

Manual de desarrollador

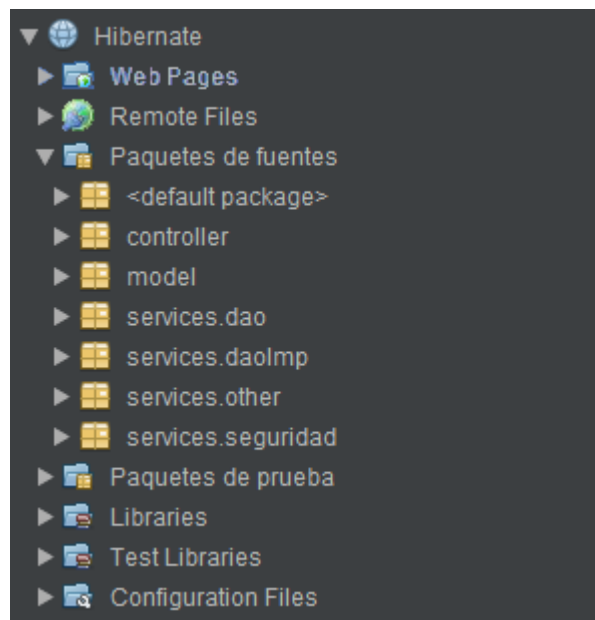
1. Descripción general	pg.1
2. Tecnologías y software empleado	pg.2
3. Librerías	pg.5
4. Base de datos	pg.7
5. Vistas	pg.9
6. Código	pg.13
6.1 Metodología y Creación del proyecto	pg.13
6.2 Configuración	pg.15
6.3 POJO's	pg.18
6.4 Servicios DAO	pg.20
6.5 Controladores	pg.23

1. Descripción general.

MyAcademy Lite se trata de un proyecto para la gestión y administración de una academia de idiomas, aunque eso no lo hace incompatible con otros tipos de academias pues apenas tiene características exclusivas para academias de idiomas.

Permite un vista y control bastante rápido e intuitivo de elementos recurrentes como números de teléfono o emails.

La estructura general del proyecto es la siguiente:



- En la carpeta de Web Pages encontraremos los recursos web correspondientes al Front-end, es decir, todo el material necesario para las vistas. Archivos jsp, css, algún javascript e imágenes.

- En el default package encontramos el archivo de configuración de hibernate “hibernate.cfg.xml” que es dónde configuraremos la conexión a la base de datos y el pool size o tiempo para autocommit (cierre automático de las conexiones).
- En controller encontraremos implementaciones de la clase HttpServlet para hacer los get y post y sacar o recoger información de ellas.
- En model encontraremos los pojos, es decir, todas aquellas entidades que han sido mapeadas de la base de datos. Además de sus correspondientes archivos .hbm.xml que son los que utiliza hibernate para crear la entidad. Estos archivos han sido generados gracias al archivo hibernate.reveng.xml, archivo de ingeniería inversa de hibernate además del hibernate.cfg.xml
- En services.dao. hemos creado las interfaces que utilizaremos para acceder a los datos de la base de datos.
- En services.daoImp hemos implementado las interfaces anteriores escribiendo todos los métodos necesarios para hacer nuestras operaciones contra la base de datos. Todas las implementaciones llevan un CRUD básico además de algunas consultas personalizadas con Hibernate Query Language (HQL).
- Por último en el paquete de services.seguridad hemos creado una clase llamada Cifrador para poder cifrar todas las contraseñas entrantes y guardarlas y compararlas con la base de datos.

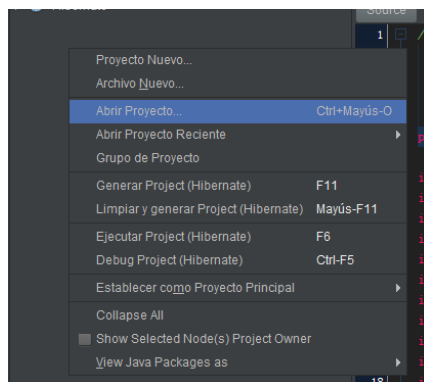
2. Tecnologías empleadas

En esta sección enumeraremos y haremos una breve descripción de las principales tecnologías empleadas.

Nuestro IDE ha sido **Netbeans**. Podemos descargarlo desde aquí:

<https://netbeans.org/features/index.html>

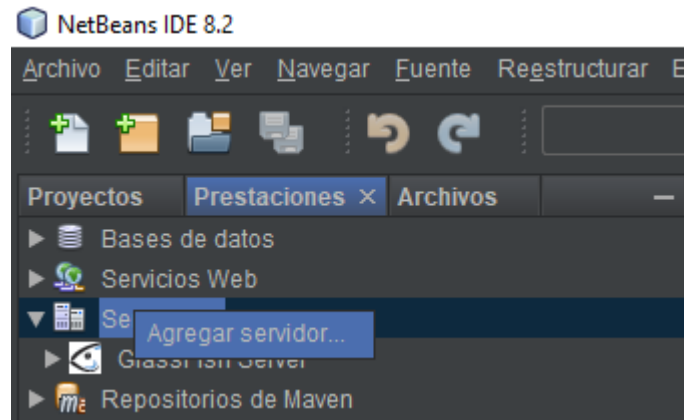
Para continuar con el desarrollo del proyecto solo tenemos que abrir el archivo comprimido, abrir nuestro IDE y abrir el proyecto usando click derecho o Control+Mayusc+O



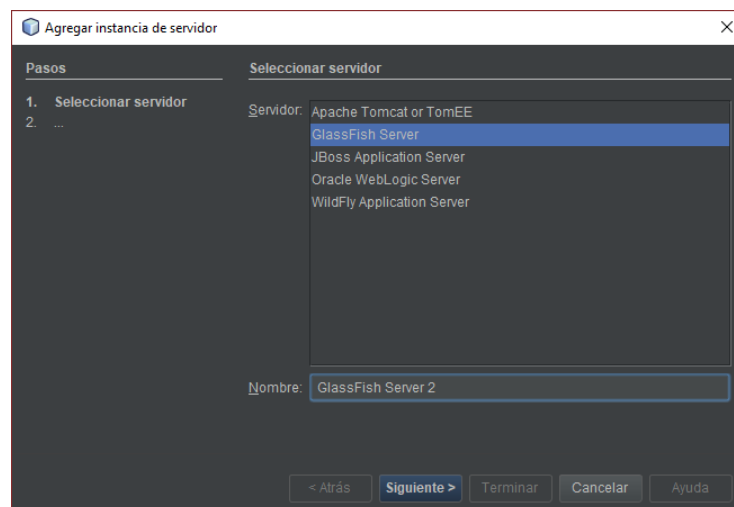
El servidor web que hemos usado ha sido **GlassFish** que viene ya integrado con Netbeans para poder descargarlo. Tambien podemos descargarlo desde aquí:

<https://javaee.github.io/glassfish/download>

Para su instalación en nuestro IDE irnos a Prestaciones (Services en la versión inglesa) y con el click derecho añadiremos un nuevo servidor web.



En la siguiente ventana nos pedirá el tipo de servidor, seleccionamos GlassFish y le asignamos el nombre que deseemos.



Asignamos los diferentes puertos y un nombre de usuario y contraseña si así lo deseamos

Agregar instancia de servidor

Pasos

1. Seleccionar servidor
2. Server Location
3. Domain Name/Location

Domain Location

Domain: domain2

Host: localhost ☒ Loopback

DAS Port: 4848 HTTP Port: 8080 ☐ Default

Target:

User Name: usuario

Password:

[Create new embedded domain: domain2](#)

< Atrás Siguiente > Terminar Cancelar Ayuda

Una vez creado el servidor podemos seguir con el desarrollo del proyecto. La primera vez que corramos el proyecto nos solicitará un servidor y le asignamos el que hemos creado.

Hemos usado el **framework Hibernate** para mapear la base de datos, generar una capa de persistencia y realizar las operaciones de acceso a la base de datos. Las librerías necesarias vienen en Netbeans pero igualmente se puede descargar de aquí:

<http://hibernate.org/orm/>

JSP: Son funcionalmente páginas web basadas en HTML y XML y Java. El rendimiento de una página JSP es el mismo que tendría el servlet equivalente, ya que el código es compilado como cualquier otra clase Java. A su vez, la máquina virtual compilará dinámicamente a código de máquina las partes de la aplicación que lo requieran. Esto hace que JSP tenga un buen desempeño y sea más eficiente que otras tecnologías web que ejecutan el código de una manera puramente interpretada.

El **HTTPServlet** es una clase en el lenguaje de programación Java, utilizada para ampliar las capacidades de un servidor. Aunque los servlet pueden responder a cualquier tipo de solicitudes, éstos son utilizados comúnmente para extender las aplicaciones alojadas por servidores web, de tal manera que pueden ser vistos como applets de Java que se ejecutan en servidores en vez de navegadores web. Este tipo de servlets son la contraparte Java de otras tecnologías de contenido dinámico Web, como PHP

*Nota: La palabra servlet deriva de otra anterior, applet, que se refiere a pequeños programas que se ejecutan en el contexto de un navegador web.

Bibliografía y documentación de interés para su desarrollo:

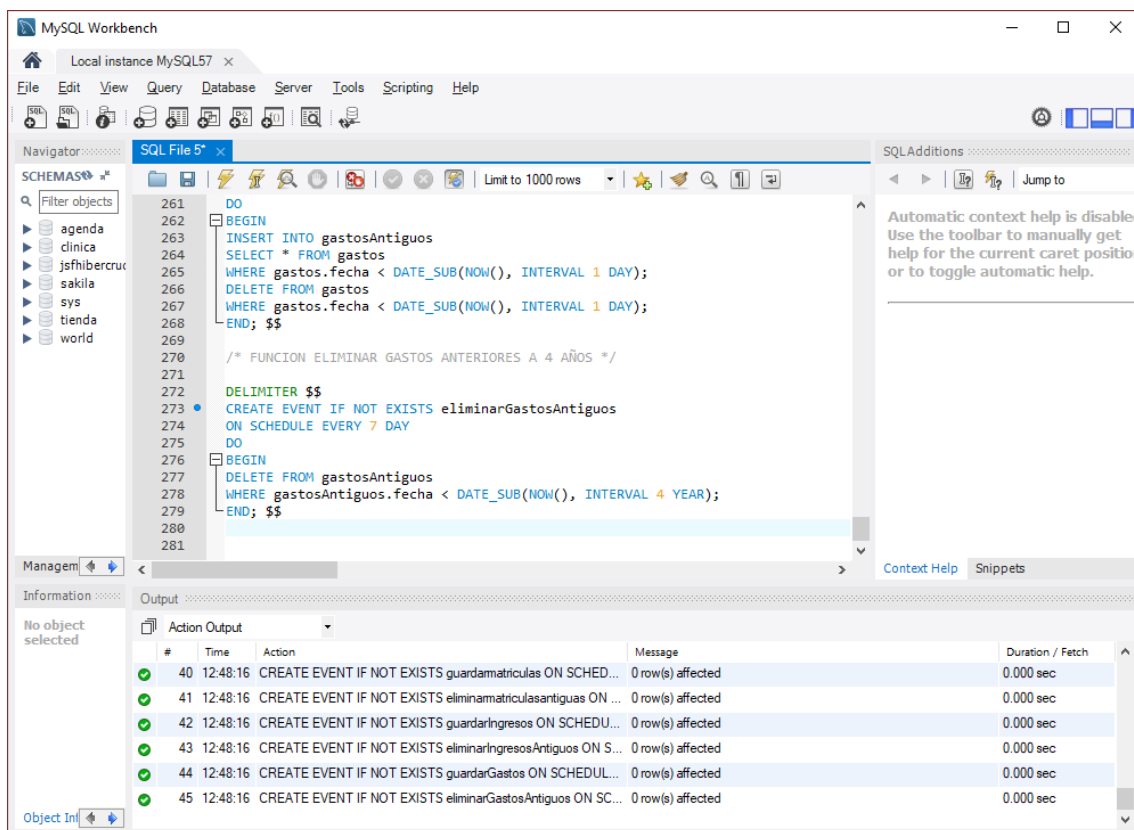
3.14. Automatic schema generation: hibernate.hbm2ddl.auto

https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#annotations

hibernate.hbm2ddl.auto for dummies

<http://www.onlinetutorialspoint.com/hibernate/hbm2ddl-auto-example-hibernate-xml-config.html>

MySQL Workbench es una herramienta visual unificada para arquitectos, desarrolladores y administradores/gestores de bases de datos. Básicamente se trata de una interfaz gráfica más cómoda que una consola.



Con esta interfaz podremos restaurar la base de datos insertando todo el archivo SQL completo cada vez que sea necesario. (Ejemplo en la vista anterior).

Enlace de descarga: <https://dev.mysql.com/downloads/workbench/>

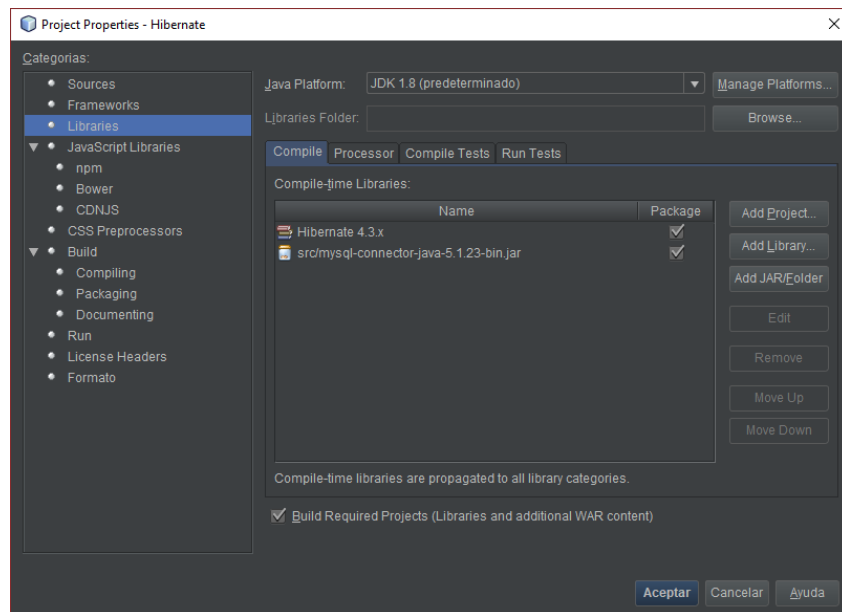
Bootstrap es el framework que hemos utilizado para darle estilo a nuestras vistas, es perfectamente compatible con JSP y además puedes descargarlo o usarlo referenciado desde su página web.

Podrás encontrar toda la documentación, ejemplos y links de descarga en tu página oficial:
<https://getbootstrap.com/>

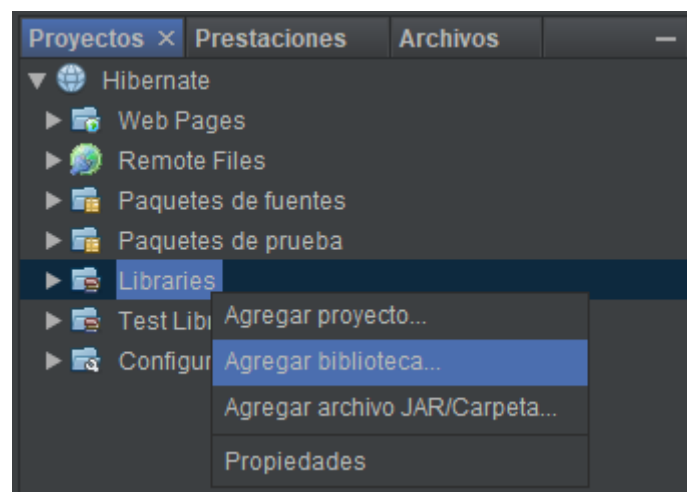
3. Librerías

Si se deseara continuar con el desarrollo del proyecto no serían necesarias la importación de ninguna otra librería más. Ni de las librerías ya mencionadas siempre y cuando usemos el mismo IDE (Netbeans).

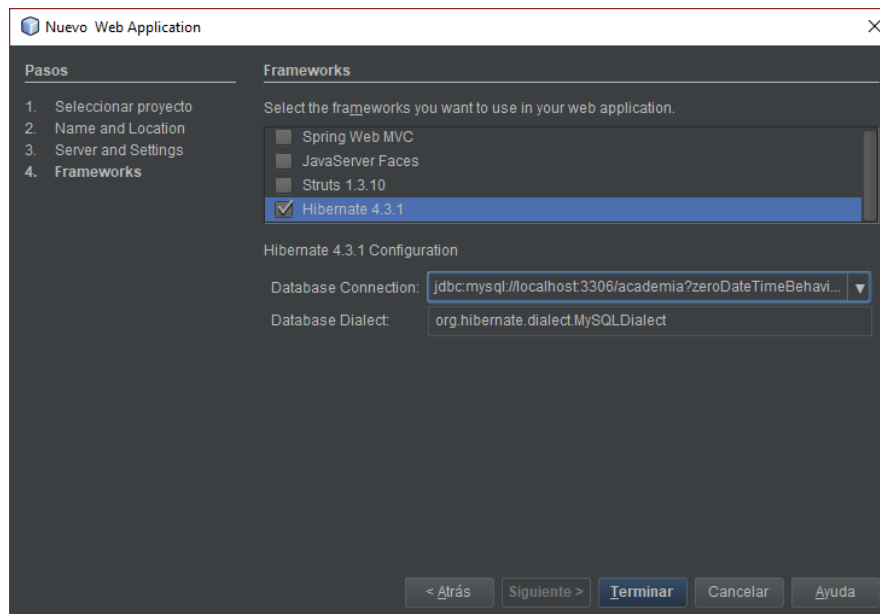
Las librerías utilizadas son las siguientes:



Podemos verlas o añadir nuevas haciendo click derecho sobre el proyecto y seleccionando el último apartado "Propiedades". O también haciendo click derecho sobre la carpeta de Librerías dentro del proyecto (siguiente imagen).



Las librerías utilizadas han sido Hibernate 4.3 integrada en Netbeans.



Y `mysql _conector_5.1` Esta librería ya se encuentra dentro de la carpeta del proyecto dentro del directorio `"/src"`. A la hora de referenciar las librerías es recomendable poner una ruta relativa al proyecto y mantener la estructura del directorio.

Encontraremos el archivo descargable en <https://dev.mysql.com/downloads/connector/j/>

4. Base de datos

La base de datos está instalada en un servidor mysql, para ello podemos descargar XAMPP o MySQL community server.

XAMPP: <https://www.apachefriends.org/es/index.html>

MySQL server: <https://dev.mysql.com/downloads/mysql/>

Hemos usado ambas tecnologías. Recomendaría MySQL server porque permite más opciones de configuración (puedes escoger entre máquina "mixta", de desarrollo o servidor dedicado) y es fácil dejarlo corriendo en segundo plano, aunque para un despliegue rápido utilizar XAMPP nos permitirá probar la aplicación sin tocar ninguna opción de configuración demorándonos el tiempo justo de descargar el programa

EN caso de cualquier duda sobre la instalación de MySQL server puedes consultar este enlace: <https://es.wikihow.com/instalar-un-servidor-de-MySQL-en-una-PC>

Nuestro archivo SQL

```
1 drop database if exists academia;
2 CREATE DATABASE academia;
3 USE academia;
4
5 /* TABLA USER */
6
7 CREATE TABLE user(
8   name varchar(50) NOT NULL,
9   passw varchar(50) NOT NULL,
10  role varchar(5) NOT NULL,
11  PRIMARY KEY (name)
12 );
13
14 /* INSERT USER */
15
16 INSERT INTO user (name, passw, role) VALUES ('user', '04f8996da76387a96981028EE3007569EAF3A635486DDA8211D512C85B9DF8FB', 'user');
17 INSERT INTO user (name, passw, role) VALUES ('admin', '8c6976e5b541041580e908804DEE15DF8167A9C873FC488B816F2AB448A918', 'admin');
18
19 /* TABLA GASTOS */
20
21 CREATE TABLE gastos(
22   idgastos INT NOT NULL AUTO_INCREMENT,
23   importe float(7,2) NOT NULL,
24   fecha timestamp default localtime,
25   concepto varchar(50) NOT NULL,
26   descripcion varchar(200) default NULL,
27   formapago varchar(20) default NULL,
28   PRIMARY KEY (idgastos)
29 );
30
31 /* INSERT GASTOS */
32
33 INSERT INTO gastos (importe, concepto, descripcion, formapago) VALUES (115.95, '2 Mensualidades + Matricula', 'Se le ha rebajado un 15% por apuntarse con un amigo', 'Efectivo' );
34 INSERT INTO gastos (importe, concepto, descripcion, formapago) VALUES (22.95, 'Fotocopiadora Nueva', 'Comprada en Mediamark', 'Efectivo' );
35 INSERT INTO gastos (importe, concepto, descripcion, formapago) VALUES (22.95, 'Agua Enero', 'Inasista', 'Domiciliado' );
36 INSERT INTO gastos (importe, concepto, descripcion, formapago) VALUES (77.15, 'Luz Enero', 'Endesa', 'Domiciliado' );
37 INSERT INTO gastos (importe, concepto, descripcion, formapago) VALUES (675.50, 'Nomina Diciembre Jose Luis', 'Incluidas las 4h extras', 'Transferencia' );
38
39 /* TABLA CLASE */
40
41 CREATE TABLE clase(
42   idClase INT NOT NULL AUTO_INCREMENT,
43   capacidad INT default NULL,
44   nombreclase varchar(50) default NULL,
45   PRIMARY KEY (idClase)
46 );
47
48 /* INSERT CLASE */
49
50
```

Hemos creado las tablas con algunos datos dentro para poder restaurar la base de datos durante el desarrollo.

La base de datos contiene los siguientes eventos:

- Las matriculas se mueven de la tabla “matriculas” a “matriculasantiguas”

- Las matriculas antiguas se borran cuando han pasado más de 4 años de la fecha de la matriculación

- Igual para “ingresos” y para “gastos”

Así mostramos y mantenemos siempre los datos del mes actual y conservamos los anteriores registros. Respecto a este tema la próxima ampliación del programa debería ser la capacidad e mostrar los registros antiguos.

Mueve las matriculas del año pasado a la tabla de matrículas viejas (las matriculas hay que renovarlas cada año)

```
/* FUNCION MOVER MATRICULAS ANTIGUAS */

DELIMITER $$
CREATE EVENT IF NOT EXISTS guardarmatriculas
ON SCHEDULE EVERY 1 YEAR
STARTS "2018-09-01" ENABLE
DO
BEGIN
INSERT INTO matriculasantiguas
SELECT * FROM matricula;
DELETE FROM matricula;
END; $$
```


Borra las matriculas anteriores a 4 años

```
/* FUNCION ELIMINAR MATRICULAS ANTERIORES A 4 AÑOS */

DELIMITER $$
CREATE EVENT IF NOT EXISTS eliminarmatriculasantiguas
ON SCHEDULE EVERY 7 DAY
DO
BEGIN
DELETE FROM matriculasantiguas
WHERE matriculasantiguas.fechmatriculacion < DATE_SUB(NOW(), INTERVAL 4 YEAR);
END; $$
```

Hay que prestar especial atención a las tablas “matriculas” y “matriculasantiguas” pues son las únicas con ON DELETE CASCADE, esto implica que si borramos un alumno se borrarán sus matrículas.

```
134 /* TABLA MATRICULA */
135
136 CREATE TABLE matricula (
137     idMatricula INT NOT NULL AUTO_INCREMENT,
138     DNIALumno varchar(9) NOT NULL,
139     fechmatriculacion timestamp default localtime,
140     precio float (5,2) NOT NULL,
141     PRIMARY KEY (idMatricula),
142     FOREIGN KEY (DNIALumno) REFERENCES alumno (DNIALumno) ON DELETE CASCADE
143 );
144
145 /* INSERT MATRICULA */
146
147 INSERT INTO matricula (DNIALumno, precio) VALUES ( '11112222a',50);
148 INSERT INTO matricula (DNIALumno, precio) VALUES ( '11112222a',25);
149 INSERT INTO matricula (DNIALumno, precio) VALUES ( '00002222p',50);
150
```

5. Vistas (Archivos JSP)

En los archivos JSP encontraremos código java y HTML. A continuación, veremos los más relevantes pues todas han sido creadas de forma sistemática y funcionan prácticamente igual.

Para las vistas no hemos puesto una llamada e un GET o POST directamente en los iconos (ejemplo: el icono borrar) sino que hemos usado modales para todo. Rellenar formularios, actualizar campos, confirmación a la hora de borrar un registro.

Veremos:

- Login
- Inicio
- GruposyClases
- Perfil

LOGIN

```
Salida - Output x GrupoDaoImp.java x AlumnoController.java x Login.jsp x
Source History
22 <body style="background: url('resources/Library-books-009.png')">
23
24
25 try {
26     //Recogemos el http session creado en el login (o uno nuevo si no hemos logeado)
27     HttpSession sessionStatus = request.getSession();
28     //Calculamos el tiempo desde el ultimo uso de la sesion (aqui no lo usaremos todavia)
29     Date timeSinceLastUse = new Date(System.currentTimeMillis() - sessionStatus.getLastAccessedTime());
30     System.out.println("Tiempo desde el ultimo acceso" + timeSinceLastUse.getMinutes() + ":" + timeSinceLastUse.getSeconds());
31     try {
32         System.out.println(sessionStatus.getAttribute("userRole"));
33         //Si el userRole es admin o user(eso significa que tenemos iniciada la sesion de antes) nos reenvia al la ventana de inicio
34         if (sessionStatus.getAttribute("userRole").equals("admin") || sessionStatus.getAttribute("userRole").equals("user")) {
35             response.sendRedirect("Inicio.jsp");
36         }
37     } catch (Exception e2) {
38         //Si no puede recoger el userRole significa que la sesion es nueva o no ha sido creada.
39         //No hacemos nada en ese caso pues estamos en el login
40     }
41     catch (Exception e) {
42         //Si ocurriese algun error nos reenviamos al login nuevamente
43         System.out.println(e.getMessage());
44         response.sendRedirect("Login.jsp");
45     }
46 }
```

Hemos usado el siguiente fragmento de código para comprobar si hay alguna sesión abierta y dirigirnos al inicio en tal caso. No hay más código Java en este archivo.

El único formulario que hay es el necesario para realizar el login.

```
72 <form class="form" role="form" autocomplete="off" id="formLogin" action="UserController" method="POST">
73
74     <div class="form-group">
75         <label for="username">Username</label>
76         <input type="text" class="form-control form-control-lg rounded" name="name" required="true" maxlength="15">
77     </div>
78     <div class="form-group">
79         <label>Password</label>
80         <input type="password" class="form-control form-control-lg rounded" name="password" required="true">
81     </div>
82     <br>
83     <button type="submit" value="Submit" name="logUser" class="btn btn-success btn-lg container-fluid">Login</button>
84 </form>
```

De ahí recogemos los datos del nombre de usuario y contraseña, y comparamos si la contraseña de dicho usuario es la correcta.

INICIO

El inicio (cómo todas las demás páginas) tiene una cabecera parecida a la del login pero funcionando al contrario, si no hay sesión iniciada o esta ha caducado por inactividad la invalidará y te reenviará al inicio.

```

27 <body style="background: url('resources/Library-books-009.png');" >
28     HttpSession sessionStatus;
29     try {
30         //recogemos la sesion
31         sessionStatus = request.getSession();
32         //calculamos el tiempo desde su ultimo uso
33         Date timeSinceLastUse = new Date(System.currentTimeMillis() - sessionStatus.getLastAccessedTime());
34         System.out.println("Tiempo desde el ultimo acceso" + timeSinceLastUse.getMinutes() + ":" + timeSinceLastUse.getSeconds());
35         //si hace mas de 10 min la cerramos y enviamos al login
36         if (timeSinceLastUse.getMinutes() > 10) {
37             System.out.println("Han pasado mas de 10 min desde el ultimo acceso a la session anterior. Invalida la session");
38             session.invalidate();
39             response.sendRedirect("Login.jsp");
40         }
41         //probamos si la session corresponde a algun usuario o administrador
42         try {
43             System.out.println(sessionStatus.getAttribute("userRole"));
44             //si no es asi enviamos al login
45             if (!sessionStatus.getAttribute("userRole").equals("admin") && !sessionStatus.getAttribute("userRole").equals("user")) {
46                 response.sendRedirect("Login.jsp");
47             }
48         } catch (Exception e2) {
49             response.sendRedirect("Login.jsp");
50         }
51     } catch (Exception e) {
52         System.out.println("Redirigiendo de Inicio a Login" + e.getMessage());
53         response.sendRedirect("Login.jsp");
54     }
55     //Declaramos los siguientes elementos que usaremos a continuacion
56     AlumnoDaoImp adi = new AlumnoDaoImp();
57     IngresosDaoImp idi = new IngresosDaoImp();
58     GastosDaoImp gdi = new GastosDaoImp();
59
60 <!-- NAVBAR-->

```

Con el siguiente código evitamos que el usuario con rol “user” tenga acceso al botón de gestión usuarios.

```

99 <li class="nav-item dropdown">
100     <a class="nav-link dropdown-toggle" id="navbarDropdownMenuLink" role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
101         Configuración
102     </a>
103     <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
104         <a class="dropdown-item" href="http://localhost:8080/Hibernate/Perfil.jsp">Perfil</a>
105         <a class="dropdown-item" href="http://localhost:8080/Hibernate/Usuarios.jsp">Usuarios</a>
106         <a class="dropdown-item" href="http://localhost:8080/Hibernate/Ingresos.jsp">Ingresos</a>
107         <a class="dropdown-item" href="http://localhost:8080/Hibernate/Gastos.jsp">Gastos</a>
108     </div>
109 </li>
110 <div class="form-group">
111     <input type="text" value="Buscar" />
112     <button type="button" value="Buscar" />
113 </div>
114

```

Comprobamos el rol de la sesión y solo muestra el botón si el rol es “admin”.

```

141 <h4>INFORMACION GENERAL</h4>
142 <table class="table">
143     <thead>
144         <tr>
145             <th>Alumnos Matriculados</th>
146             <th><%= adi.numAlMat() %></th>
147         </tr>
148     </thead>
149     <tbody>
150         <tr>
151             <th>Alumnos NO matriculados</th>
152             <th><%= adi.numAlNoMat() %></th>
153         </tr>
154         <tr>
155             <th>Ingresos Mensuales</th>
156             <th><%= idi.getImporteTotal() %> €</th>
157         </tr>
158         <tr>
159             <th>Gastos Mensuales</th>
160             <th><%= gdi.getImporteTotal() %> €</th>
161         </tr>
162         <tr>
163             <th>Ganancias Mensuales</th>
164             <th><%= (idi.getImporteTotal() - gdi.getImporteTotal()) %> €</th>
165         </tr>
166     </tbody>
167 </table>

```

Para los CRUD de todas las entidades, listados , generación de botones de forma sistemática por cada elemento, etc. Hemos usado una estructura como la siguiente:

```

360 <table class="table table-sm table-striped mt-2 p-1" style="background: rgba(255, 255, 255, 0.90)"
361 <thead>
362 <tr>
363 <th scope="col">ID Clase</th>
364 <th scope="col">Capacidad</th>
365 <th scope="col">Nombre</th>
366 <th scope="col">Opciones</th>
367 </tr>
368 </thead>
369 <tbody id="myTable2">
370
371 //Con este ciclo listamos todos los elementos de la tabla y generamos sus modales para el update
372 // y los botones para el borrado, confirmaciones, etc
373 for (int i = 0; i < clases.size(); i++) {
374     Clase c = (Clase) clases.get(i);
375
376 <form action="ClassController" method="POST">
377 <tr class="align-middle">
378 <td class="align-middle">{{ c.getIdClase() }}</td>
379 <td class="align-middle">{{ c.getCapacidad() }}</td>
380 <td class="align-middle">{{ c.getNombreClase() }}</td>
381 <select class="idClase" hidden="true">
382 <option value="{{ c.getIdClase() }}" selected="true" hidden="true">{{ c.getIdClase() }}</option>
383 </select>
384 <!-- BOTONES BORRAR Y ACTUALIZAR -->
385 <...8 lines />
386 <!-- MODAL PARA EL ELIMINAR -->
387 <div id="modal fade bd-example-modal-lg" id="delClase{{ c.getIdClase() }}" tabindex="-1" role="dialog" aria-labelledby="exampleModalLabel" aria-hidden="true">
388 <...26 lines />
389 </div>
390 <!-- MODAL UPDATE CLASES -->
391 <div id="modal fade" id="claseUpdate{{ c.getIdClase() }}" tabindex="-1" role="dialog" aria-labelledby="exampleModalLabel" aria-hidden="true">
392 <...30 lines />
393 </div>
394 </form>
395
396
397 }
398 </tbody>
399 </table>

```

PERFIL

De aquí solamente comentar que hemos filtrado los usuarios que mostramos para que solo se muestre el usuario que está registrado en ese momento en la sesión

```

138 <tbody id="mytable">
139
140 UserDaoImp udi = new UserDaoImp();
141 List users = udi.listUsers();
142 for (int i = 0; i < users.size(); i++) {
143     User u = (User) users.get(i);
144     if (u.getName().equals(sessionStatus.getAttribute("userName"))) {
145

```

6. Código

6.1. Metodología Y Creación del proyecto

Para realizar el proyecto hemos usado el patrón estructural DAO y MVC

El DAO consiste en separar las implementaciones de los objetos y métodos que acceden a los datos y cargan las entidades del resto de los componentes. Así hemos creado la clase DAO como interfaz para los métodos de forma sistemática una tras otra y después los EntityDaoImp que contienen los métodos usados para acceder a los objetos.

A continuación, otra explicación y breve ejemplo de cómo funciona el patrón.

La idea de este patrón es sencilla. En primer lugar, debemos hacernos las clases que representan nuestros datos. Por ejemplo, podemos hacer una clase *Persona* con los datos de la persona y los métodos *set()* y *get()* correspondientes.

Luego hacemos una interface. Esta interface tiene que tener los métodos necesarios para obtener y almacenar Personas. Esta interface no debe tener nada que la relacione con una base de datos ni cualquier otra cosa específica del medio de almacenamiento que vayamos a usar, es decir, ningún parámetro debería ser una Connection, ni un nombre de fichero, etc. Por ejemplo, puede ser algo así

```
public interface InterfaceDAO {
    public List<Persona> getPersonas();
    public Persona getPersonaPorNombre (String nombre);
    ...
    public void salvaPersona (Persona persona);
    public void modificaPersona (Persona persona);
    ...
    public void borraPersonaPorNombre (String nombre);
    ...
}
```

y todos los métodos con todas las variantes que necesitemos en nuestra aplicación.

Con esto deberíamos construir nuestra aplicación, usando la clase *Persona* y usando la *InterfaceDAO* para obtener y modificar Personas.

Por ejemplo, podríamos hacer implementaciones diferentes según la base de datos *PersonaMySQLDAO_imp* y *PersonaOracleDAO_imp*.

En nuestro caso no es necesario debido a que usamos Hibernate, pero este patrón aunque si es muy usado en Hibernate no es un patrón estructural exclusivo de este framework.

Normalmente (no en nuestro caso) el patrón DAO se completa con algún tipo de Factoría: Una clase a la que al pedirle la *InterfaceDAO* decide cuál de las implementaciones instanciar y la devuelve.

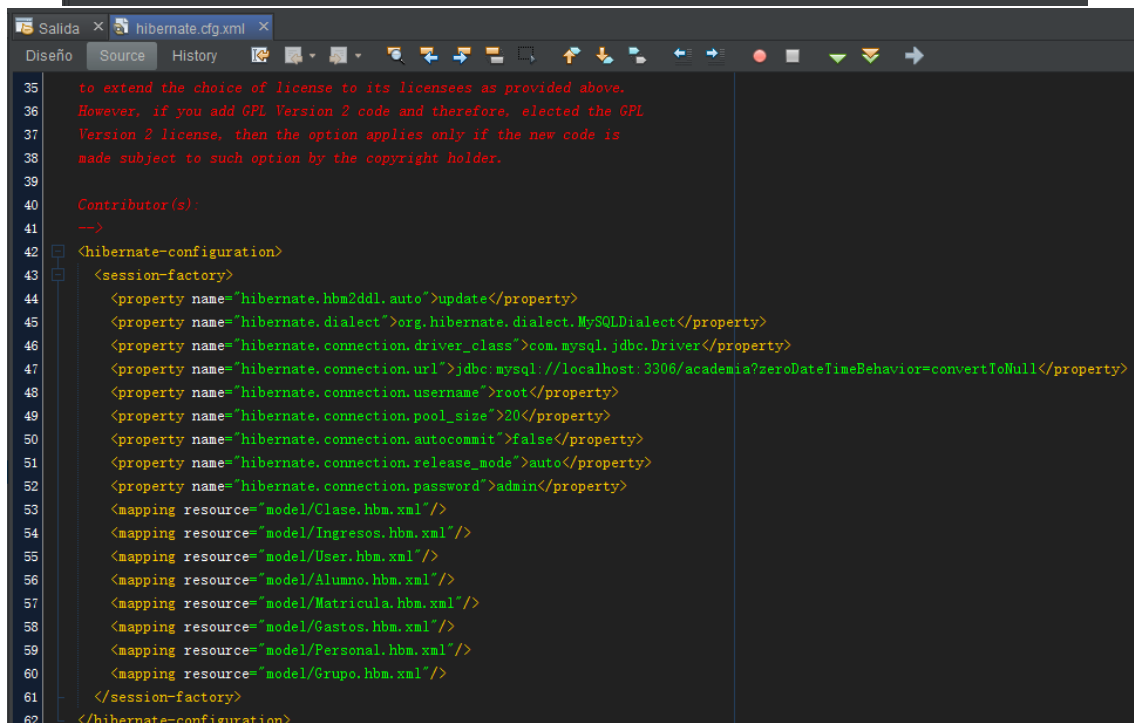
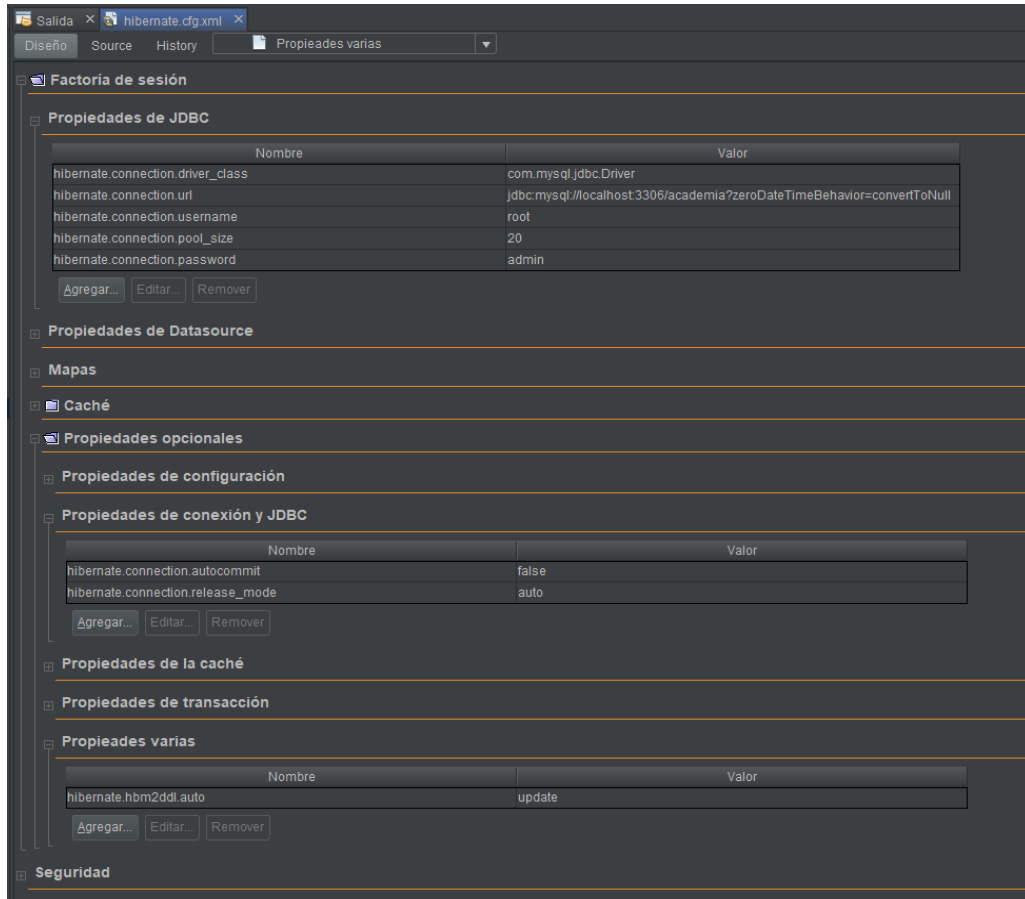
```
public class FactoriaDAO{
    public static InterfaceDAO getDAO() {
        if (baseDatos.equals("oracle")
            return new ImplementacionDAOOracle (parametrosConexionOracle);
        else if (baseDatos.equals("MySQL")
            return new ImplemetancionDAOMySQL (parametrosConexionMySQL);
        }
    }
}
```

Estructuralmente también hemos separado las vistas y controladores (excepto por el listar que no lo controlamos directamente desde los controladores, sino que está implícito en los ficheros JSP)

Hemos realizado un controlador por entidad en vez de un controlador por vista pero hemos modificado alguno de los controladores para que funcionen de una forma u otra según el objeto *HttpSession* vigente en ese momento.

6.2. Configuración

Ahora veremos las opciones de configuración de Hibernate. Hay muchas posibilidades, se pueden poner manualmente en el fichero hibernate.cfg.xml o seleccionarla mediante una interfaz que nos proporciona Netbeans.



Vamos a ver las opciones que hemos seleccionado:

hibernate.hbm2ddl.auto Valida o exporta automáticamente el esquema DDL(Data Definition Language) a la base de datos cuando se crea el SessionFactory. Con create-drop, el esquema de la base de datos se eliminará cuando SessionFactory se cierre explícitamente.

Entonces la lista de opciones posibles es:

validate: valida el esquema, no realiza cambios en la base de datos.

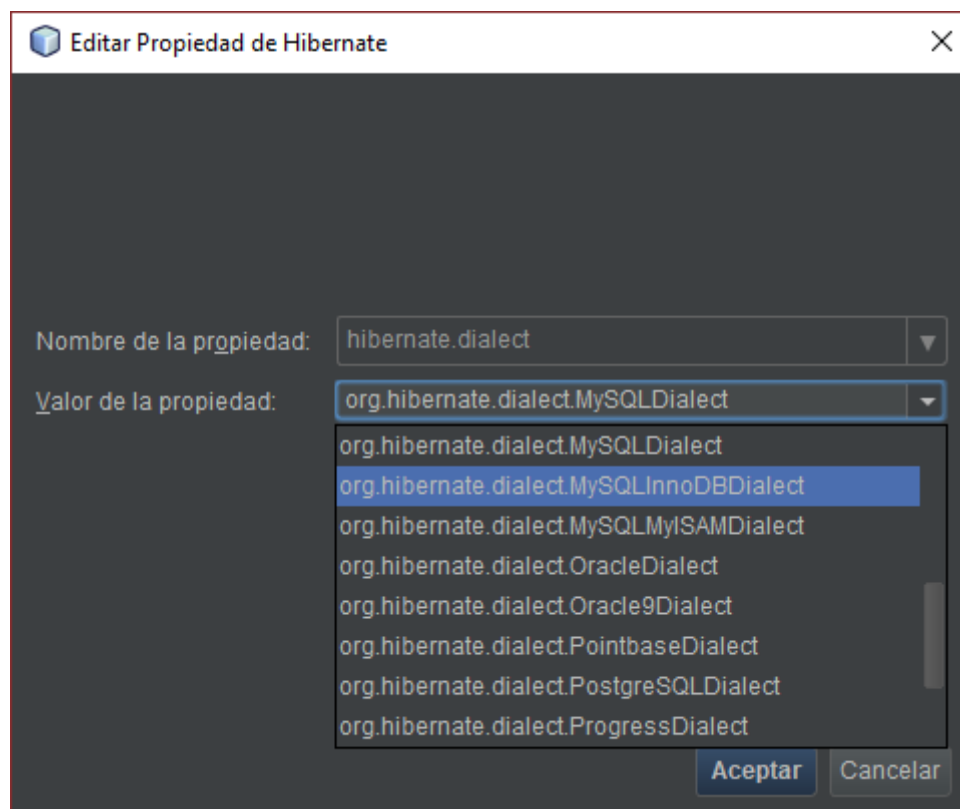
update: actualizar el esquema.

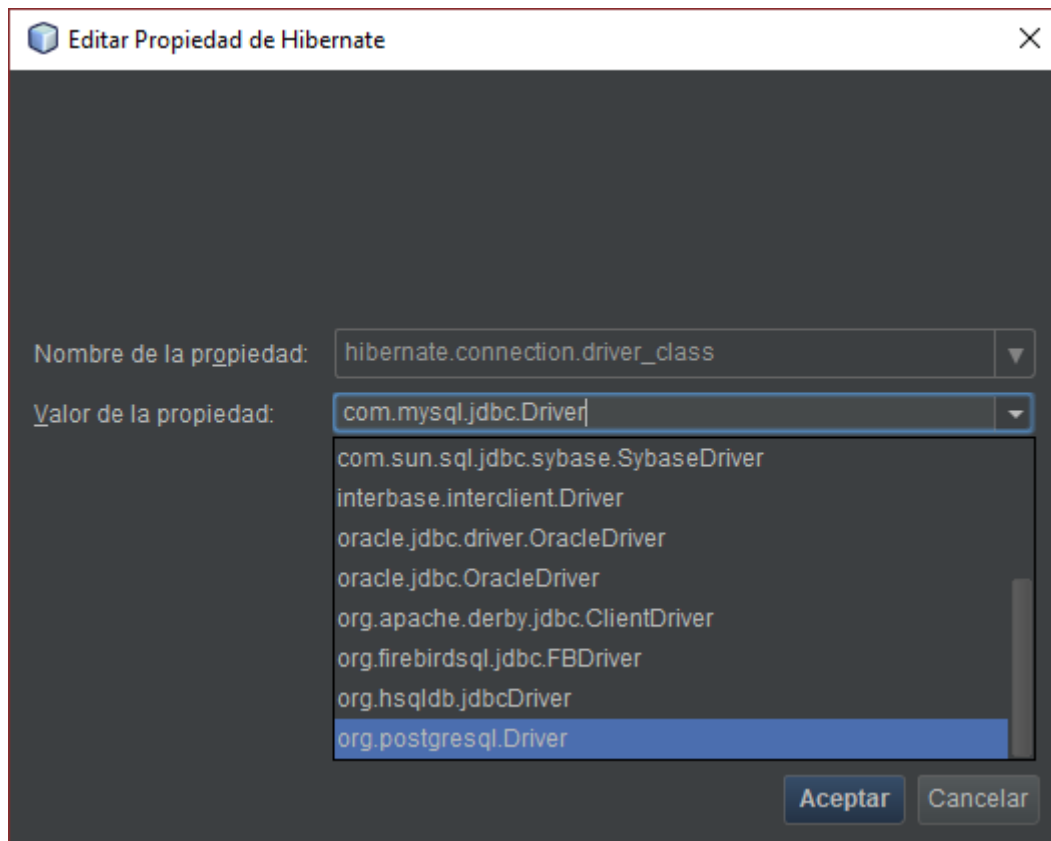
create: crea el esquema, destruyendo datos previos.

create-drop: crea y luego borra el esquema cuando SessionFactory se cierra explícitamente, generalmente cuando se detiene la aplicación.

Escogimos update porque estuvimos probando realizando cambios en la base de datos y en las entidades y tuvimos problemas así que así nos aseguramos que el esquema de la base de datos coincidiría con las entidades. También podríamos haber escogido validate con este propósito pero con update obtuvimos un resultado más “flexible” ya que podíamos realizar pequeños cambios sin tener que estar cambiándolo todo continuamente.

Hibernate Dialect. Seleccionamos MySQL porque es el que corresponde a la base de datos que utilizamos. Aquí vemos otras opciones que deberemos contemplar si vamos a cambiar de base de datos:





```
<property name="hibernate.connection.url">
```

```
jdbc:mysql://localhost:3306/academia?zeroDateTimeBehavior=convertToNull
```

```
</property>
```

Seleccionamos la dirección de la base de datos.

```
<property name="hibernate.connection.username">root</property>
```

El nombre de usuario para la conexión (podemos crear los usuarios que queramos para la base de datos)

```
<property name="hibernate.connection.pool_size">20</property>
```

Un límite máximo de conexiones de 20. Hibernate ya hace un control del pool de conexiones automático aunque si se desea un control más exhaustivo se puede usar la librería C3P0. Aquí tenemos documentación detallada sobre esta herramienta:

https://developer.jboss.org/wiki/HowToConfigureTheC3P0ConnectionPool?_sscc=t

```
<property name="hibernate.connection.autocommit">>false</property>
```

Impide el cerrado automatico de las conexiones del pool de hibernate. Tenemos en el código cerradas las sesiones cada vez que hacemos una transacción así que como el proyecto es pequeño y no está pensado para gran cantidad de conexiones podemos dejar esto el false.

```
<property name="hibernate.connection.release_mode">auto</property>
```

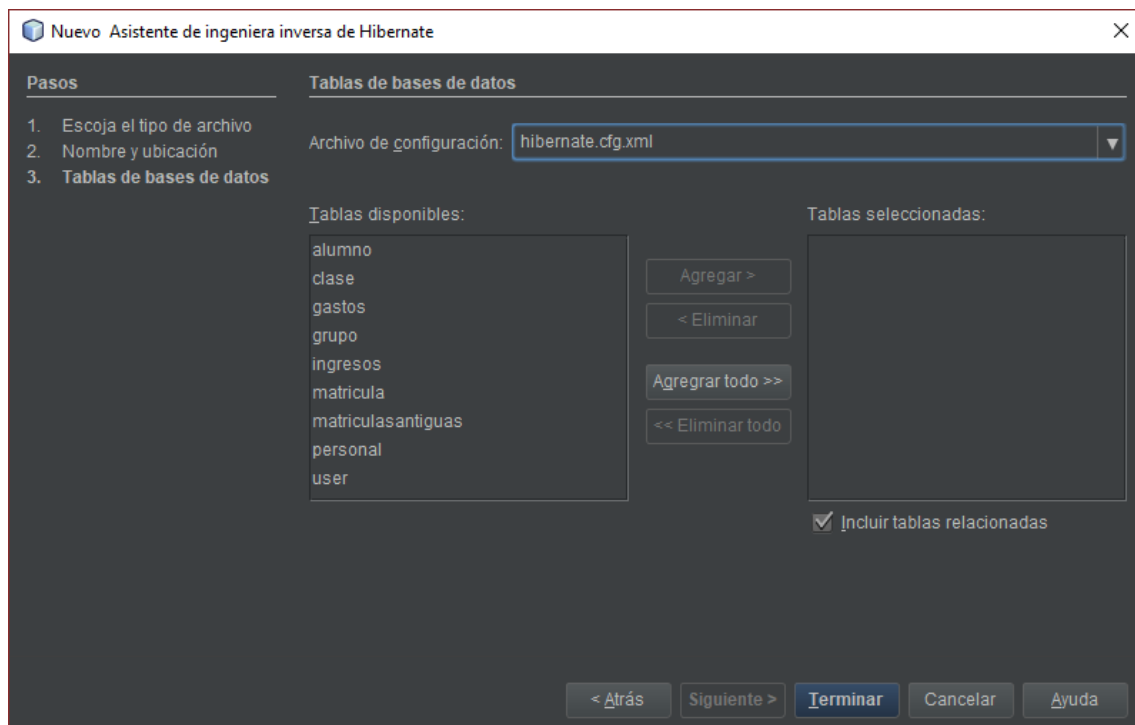
Esta configuración solo afecta a las Sesiones devueltas desde `SessionFactory.openSession`. Para las sesiones obtenidas a través de `SessionFactory.getCurrentSession`, la implementación `CurrentSessionContext` configurada para su uso controla el modo de liberación de la conexión para esas sesiones.

```
<property name="hibernate.connection.password">admin</property>
```

Es la contraseña que usamos para la base de datos.

6.3. POJO's

Los pojos (o las entidades) las hemos generado automáticamente con Hibernate ORM. Una vez creado el `hibernate.cfg.xml` generamos el archivo `hibernate.reveng.xml`



Agregamos todas las entidades que deseemos y automáticamente nos generará los POJO's necesarios.

Vamos a ver un ejemplo:

Todos los pojos tienen la misma estructura. Son clases con los atributos necesarios según hayamos creado la base de datos. El Set que contienen corresponde a los grupos que dan clase en esa aula. Contienen un constructor vacío, uno con los atributos "NOT NULL" y otro con todos los atributos de la tabla.

```
Source History
1 package model;
2 // Generated 15-feb-2018 17:54:17 by Hibernate Tools 4.3.1
3
4
5 import java.util.HashSet;
6 import java.util.Set;
7
8 /**
9  * Clase generated by hbm2java
10  */
11 public class Clase implements java.io.Serializable {
12
13
14     private Integer idClase;
15     private Integer capacidad;
16     private String nombreclase;
17     private Set grupos = new HashSet(0);
18
19     public Clase() {
20     }
21
22     public Clase(Integer capacidad, String nombreclase, Set grupos) {
23         this.capacidad = capacidad;
24         this.nombreclase = nombreclase;
25         this.grupos = grupos;
26     }
27
28     public Integer getIdClase() {
29         return this.idClase;
30     }
31
32     public void setIdClase(Integer idClase) {
33         this.idClase = idClase;
34     }
35
36     public Integer getCapacidad() {
37         return this.capacidad;
38     }
39
40     public void setCapacidad(Integer capacidad) {
41         this.capacidad = capacidad;
42     }
43
44     public String getNombreclase() {
45         return this.nombreclase;
46     }
47 }
```

6.4. Servicios DAO

Aquí vamos a ver las interfaces de los DAO y los DAO_implementations, por ultimo también veremos la clase HibernateUtil.

Antes de realizar las implementaciones de los servicios hemos creado las interfaces con los métodos que a continuación implementaremos.

```
1 package services.dao;
2
3 import java.util.List;
4 import model.Alumno;
5
6 public interface AlumnoDao {
7     //Interfaz que se usará en AlumnoDaoImp
8     public String addAlumno(Alumno alumno);
9     public String updateAlumno(Alumno alumno);
10    public String deleteAlumno(Alumno alumno);
11    public Alumno searchAlumno(String dni);
12    public List<Alumno> listAllAlumnos();
13    public List<Alumno> listAlumnosMat();
14    public List<Alumno> listAlumnosNoMat();
15    public int numAlMat();
16    public int numAlNoMat();
17 }
```

El proceso que seguimos para interactuar con la base de datos siempre es el mismo.

Abrimos la sesión usando el método `getSessionFactory().openSession()`. Podemos decir que la clase `hibernateUtil` es una factoría de sesiones. Las sesiones son creadas a partir de la configuración que hemos establecido en Hibernate en el `hibernate.cfg.xml`

Creamos una transacción y la abrimos con `.begin()`

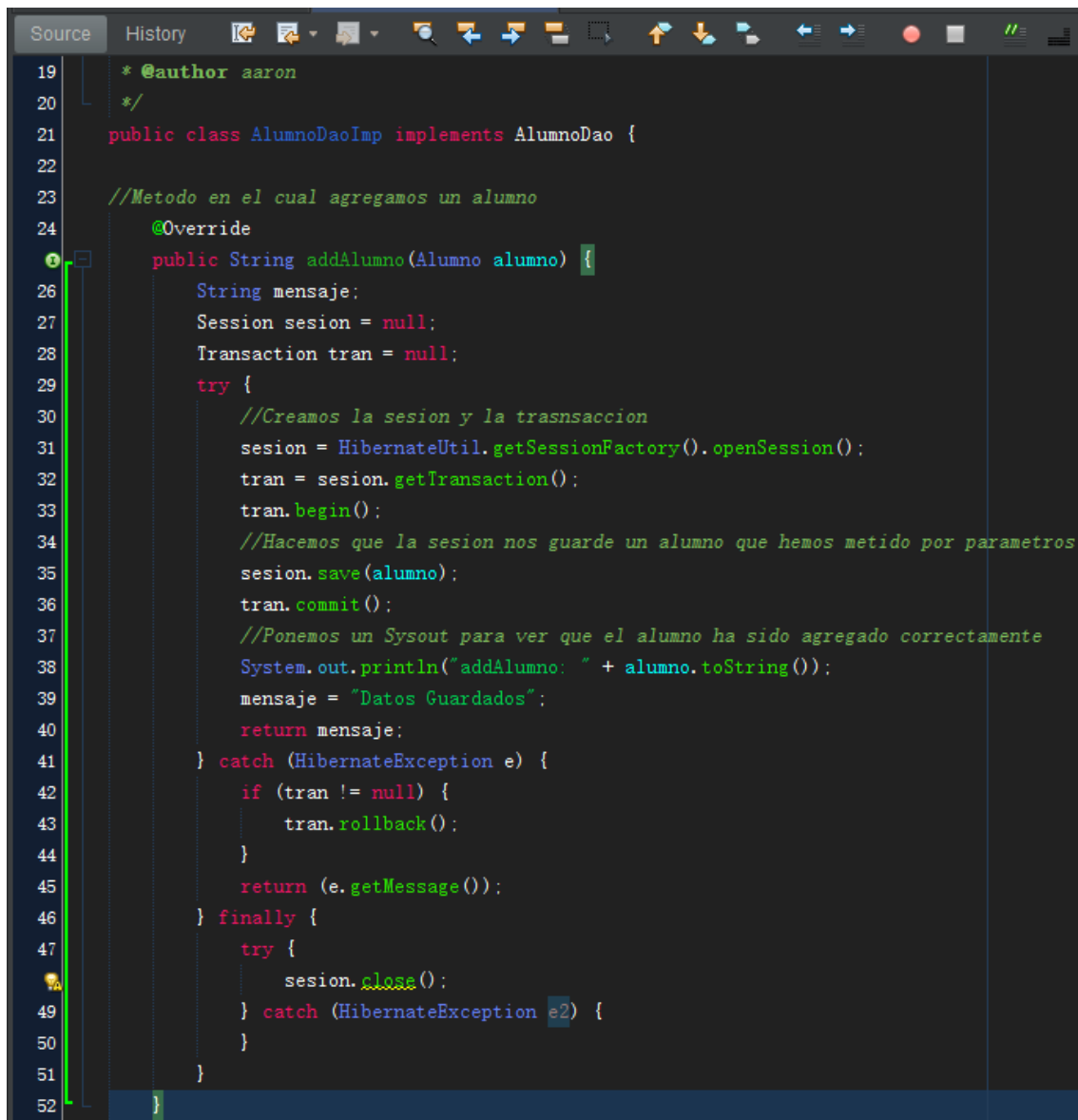
Guardamos el alumno con `sesion.save()`

Cerramos la transacción

Usamos `transaction.rollback()` en el “catch” por si algo sale mal para volver al punto anterior

`rollback()`: revierte una transacción al comienzo de la transacción, o a un punto de rescate dentro de la transacción. También libera los recursos que posee la transacción.

Y cerramos la sesión. A continuación veremos un ejemplo:



```
19  * @author aaron
20  */
21  public class AlumnoDaoImp implements AlumnoDao {
22
23      //Metodo en el cual agregamos un alumno
24      @Override
25      public String addAlumno(Alumno alumno) {
26          String mensaje;
27          Session session = null;
28          Transaction tran = null;
29          try {
30              //Creamos la sesion y la transaccion
31              session = HibernateUtil.getSessionFactory().openSession();
32              tran = session.getTransaction();
33              tran.begin();
34              //Hacemos que la sesion nos guarde un alumno que hemos metido por parametros
35              session.save(alumno);
36              tran.commit();
37              //Ponemos un Sysout para ver que el alumno ha sido agregado correctamente
38              System.out.println("addAlumno: " + alumno.toString());
39              mensaje = "Datos Guardados";
40              return mensaje;
41          } catch (HibernateException e) {
42              if (tran != null) {
43                  tran.rollback();
44              }
45              return (e.getMessage());
46          } finally {
47              try {
48                  session.close();
49              } catch (HibernateException e2) {
50              }
51          }
52      }
53  }
```

Todos los métodos CREATE/UPDATE/DELETE siguen el mismo procedimiento. La diferencia es que algunos usan sesión.save(), sesión.update() o sesión.delete()

A continuación, veremos cómo funcionan las queries o consultas HQL (Hibernate Query Language).

En la siguiente foto podemos ver el método searchAlumno(String dni) en este caso recogemos un alumno utilizando la clase Query. Para crear una consulta lo hacemos usando Session.greateQuery("String de la consulta").

El lenguaje de consulta de hibernate hace las consulta a las clases y los objetos, NO realiza la consulta a la base de datos directamente asi que en vez de poner los nombre de las tablas escribimos los nombres de las clases.

Es muy parecido a SQL. No es "case sensitive" excepto para el nombre de las clases.

O sea "FrOm Alumno" nos devolverá todos los alumno igual que "FROM Alumno".

```

96 //Metodo por el cual buscamos un alumno ingresando su PK (primary key)
97 @Override
98 public Alumno searchAlumno(String dni) {
99     Session session = null;
100     Transaction tran = null;
101     Alumno alumno = null;
102     try {
103         session = HibernateUtil.getSessionFactory().openSession();
104         tran = session.getTransaction();
105         tran.begin();
106         //Creamos una query en la cual buscamos un alumno por su DNI
107         Query query = session.createQuery("FROM Alumno WHERE dnialumno='" + dni + "'");
108         //Guardamos el alumno que nos devuelve la query
109         alumno = (Alumno) query.uniqueResult();
110         tran.commit();
111     } catch (Exception e) {
112         if (tran != null) {
113             tran.rollback();
114         }
115         e.printStackTrace();
116     } finally {
117         try {
118             session.close();
119         } catch (Exception e) {
120         }
121     }
122     //Hacemos que nos retorne el alumno seleccionado
123     return alumno;
124 }

```

En este método no debería ser necesario usar `tran.rollback()` pues no realizamos ninguna operación excepto la consulta así que no realizamos cambios en la base de datos pero por sistema lo hemos puesto.

Por ultimo el archivo `HibernateUtil`, nuestra factoría de sesiones.

```

Source History
1 package services.daoImp;
2
3 import org.hibernate.HibernateException;
4 import org.hibernate.cfg.AnnotationConfiguration;
5 import org.hibernate.SessionFactory;
6
7 public class HibernateUtil {
8
9     private static final SessionFactory sessionFactory;
10
11     static {
12         try {
13             // Create the SessionFactory from standard (hibernate.cfg.xml) config file.
14             //Implementacion del patrón Metodo Factoria (o metodo fábrica según la traduccion)
15             sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
16         } catch (HibernateException ex) {
17             // Log the exception.
18             System.err.println("Initial SessionFactory creation failed." + ex);
19             throw new ExceptionInInitializerError(ex);
20         }
21     }
22
23     public static SessionFactory getSessionFactory() {
24         return sessionFactory;
25     }
26 }

```

6.5. Controladores

Un objeto `ServletRequest/Response` (al revés) proporciona datos que incluyen nombre de parámetro y valores, atributos y una secuencia de entrada. Las interfaces que amplían `ServletRequest` pueden proporcionar datos adicionales específicos del protocolo (por ejemplo, `HttpServletRequest` proporciona datos HTTP).

Resumen de los métodos de <code>RequestDispatcher</code>	
<code>void forward (solicitud ServletRequest, respuesta ServletResponse)</code>	Reenvía una solicitud de un servlet a otro recurso (servlet, archivo JSP o archivo HTML) en el servidor.
<code>void include (solicitud ServletRequest, respuesta ServletResponse)</code>	Incluye el contenido de un recurso (servlet, página JSP, archivo HTML) en la respuesta.

<https://docs.oracle.com/javaee/6/api/javax/servlet/RequestDispatcher.html>

```
1 package controller;
2
3 import java.io.IOException;
4 import java.text.ParseException;
5 import java.text.SimpleDateFormat;
6 import java.util.Date;
7 import java.util.logging.Level;
8 import java.util.logging.Logger;
9 import javax.servlet.RequestDispatcher;
10 import javax.servlet.ServletException;
11 import javax.servlet.http.HttpServlet;
12 import javax.servlet.http.HttpServletRequest;
13 import javax.servlet.http.HttpServletResponse;
14 import model.Alumno;
15 import model.Grupo;
16 import services.daoImp.AlumnoDaoImp;
17 import services.daoImp.GrupoDaoImp;
18
19 public class AlumnoController extends HttpServlet {
20
21     //Al implementar la interfaz HttpServlet adquirimos las clases doGet y doPost, con estas clases haremos peticiones o recogeremos datos
22     Alumno alumno = new Alumno();
23     SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
24     AlumnoDaoImp alunodaoimp = new AlumnoDaoImp();
25     GrupoDaoImp grupodaoimp = new GrupoDaoImp();
26
27     /**
28      * @param request //nuestra peticion
29      * @param response // la respuesta
30      * @throws ServletException
31      * @throws IOException
32      */
33     @Override
34     protected void doGet(HttpServletRequest request, HttpServletResponse response)
35         throws ServletException, IOException {
36         //Cada if(request.getParameter("addAlumno") != null) salta si el string corresponde
37         //al "name" del "submit" que realizamos
38         if (request.getParameter("addAlumno") != null) {
39             //Recogemos los diferentes datos de los inputs
40             String dniAlumno = request.getParameter("DNIAlumno");
41             int grup = Integer.parseInt(request.getParameter("IDGrupo"));
42             Grupo grupo = grupodaoimp.searchGrupo(grup);
43             String nombre = request.getParameter("nombre");
```

```

    } catch (NumberFormatException ex) {
        telefonofijo = 000000000;
    }
    String nombrefamiliar = request.getParameter("nombrefamiliar");
    String apellidofamiliar = request.getParameter("apellidofamiliar");
    Integer telefonofamiliar;
    try {
        telefonofamiliar = Integer.parseInt(request.getParameter("telefonofamiliar"));
    } catch (Exception e) {
        telefonofamiliar = 000000000;
    }
    String emailfamiliar = request.getParameter("emailfamiliar");

    //Asignamos todos los datos al alumno que tenemos creado arriba
    alumno.setDnialumno(dnialumno);
    alumno.setGrupo(grupo);
    alumno.setNombre(nombre);
    alumno.setApellidos(apellidos);
    alumno.setFechnac(fechnac);
    alumno.setNacionalidad(nacionalidad);
    alumno.setEmail(email);
    alumno.setTelefonopersonal(telefonopersonal);
    alumno.setDireccion(direccion);
    alumno.setCodpostal(codpostal);
    alumno.setTelefonofijo(telefonofijo);
    alumno.setNombrefamiliar(nombrefamiliar);
    alumno.setApellidofamiliar(apellidofamiliar);
    alumno.setTelefonofamiliar(telefonofamiliar);
    alumno.setEmailfamiliar(emailfamiliar);
    /*
    La clase RequestDispatcher define un objeto que recibe solicitudes del cliente y las envía
    a cualquier recurso (como un servlet, archivo HTML o archivo JSP) en el servidor. El contenedor de
    servlets crea el objeto RequestDispatcher, que se utiliza como un contenedor alrededor
    de un recurso de servidor ubicado en una ruta particular o dado por un nombre particular.
    */
    alumnodaoimp.addAlumno(alumno);
    //Recogemos el gestor de solicitudes correspondiente a nuestra pagina
    RequestDispatcher rd = request.getRequestDispatcher("Alumnos.jsp");
    //lanzamos la peticion y la respuesta (implica una recarga de la pagina)
    rd.forward(request, response);

```

Las dos últimas clases que vamos a aver son las de UserController y Cifrador.


```
UserController.java x
Source History
28
29 @Override
30 protected void doGet(HttpServletRequest request, HttpServletResponse response)
31     throws ServletException, IOException {
32     //Añadimos usuario tal y como lo hemos hecho con el resto de las entidades
33     if (request.getParameter("addUser") != null) {
34         String name = request.getParameter("name");
35         String passwd = request.getParameter("passwd");
36         String role = request.getParameter("role");
37
38         user.setName(name);
39         //Ciframos la contraseña antes de añadirla
40         user.setPassw(getSHA_256_Digest(passwd));
41         user.setRole(role);
42         userdaoimp.addUser(user);
43
44         RequestDispatcher rd = request.getRequestDispatcher("Usuarios.jsp");
45         rd.forward(request, response);
46     }
47     //Invalidamos el objeto HttpSession y volvemos al login
48     if (request.getParameter("salir") != null) {
49         HttpSession mySession = request.getSession();
50         mySession.invalidate();
51         response.sendRedirect("Login.jsp");
52     }
53 }
54
55 @Override
56 protected void doPost(HttpServletRequest request, HttpServletResponse response)
57     throws ServletException, IOException {
58     //Borramos al gual que el resto de la entidades
59     if (request.getParameter("deleteUser") != null) {
60         String name = request.getParameter("namedel");
61         User us = userdaoimp.searchUser(name);
62         userdaoimp.deleteUser(us);
63         RequestDispatcher rd = request.getRequestDispatcher("Usuarios.jsp");
64         rd.forward(request, response);
65     }
66 }
```

Usamos un algoritmo SHA-256 para cifrar las contraseñas y compararlas.

Como podemos ver el comportamiento de UserController varía en función del estado en el que se encuentre el objeto de HttpSession vigente en ese momento. Así los usuarios no pueden cambiarse el rol para obtener privilegios de administrador.

NOTA: Para la próxima actualización sería muy recomendable implementar este sistema en todos los controladores. Porque aunque la sesión caduca y te devuelve al login, si has dejado una página abierta se puede lanzar una última petición antes de que sea necesario recargar la página que es cuando el sistema implementado comprueba la sesión y te redirige a la página correspondiente.

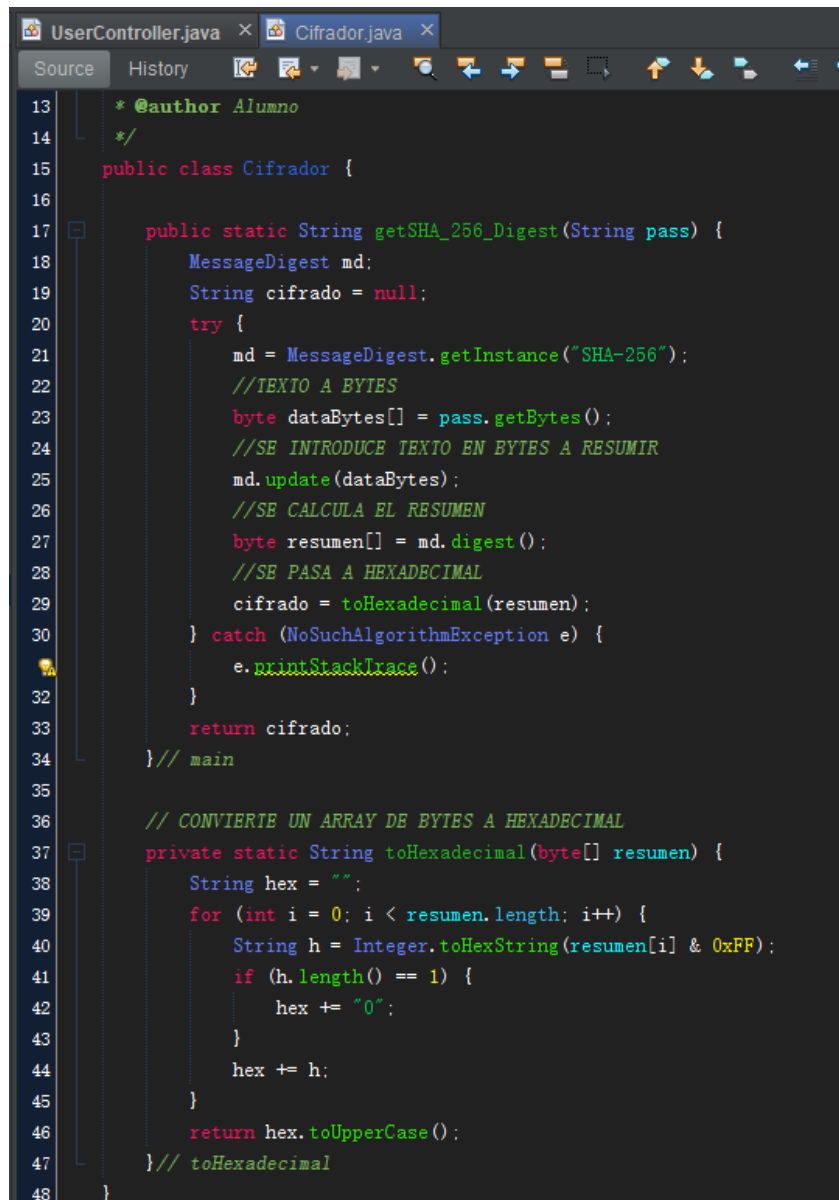
```

//Actualizamos al igual que el resto de las entidades excepto por ...
if (request.getParameter("updateUser") != null) {
    String name = request.getParameter("upname");
    String passwd = request.getParameter("uppasswd");
    String role = request.getParameter("uprole");
    HttpSession mySession = request.getSession();
    user.setName(name);
    user.setPassw(getSHA_256_Digest(passwd));
    user.setRole(role);

    //Si el rol del usuario logeado en este momento seguira manteniendose como user
    //aunque modifique el html para poder introducir un rol diferente al lanzar la peticion
    if (mySession.getAttribute("userRole").equals("user")) {
        user.setRole("user");
        userdaoimp.updateUser(user);
        RequestDispatcher rd = request.getRequestDispatcher("Perfil.jsp");
        rd.forward(request, response);
    } else {
        userdaoimp.updateUser(user);
        RequestDispatcher rd = request.getRequestDispatcher("Usuarios.jsp");
        rd.forward(request, response);
    }
}
}

```

Cifrador.java



```

UserController.java x Cifrador.java x
Source History
13  * @author Alumno
14  */
15  public class Cifrador {
16
17      public static String getSHA_256_Digest(String pass) {
18          MessageDigest md;
19          String cifrado = null;
20          try {
21              md = MessageDigest.getInstance("SHA-256");
22              //TEXTO A BYTES
23              byte dataBytes[] = pass.getBytes();
24              //SE INTRODUCE TEXTO EN BYTES A RESUMIR
25              md.update(dataBytes);
26              //SE CALCULA EL RESUMEN
27              byte resumen[] = md.digest();
28              //SE PASA A HEXADECIMAL
29              cifrado = toHexadecimal(resumen);
30          } catch (NoSuchAlgorithmException e) {
31              e.printStackTrace();
32          }
33          return cifrado;
34      } // main
35
36      // CONVIERTE UN ARRAY DE BYTES A HEXADECIMAL
37      private static String toHexadecimal(byte[] resumen) {
38          String hex = "";
39          for (int i = 0; i < resumen.length; i++) {
40              String h = Integer.toHexString(resumen[i] & 0xFF);
41              if (h.length() == 1) {
42                  hex += "0";
43              }
44              hex += h;
45          }
46          return hex.toUpperCase();
47      } // toHexadecimal
48  }

```