y hacemos clic en Next:

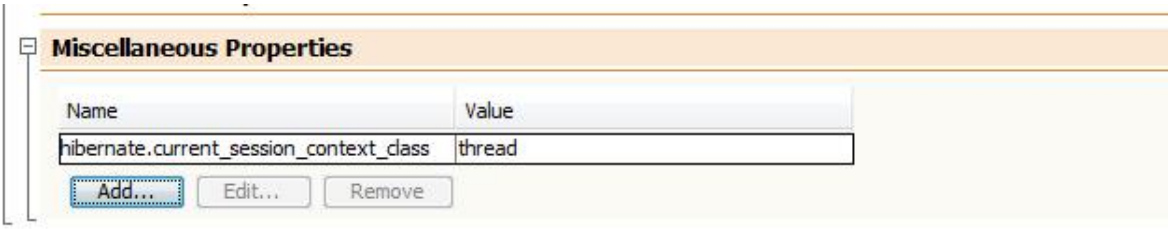


Se seleccionan las tablas que forman parte del modelo y clic en Finish

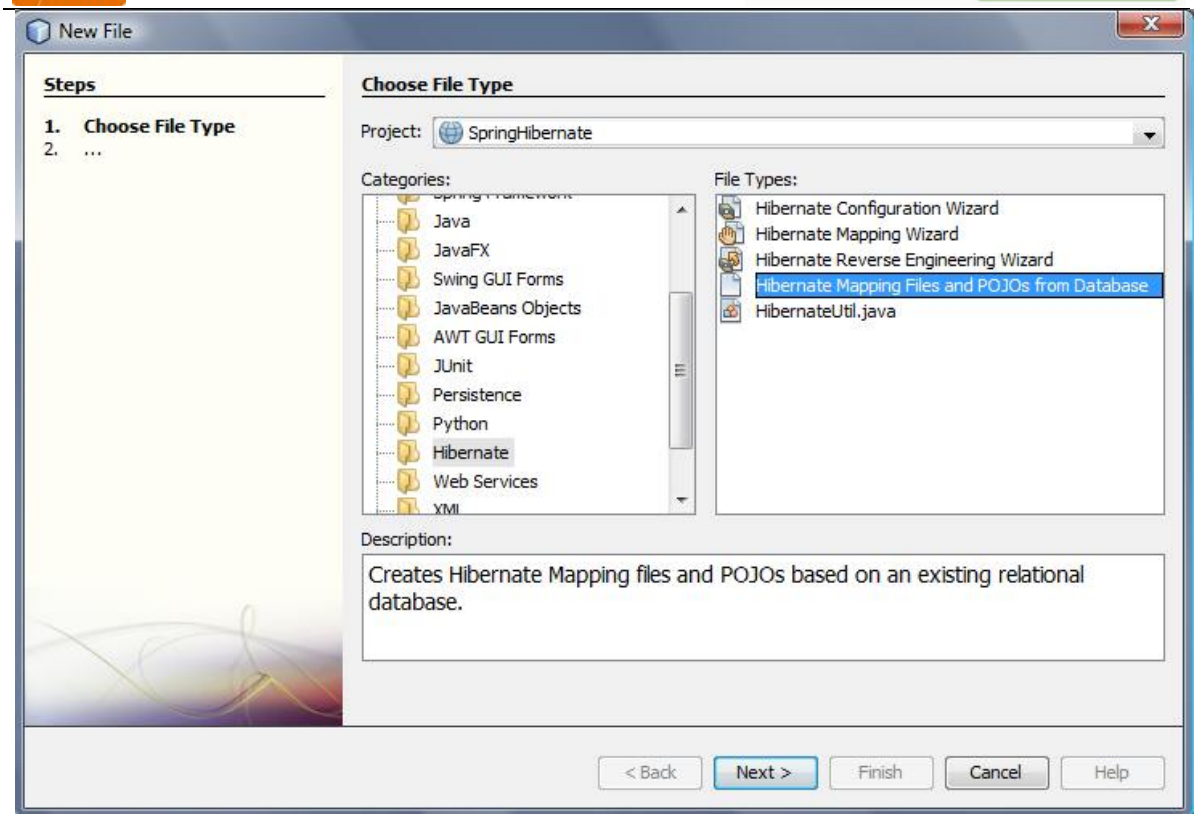


Antes de continuar, y para no olvidarlo después, se hacen los cambios en el XML de la configuración de Hibernate:

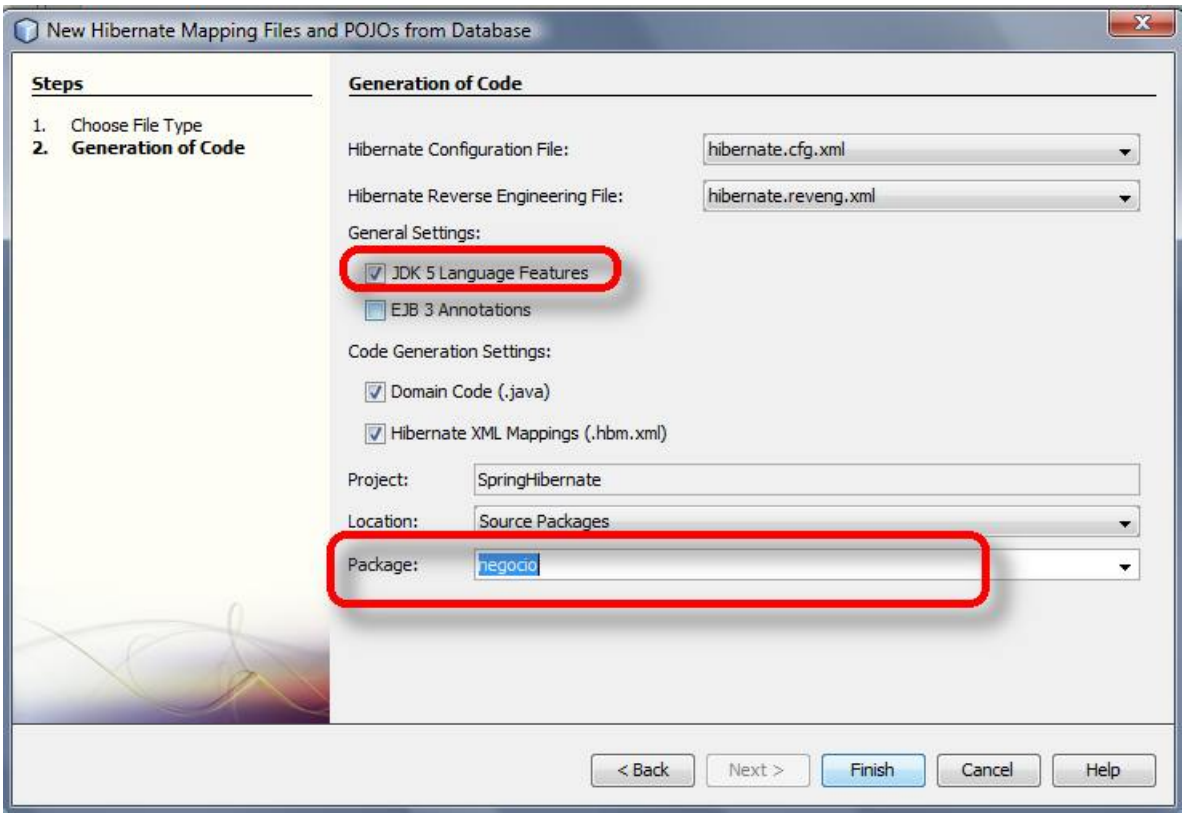




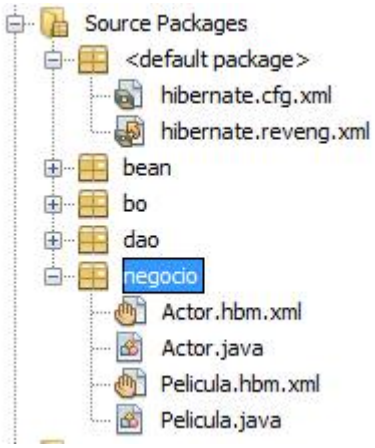
Ahora si estamos en condiciones de generar las clases que representan el mapeo entre el mundo relacional y el orientado a objetos:



Seteamos algunos valores y clic en Finish



Las clases se han creado y la estructura de los archivos de mi proyecto es:



Hasta ahora hemos considerado en nuestro proyecto la fuente de los datos, lo que viene a corresponder a la parte de modelo (la M de MVC) de la arquitectura de Spring. Ahora debemos trabajar en la vista (la V de MVC) y el controlador (la C de MVC).

LA CAPA DE ACCESO A DATOS (DAO)

```
6   package dao;
7
8   import java.util.List;
9   import negocio.Actor;
10
11  /**
12   *
13   * @author Yasna Meza Hidalgo
14   */
15  public interface IActorDAO {
16      public void addActor(Actor a);
17      public List<Actor> findAllActor();
18      public int generarID();
19  }
```

```java
package dao;

import java.util.List;
import negocio.Actor;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

/**
 *
 * @author Yasna Meza Hidalgo
 */
public class ActorDAO extends HibernateDaoSupport implements IActorDAO {

    public void addActor(Actor a){
        getHibernateTemplate().saveOrUpdate(a);
    }

    public List<Actor> findAllActor(){
        return getHibernateTemplate().find("from Actor");
    }

    public int generarID(){
        List<Actor> lista;
        lista = getHibernateTemplate().find("from Actor as a order by a.id desc");
        if (lista.size() == 0) return 1;
        return lista.get(0).getId() + 1;
    }
}
```

La clase ActorDAO se extiende de la clase **HibernateDaoSupport** lo que facilitará el acceso a los datos almacenados en la base de datos. Es conveniente hacer notar que sólo invocando al método getHibernateTemplate() se pueden implementar las operaciones sobre la base de datos, en este caso, actualizar registro (método saveOrUpdate()) y listar (método find())

LA CAPA DE LÓGICA DE NEGOCIO Y EL MARCO DE TRABAJO SPRING

Los objetos y servicios de negocio existen en la capa de lógica de negocio. Un objeto de negocio no sólo contiene datos, también la lógica asociada con ese objeto específico. En la aplicación de ejemplo se han identificado dos objetos de negocio: Actor y Película.

Los servicios de negocio interactúan con objetos de negocio y proporcionan una lógica de negocio de más alto nivel. Se debería definir una capa de interfaz de negocio formal, que contenga los interfaces de servicio que el cliente utilizará directamente. POJO, con la ayuda del marco de trabajo Spring, implementará la capa de lógica de negocio de la aplicación. Hay dos servicios de negocio: IActorBO contiene la lógica de negocio relacionada con el manejo de los actores, y IPeliculaBO contiene la lógica de manejo de las películas.

```java
package bo;

import java.util.List;
import negocio.Actor;

/**
 *
 * @author Yasna Meza Hidalgo
 */
public interface IActorBO {
    public void addActor(Actor a);
    public List<Actor> findAllActor();
    public int generarID();
}
```

Ahora definimos la clase para implementar los servicios:

```java
package bo;

import dao.ActorDAO;
import java.util.List;
import negocio.Actor;

/**
 *
 * @author Yasna Meza Hidalgo
 */
public class ActorBO implements IActorBO{
    private ActorDAO actorDAO;

    public void setActorDAO(ActorDAO actorDAO){
        this.actorDAO = actorDAO;
    }
    public void addActor(Actor a){
        this.actorDAO.addActor(a);
    }
    public List<Actor> findAllActor(){
        return this.actorDAO.findAllActor();
    }

    public int generarID(){
        return this.actorDAO.generarID();
    }
}
```

NOTA. Las interfaces/clases anteriores DEBERÁN estar incluidas en el paquete ***bo***.

CLASES DEL NEGOCIO (MODELO/NEGOCIO)

Estas clases son las que fueron generadas a través del proceso de ingeniería inversa.

CAPA DE PRESENTACIÓN

La implementación de la capa de presentación implica crear las páginas JSP, definir la navegación por las páginas, crear y configurar los beans de respaldo, e integrar JSF con la capa de lógica de negocio.

Primero vamos a revisar la clase que define el bean:

```
6   package bean;
7
8   import bo.ActorBO;
9   import java.util.List;
10  import javax.faces.application.FacesMessage;
11  import javax.faces.component.html.HtmlDataTable;
12  import javax.faces.context.FacesContext;
13  import negocio.Actor;
14
15  /**
16   *
17   * @author Yasna Meza Hidalgo
18   */
19  public class ActorBean implements java.io.Serializable{
20      private String nombre;
21      private int edad;
22      private int id;
23
24      // DI via Spring
25      private ActorBO actorBO;
26
27      /* Algunos atributos para la presetación */
28      private List<Actor> lista;
29      private  HtmlDataTable tabla;
30      private boolean seleccion;
```

Algunos métodos getter/setter:

```
32      public void setActorBO(ActorBO actorBO){
33          this.actorBO = actorBO;
34      }
35
36      public ActorBO getActorBO(){
37          return actorBO;
38      }
39
40      public HtmlDataTable getTabla() {
41          return tabla;
42      }
43
44      public void setTabla(HtmlDataTable tabla) {
45          this.tabla = tabla;
46      }
```

```
93      /**
94       * @return the nombre
95       */
96      public String getNombre() {
97          return nombre;
98      }
99
100     /**
101      * @param nombre the nombre to set
102      */
103     public void setNombre(String nombre) {
104         this.nombre = nombre;
105     }
106
107     /**
108      * @return the edad
109      */
110     public int getEdad() {
111         return edad;
112     }
113
114     /**
115      * @param edad the edad to set
116      */
117     public void setEdad(int edad) {
118         this.edad = edad;
119     }
```

```
56      public List<Actor> getLista(){
57          lista = actorBO.findAllActor();
58          return lista;
59      }
```

Un método importante:

```
61      public String addActor(){
62          Actor a = new Actor();
63          /* Verifica si es nuevo o seleccionado */
64          if (!this.seleccion) a.setId(actorBO.generarID());
65          else a.setId(id);
66
67          a.setNombre(this.getNombre());
68          a.setEdad(this.getEdad());
69
70          try{
71              actorBO.addActor(a);
72              FacesContext facesContext = FacesContext.getCurrentInstance();
73              facesContext.addMessage("formActor:btn_guardar", new FacesMessage(
74              FacesMessage.SEVERITY_INFO, "Actor actualizado satisfactoriamente", null));
75          }
76          catch(Exception e){
77              FacesContext facesContext = FacesContext.getCurrentInstance();
78              facesContext.addMessage("formActor:btn_guardar", new FacesMessage(
79              FacesMessage.SEVERITY_INFO, "Error al actualizar datos", null));
80          }
81          finally{
82              limpiarForm();
83              this.seleccion = false;
84          }
85          return "";
86      }
```

```
48      public void seleccionLista(){
49          Actor a = (Actor) tabla.getRowData();
50          this.id = a.getId();
51          this.edad=a.getEdad();
52          this.nombre=a.getNombre();
53          this.seleccion = true;
54      }
```

Finalmente, el método para "limpiar" el formulario:

```
88      public void limpiarForm(){
89          this.setNombre("");
90          this.setEdad(0);
91      }
```

ARCHIVOS DE CONFIGURACIÓN

A continuación se detallarán los archivos de configuración con los cuales se trabajará.

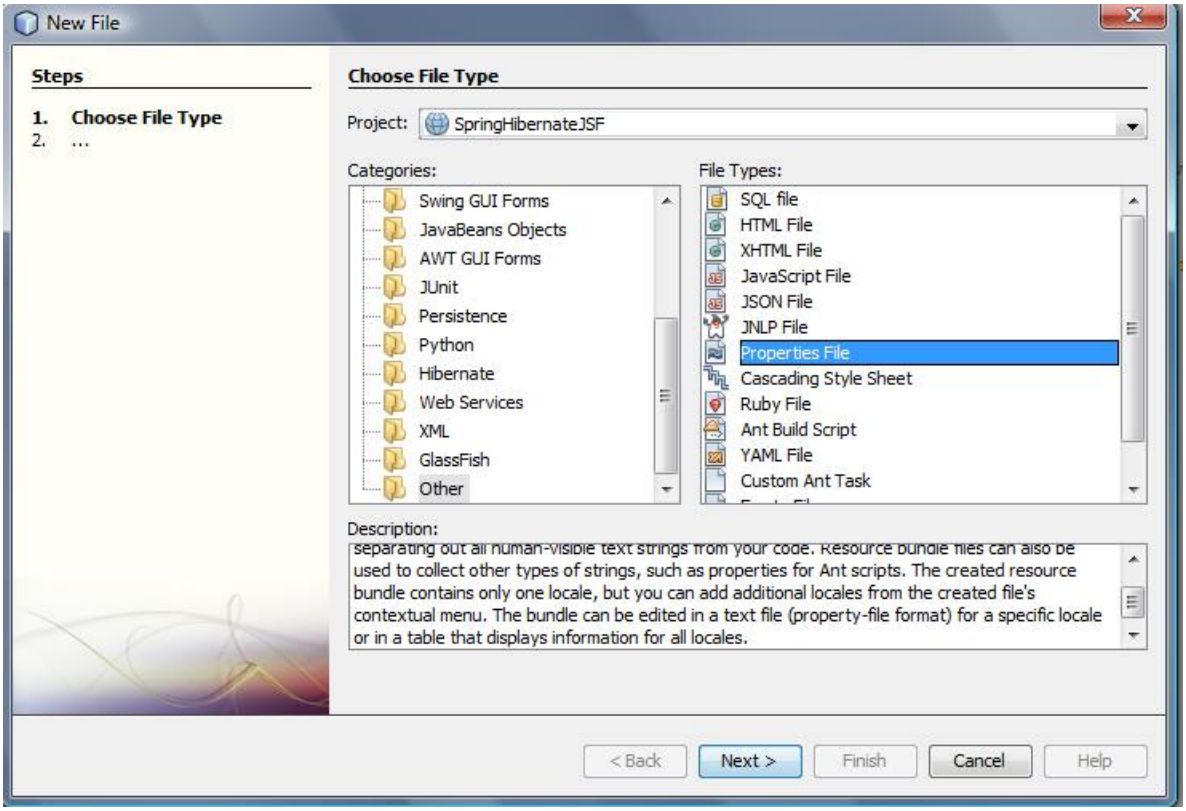
Archivo applicationContext.xml

Contiene las definiciones de los beans que usará la aplicación. Hay una parte que viene comentada en el archivo que tiene el proyecto y se va a descomentar, que consiste en la definición de dos beans que tienen que ver con la fuente de datos:

```
<bean id="propertyConfigurer"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
      p:location="/WEB-INF/jdbc.properties" />

<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="${jdbc.driverClassName}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}" />
```

Como se aprecia en el código anterior se pueden definir los parámetros de conexión con la base de datos en un archivo llamado jdbc.properties, así es que creamos un archivo de properties en nuestro proyecto:



El contenido de este archivo DEBE ser el que se indica a continuación:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/nombre-base-de-datos
jdbc.username=user
jdbc.password=password
```

En caso de que haya configurado el motor de base de datos para que no utilice password, entonces simplemente **NO INCLUYA LA LÍNEA.**

Ahora agregamos el bean para la configuración de la base de datos propiamente tal y tenemos:

```
<bean id="sessionFactory"
   class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

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

<property name="hibernateProperties">
   <props>
     <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
     <prop key="hibernate.show_sql">true</prop>
     <prop key="hibernate.current_session_context_class">thread</prop>
   </props>
</property>
<property name="mappingResources">
    <list>
      <value>negocio/Pelicula.hbm.xml</value>
      <value>negocio/Actor.hbm.xml</value>
    </list>
 </property>
</bean>
```

Lo anterior define un bean llamado ***sessionFactory*** el cual contiene trs propiedades:

- dataSource
- hibernateProperties
- zappingResources

El property (propiedad) dataSource usa como referencia el bean definido anteriormente. En el caso del property hibenateProperties tiene los ítems (key) que se encuentran en el archivo de configuración de Hibernate (que se encuentra en el archivo ***hibernate.cfg.xml***. Finalmente, el property mappingResources tiene la lista de XML de mapeo entre las clases y las tablas (que fueron generados al principio del tutorial).

Archivo ActorBean.xml

Contiene las definiciones de los beans asociados a la capa de lógica de negocio y la capa de acceso a datos. El contenido de este archivo DEBERÁ ser:

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
    Document   : ActorBean.xml
    Created on : 21 de julio de 2011, 12:46 PM
    Author     : Yasna Meza Hidalgo
    Description:
        Purpose of the document follows.
-->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://

        <bean id="actorBo" class="bo.ActorBO" >
            <property name="actorDAO" ref="actorDao" />
        </bean>

        <bean id="actorDao" class="dao.ActorDAO" >
            <property name="sessionFactory" ref="sessionFactory" />
        </bean>

</beans>
```

En el se crean los beans asociados a las clases de lógica de negocio y acceso a datos. La lógica dice que la capa de lógica de negocios se conecta con la capa de acceso a datos y de ahí el vínculo entre estos beans.

En este caso cada bean tiene asociada una clase y las property que se definen tienen que ver con los atributos que se están seteando. Si revisamos la clase ActorBO tiene un atributo llamado ActorDAO que es el que estamos seteando. Algo similar sucede con el atributo sessionFactory de la clase ActorDAO, la diferencia es que este atributo viene heredado desde la clase ***HibernateDaoSupport*** de la cual está extendiendo.

Ahora vamos a agregar las siguientes líneas al archivo applicationContext.xml:

```
<!-- Beans Declaration -->
<import resource="ActorBean.xml"/>
```

Con lo anterior estamos "importando" el contenido del XML con la definición de los bean asociados al Actor.

Archivo dispatcher-servlet.xml

Contiene la definición de los beans con los cuales va a trabajar la aplicación. Es un archivo generado al momento de crear el proyecto y ahora le vamos a agregar los beans que acabamos de definir. Las líneas que se deben AGREGAR AL FINAL DEL ARCHIVO son:

```
<!-- Mis propios beans -->
<bean id="actorBo" class="bo.ActorBO" >
    <property name="actorDAO" ref="actorDao" />
</bean>

<bean id="actorDao" class="dao.ActorDAO" >
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Archivo faces-config.xml

Contiene la configuración para trabajar con JSF. También es un archivo generado cuando se crea el proyecto y le agregaremos algunos tag.

```
<application>
<el-resolver>
        org.springframework.web.jsf.el.SpringBeanFacesELResolver
</el-resolver>
</application>
<managed-bean>
        <managed-bean-name>actor</managed-bean-name>
        <managed-bean-class>bean.ActorBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
                <property-name>actorBO</property-name>
                <value>#{actorBo}</value>
        </managed-property>
</managed-bean>
```

Acá estamos definiendo el bean que va a trabajar con la aplicación que tiene asociada la clase ActorBean que fue descrita anteriormente y seteamos el atributo (property) actorBO con el bean definido anteriormente.

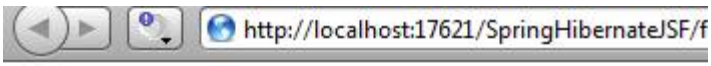
Archivo web.xml

Contiene la configuración general del proyecto, también es un archivo generado y acá sólo vamos a modificar el archivo de bienvenida que, por defecto, viene seteado con redirect.jsp y lo cambiaremos por frmActor.jsp:

```
<welcome-file-list>
    <welcome-file>frmActor_jsp</welcome-file>
    </welcome-file-list>
```

CAPA DE PRESENTACIÓN

Ahora sólo falta crear la capa de presentación para poder probar todo. Vamos a crear una página similar a:



Agregar nuevo actor

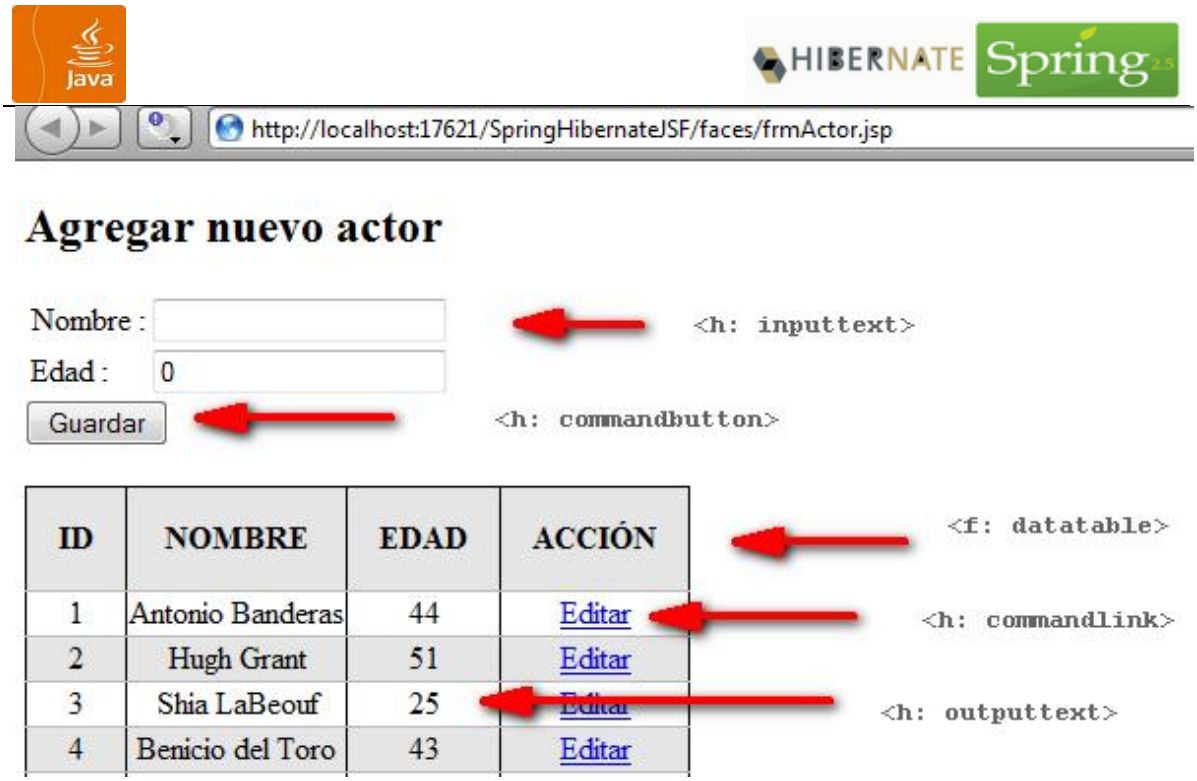
Nombre :

Edad : 0

Guardar

ID	NOMBRE	EDAD	ACCIÓN
1	Antonio Banderas	44	Editar
2	Hugh Grant	51	Editar
3	Shia LaBeouf	25	Editar
4	Benicio del Toro	43	Editar

La siguiente figura muestra algunos elementos que forman parte de la interfaz.



Ahora miremos un poco el código que hay:

La página está formada por una vista principal <f: view> la que contiene dos formularios <h: form> uno para los datos y otro para la tabla. Vamos a revisar el formulario de los datos:

```
<h:form id="formActor">
    <h:messages id="messageList" styleClass="error" showSummary="true" showDetail="false" />
    <h:panelGrid columns="3">
        Nombre :
        <h:inputText id="name" value="#{actor.nombre}"
            size="20" required="true" label="Nombre" ></h:inputText>
        <h:message for="name" style="color:red" />
        Edad :
        <h:inputText id="edad" value="#{actor.edad}"
            size="20" required="true" converter="#{Integer}"
            label="Edad"  >
        </h:inputText>
        <h:message for="edad" style="color:red" />
    </h:panelGrid>
    <h:commandButton id="btn_guardar" value="Guardar" action="#{actor.addActor}" />
</h:form>
```

El formulario está formado por un panel de tres columnas ***<h: panelGrid>***. Las tres columnas son para las etiquetas, los campos de texto ***<h: inputText>*** y el mensaje ***<h: message>***. Fuera del panel está el botón que va a enviar el formulario al servidor ***<h: commandButton>***.

Lo interesante acá es la propiedad value de los inputText, ya que esta propiedad tiene el valor asociado al bean que hemos definido y declarado en los archivos de configuración (recordar que lo llamamos actor en el archivo faces-config.xml). Usando la notación (.) accedemos a los

atributos y métodos definidos en el bean (lo que en el fondo está definido en la clase).

En el caso de la entrada asociada a la edad se tiene una propiedad llamada converter que nos va a permitir "validar" cuando se ingresan letras y algo similar sucede con la propiedad required que obliga a que los campos tengan valores antes de ser enviados al servidor.

Para el caso del botón se asocia a la propiedad action la llamada al método ***addActor()*** que se encuentra en el bean actor: #{actor.addActor}.

Ahora revisemos el segundo formulario (asociado a la tabla):

```
<h:form>
    <h:dataTable value="#{actor.lista}" var="a" border="1" binding="#{actor.tabla}"
            styleClass="order-table"
            headerClass="order-table-header"
            rowClasses="order-table-odd-row,order-table-even-row">
        <h:column>
            <f:facet name="header" >
                <h:outputText value="ID" />
            </f:facet>
            <h:outputText value="#{a.id}"/>
        </h:column>
```

En este extracto de código es posible apreciar que la tabla tiene una propiedad llamada ***binding*** que se relaciona con los datos que serán incluidos en la tabla. En este caso es el atributo tabla del bean actor, revisemos un poco esa parte del código para poder entender eso del binding.

La clase ActorBean tiene, entre otros, estos atributos:

```
/* Algunos atributos para la presetación */
private List<Actor> lista;
private  HtmlDataTable tabla;
private boolean seleccion;
```

Y entre otros métodos están:

```
public HtmlDataTable getTabla() {
    return tabla;
}

public void setTabla(HtmlDataTable tabla) {
    this.tabla = tabla;
}

public void seleccionLista(){
    Actor a = (Actor) tabla.getRowData();
    this.id = a.getId();
    this.edad=a.getEdad();
    this.nombre=a.getNombre();
    this.seleccion = true;
}

public List<Actor> getLista(){
    lista = actorBO.findAllActor();
    return lista;
}
```

Y como se ve en el método getLista() usa los servicios de la clase ActorBO (a través de su atributo actorBO) para obtener la lista de actores (contenido de la tabla).

Cuando en el binding del archivo JSP tenemos #{actor.lista} estamos llamando al método getLista() (acá sirve el hecho de usar la nomenclatura Bean para definir los métodos). La explicación de la propiedad ***value*** de los <h: outputText> salvo que ahora estamos haciendo referencia a un objeto (llamado a) definido en la propiedad ***var*** del <h: dataTable>. En este caso como se está asociando a un outputText se llama al método get del atributo que se está vinculando.

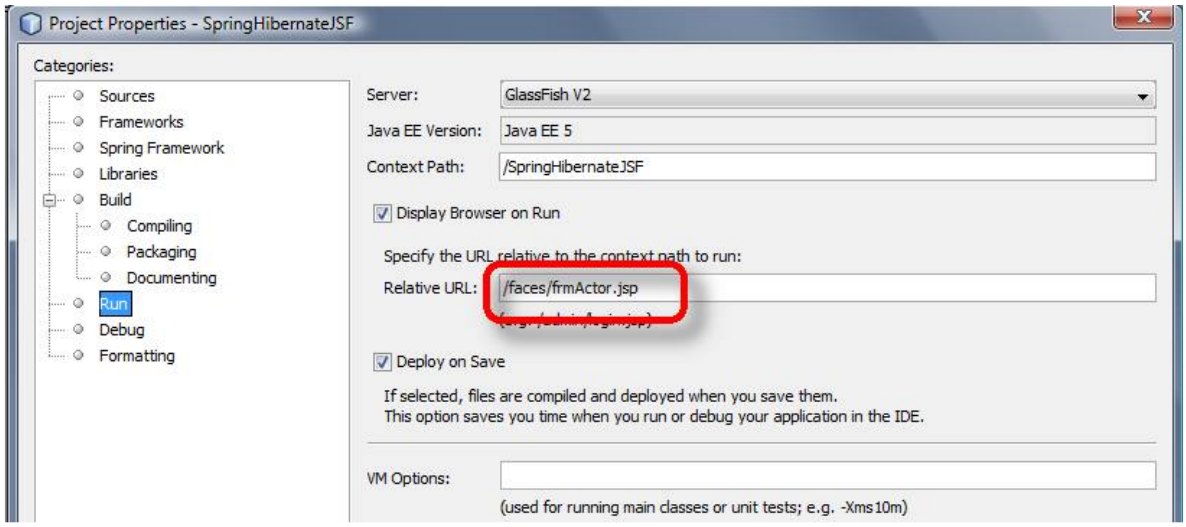
Finalmente, está la explicación del <h: commandLink>:

```
<h:commandLink id="edit" action="#{actor.seleccionLista}" immediate="true"><h:outputText value="Editar"/></h:commandLink>
```

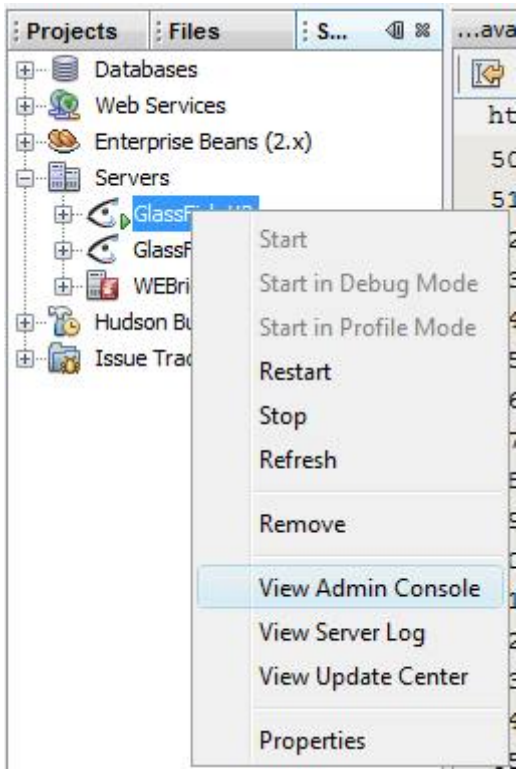
Cuya propiedad ***action*** está asociada al método ***seleccionLista()*** que se encuentra en la clase ActorBean (asociada al bean actor). Ese método selecciona el objeto desde la tabla y modifica los atributos del bean y cuando la página se recarga será ese el objeto que está cargado en memoria y sus datos se desplegarán en los ***<h: inputText>*** del formulario.

ÚLTIMO CAMBIO Y A EJECUTAR EL PROYECTO

El último cambio que debemos realizar es en las propiedades del proyecto según como lo indica la siguiente figura:

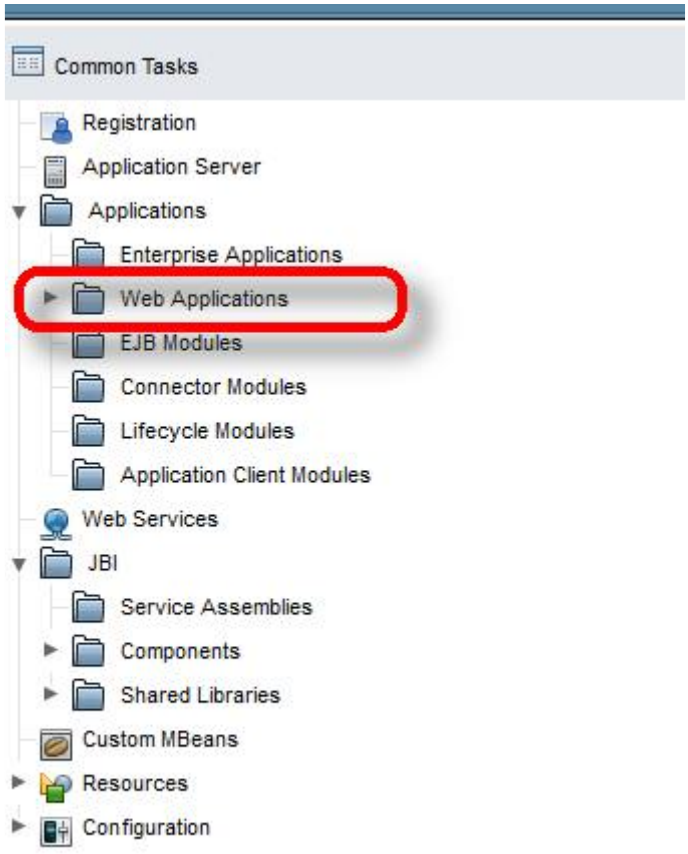


Ahora ejecutamos el proyecto y podemos visualizar el detalle del archivo LOG del servidor y si todo anduvo bien podremos ver cómo se visualiza nuestra "humilde" y "sencilla" página en el navegador. Además podemos ver nuestro proyecto alojado en el servidor, abriendo la consola de administración del servidor:

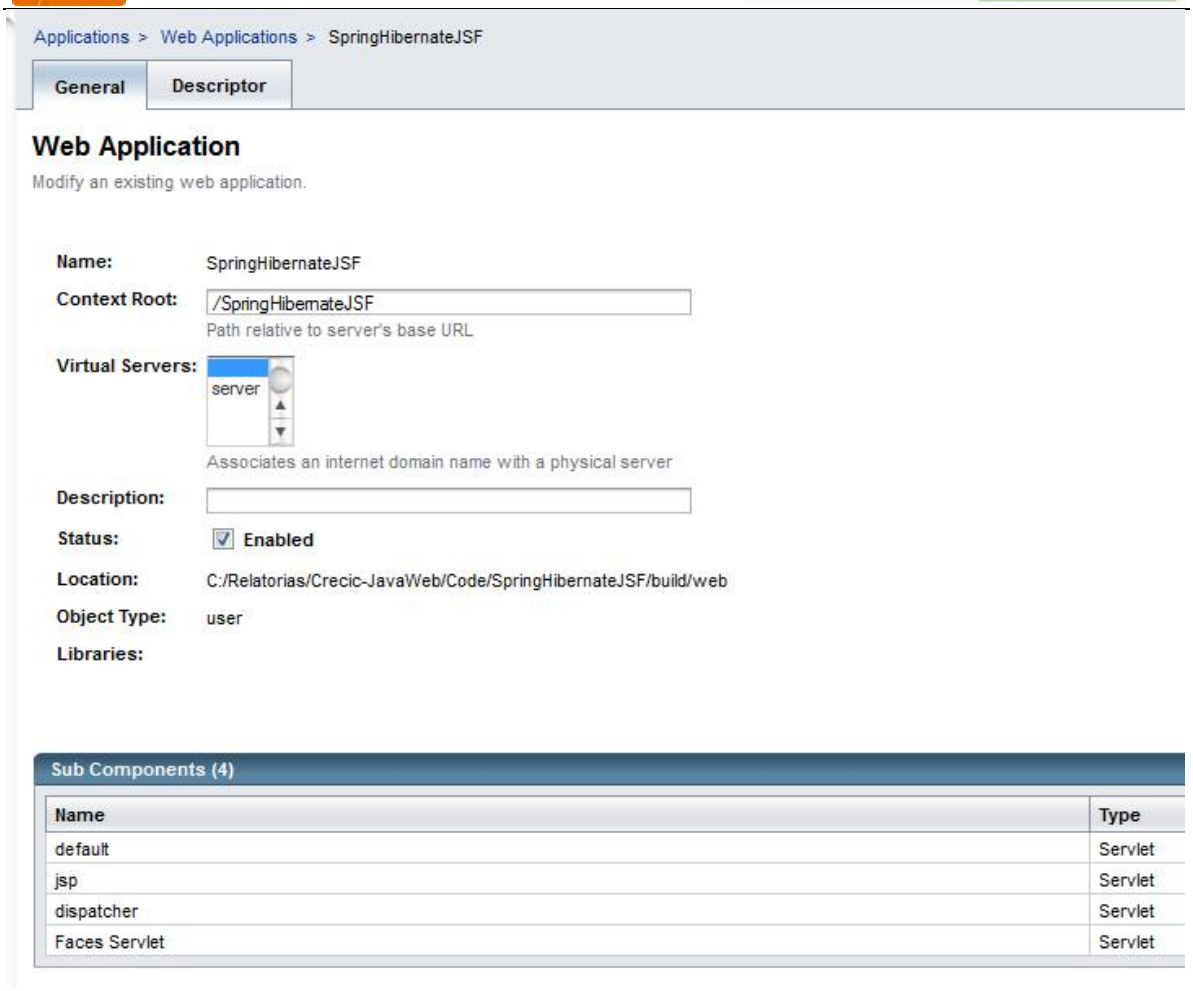


Con lo que llegarás a:

Donde debes ingresar el usuario y clave de acceso del servidor para luego visualizar la interfaz de la consola en donde, entre muchas otras cosas, podremos visualizar el detalle de las aplicaciones alojadas:



En este caso el proyecto alojado muestra las siguientes características:



TRABAJO FUTURO

En la siguiente entrega veremos como mejorar y potenciar la interfaz usando otros controles de JSF y agregar la funcionalidad asociada a las películas.