Modificación de datos

Una de las funcionalidades principales del trabajo fin de máster planteaba la posibilidad de realizar las operaciones básicas de eliminación, modificación e inserción de datos en las tablas de la base de datos.

Este capítulo trata sobre este tema, indicándose como se han abordado las distintas cuestiones técnicas y las soluciones propuestas.

# Acceso a las operaciones

El primer problema planteado consiste en definir una forma que permita realizar las operaciones de modificación y eliminación de un determinado registro de una tabla y cómo definir la inserción de un registro concreto en una tabla.

Se ha realizado un sistema común para poder efectuar las tres operaciones básicas sobre datos que afectan a una serie de pantallas y clases que se indican a continuación.

## Tabla.jsp

Los enlaces que permiten acceder a las operaciones sobre datos se definen en esta pantalla, que es la que muestra los datos de una determinada tabla en forma de tabla organizada en filas (registros) y columnas (campos).

Se modifica, pues, el código que distribuye los datos de esta forma para incluir una nueva columna (la primera) que contendrá una serie de links hacia las acciones struts que permitirán realizar las operaciones con los datos.

No todas las tablas pueden albergar esta nueva columna con los links. Solo sobre aquellas tablas que tengan una primary key se podrán realizar las operaciones de eliminación, modificación o inserción de datos.

Para determinar esta cuestión, la acción strut correspondiente debe proporcionar un método que devuelva true, si dispone de PK o false en caso contrario:

El segundo bucle recorre, como habíamos visto en el capítulo XXX, los campos del registro:

<s:iterator value="#record" status="stat">  
Si estamos en la primera columna, y es una tabla con PK (metodoLink = true)  
 <s:if test="#stat.first == true && metodoLink">  
La fila de cabecera no tiene nada  
<s:if test="#status.first == true">  
 <th><s:text name=""/></th>  
</s:if>  
y el resto de filas contiene una celda con una lista que apunta a cada una de las funcionalidades sobre los datos: eliminar, editar o insertar:

<td>

<ul class="bqY ocultar">

Para cada opción de la lista hay que definir la url a la que se deberá redirigir la request. Para eliminar:

<s:url action="borrar" var="urlBru" includeParams="get" escapeAmp="false"/>

incluyendo el contenido de la primera columna de la lista (contendrá los datos de la PK):

<s:set var="url"><s:property value="urlBru"/>&<s:property/></s:set>

Y definiendo el item de la lista que se visualizará

<li class="bqX" data-tooltip="Eliminar">

<s:a href="%{url}" class="acciones bru">&nbsp;&nbsp;</s:a>

</li>

Para la edición:

<s:url action="editar" var="urlEdt" includeParams="get" escapeAmp="false"/>

<s:set var="urle"><s:property value="urlEdt"/>&<s:property/></s:set>

Y para la adición, sin necesidad de incluir la PK:

<s:url action="insertar" var="urlIns" includeParams="get" escapeAmp="false"/>

La primera columna de la tabla, inicialmente estará oculta y solo será visible la celda correspondiente a la fila sobre la que el usuario pase el ratón. De esta manera se evita presentar al usuario un grupo grande opciones (uno por cada registro) cuando solo se va a realizar la operación sobre una de las filas en cada momento.

Para conseguir que los links se muestren o se oculten, debemos, utilizando jquery, asignar o suprimir el estilo css correspondiente agrupado bajo una clase concreta:

<tr onmouseover='$(this).children("td").children("ul.bqY").removeClass("ocultar")'

onmouseout='$(this).children("td").children("ul.bqY").addClass("ocultar")'>

Según se indica, cuando el usuario pasa sobre una fila, se remueve la clase css *ocultar* con lo que se hacen visibles los datos para esa fila.

Igualmente, cuando el usuario abandona el foco de la fila con el ratón, se vuelve a asignar la clase *ocultar* a la celda con lo que su contenido queda oculto al usuario.

## general.css

El fichero con los estilos css aplicables a la aplicación es ampliado para conseguir dos efectos correspondientes a las funcionalidades de modificación de datos.

En primer lugar se define una clase ocultar que permite mostrar u ocultar al usuario los links de cada registro:

.ocultar {

display: none !important;

}

En segundo lugar, en cada item de la lista html (ul.li) se define un formato que, entre otras cosas, determina la imagen que se mostrará en el correspondiente link.

Para eliminación:

.bru {

background-image: url(images/delete\_black\_20dp.png);

background-position: center;

background-repeat: no-repeat;

background-size: 20px;

}

Para edición:

.edt {

background-image: url(images/edit\_black\_20dp.png);

background-position: center;

background-repeat: no-repeat;

background-size: 20px;

}

Y para inserción:

.ins {

background-image: url(images/insert\_black\_20.png);

background-position: center;

background-repeat: no-repeat;

background-size: 20px;

}

## TablasAction.java

Se realizan modificaciones en distintos métodos y se construyen nuevos métodos. En relación con la fase inicial de establecimiento de links en la pantalla de consulta de una tabla, como hemos indicado, se define un nuevo método (*metodoLink*) que permitirá saber si una tabla o vista concreta es susceptible de admitir operaciones sobre sus datos.

Se modifica el método *consulta()* para incluir la definición de esta característica de la siguiente forma:

metodoLink = false; // Será true cuando exista una primary key en la tabla consultada

y mediante una consulta al método *getPrimeryKeys* del databaseMetaData de la conexión averiguamos si la tabla o vista tiene PK, y, de paso, averiguamos qué campos la componen:

List<String> pkList = new ArrayList<>();

rs2 = connectionImpl.getConnection().getMetaData().getPrimaryKeys("", TABLE\_SCHEM, TABLE\_NAME);

while (rs2.next())

pkList.add(rs2.getString("COLUMN\_NAME"));

if (pkList.size()!=0) metodoLink = true;

Finalmente, se obtenía la lista correspondiente al resultset de la tabla o vista convenientemente filtrada. La función que realizaba esta operación es modificada y ampliada para permitir añadir como primera columna de la estructura, los datos necesarios para realizar el link correspondiente:

listInfo = getListInfo(rs, metodoLink, pkList.toArray(linkParametros), "pkArgumentos", "pkValores");

Esta función se sobrecargó con dos cadenas de caracteres, *pkArgumentos* y *pkValores* que, junto con el parámetro *metodoLink*, harán que se construye la primera columna de forma adecuada:

## MenuAction.java

Se modifica la función

List<List> getListInfo(ResultSet resultSet, boolean metodoLink, String[] linkParametros, String argumentos, String valores)

para dar cabida a la funcionalidad indicada: que en la primera columna se incluya la información necesaria para establecer el link. En el caso de la modificación de datos, este link deberá incluir la información de la primary key de la tabla consultada:

Se añade un elemento vacío como primer elemento del registro

if (metodoLink) record.add(urlParametro);

Recorriendo cada campo del registro

for (int j = 1; j < i; j++) {

Si debemos establecer el link

if (metodoLink) {

Para cada parámetro a comprobar

for (String linkParametro : linkParametros)

Si coincide con la columna en la que estamos

if (linkParametro.equals(resultSet.getMetaData().getColumnName(j))) {

Y argumentos es nulo (opción de ejecución anterior), se conforma el parámetro con su propio nombre

if (argumentos==null)

urlParametro = urlParametro + "&" + linkParametro + "=" + resultSet.getObject(j);

Pero si argumentos no es nulo, contiene el nombre que habrá que darle (típicamente "pkArgumentos") y los valores se agruparán bajo el nombre recibido (típicamente "pkValores"):

else {

urlParametro = urlParametro + "&" + argumentos + "=" + linkParametro;

urlParametro = urlParametro + "&" + valores + "=" + resultSet.getObject(j);

}

...

record.add(resultSet.getObject(j));

Finalmente se sustituye el elemento primero de la lista por el calculado (quitando el primer caracter que sería el "&" inicial:

if (metodoLink) record.set(0,urlParametro.substring(1));

y se añade a la lista definitiva:

lista.add(record);

# Eliminación

La eliminación de un registro es una de las operaciones básicas sobre los datos en una tabla. Como requisito único se plantea la necesidad de que la tabla disponga de una clave primaria. Para obtener la clave primaria se utiliza el método getPrimaryKey del DatabaseMetaData del objeto Connection como se ha explicado anteriormente.

Para realizar la eliminación en la aplicación deberemos realizar la correspondiente request pasando los parámetros oportunos, que obligatoriamente deberán ser:

TABLE\_SCHEM=*nombre del esquema (opcional)*

TABLE\_NAME=*nombre de la tabla*

TABLE\_TYPE=*tipo de la tabla, vista, sinónimo (opcional)*

De 1 a n veces:

pkArgumentos=*nombre del campo k de la PK*

pkValores=*valor del campo k de la PK*

para valores de k de 1 a n.

Este link se ha formado en la llamada previa a la acción *TablasAction.consulta()* y se visualiza bajo el icono de papelera en *Tablas.jsp* como se ha explicado en el apartado anterior.

La eliminación se realiza mediante la ejecución de una instrucción *DELETE* en la correspondiente acción que recoge los parámetros, los valida y la ejecuta como se indica a continuación

## struts.xml

El fichero de configuración de struts debe contener el mapeo de la acción de borrado:

<action name="borrar" class="es.ubu.alu.mydatabasejc.actions.TablasAction" method="borrar">

Además, teniendo en cuenta el resultado de la acción, podemos realizar un encadenamiento de la acción hacia la acción *consulta***:**

<result name="consulta" type="chain">

<param name="actionName">consulta</param>

</result>

o bien una redirección hacia la presentación de la lista de tablas:

<result name="tablas" type="redirectAction">

<param name="actionName">resultset</param>

<param name="namespace">/DatabaseMetaData</param>

<param name="metodo">getTables</param>

<param name="parametros">rO0ABXVyABJbTGphdmEubGFuZy5DbGFzczurFteuy81amQIAAHhwAAAABHZyABBqYXZhLmxhbmcuU3RyaW5noPCkOHo7s0ICAAB4cHEAfgADcQB-AAN2cgATW0xqYXZhLmxhbmcuU3RyaW5nO63SVufpHXtHAgAAeHA</param>

</result>

## TablasAction.java

Paquete es.ubu.alu.mydatabasejc.actions

En esta clase se desarrolla el nuevo método que realiza la funcionalidad de borrado de un registro:

* **borrar():** Este método comienza con una comprobación de que se reciben los parámetros adecuados:

if (TABLE\_NAME == null || "".equals(TABLE\_NAME)) {

sesion.put("ACTION\_ERROR", "Faltan.esquema.o.nombre.de.tabla");

return "tablas";

}

if (pkArgumentos == null || pkArgumentos.length == 0) {

sesion.put("ACTION\_ERROR", "Falta.primary.key");

return "consulta";

}

La puesta en sesión del mensaje de error permitirá recoger dicho mensaje cuando se realiza la redirección correspondiente, puesto que la redirección o encadenamiento de acciones elimina los errores y mensajes de la pila. Debemos, pues, almacenarlos en la sesión del usuario para que puedan ser recuperados en los métodos oportunos, como se verá más adelante.

Tras la validación, se realiza la eliminación del registro:

Se crea un objeto de la clase *SQLCommand* con la conexión a la base de datos:

try {

SQLCommand sqlComand = new SQLCommand(connectionImpl);

Se ejecuta el método *executeUpdate* con los parámetros adecuados que devolverá 0 si no hay registros borrados, o <> 0 si algún registro ha sido borrado

if (sqlComand.executeUpdate(TABLE\_SCHEM, TABLE\_NAME, mapaCompleto, SQLCommand.OPERACION\_DELETE, pkArgumentos, pkValores, null, null)==0) {

Se añade el mensaje correspondiente a la sesión del usuario

sesion.put(ACTION\_ERROR, "No.se.ha.eliminado.ningún.registro");

...

sesion.put(ACTION\_MESSAGE, "El.registro.ha.sido.eliminado");

En caso de error, también se guardará el mensaje en la sesión para ser presentado con posterioridad

} catch (SQLCommandException ex) {

sesion.put(ACTION\_ERROR, ex.getLocalizedMessage());

Finalmente, se vuelve a presentar la consulta de la tabla

return "consulta";

* **consulta()**: Este método se ve completado por una recuperación de los mensajes del espacio de sesión del usuario para ser presentados en la correspondiente pantalla:

Recoge los mensajes de error previos y los establece en esta acción:

if (sesion.get(ACTION\_ERROR)!=null) {

addActionError(getText((String)sesion.get(ACTION\_ERROR)));

sesion.remove(ACTION\_ERROR);

}

y recoge los mensajes informativos previos y los establece en esta acción

if (sesion.get(ACTION\_MESSAGE)!=null) {

addActionMessage(getText((String)sesion.get(ACTION\_MESSAGE)));

sesion.remove(ACTION\_MESSAGE);

}

## SQLCommand.java

Paquete es.ubu.alu.mydatabasejc.jdbc

Se crea esta clase para manejar todos los comandos propios de la ejecución de sentencias SQL contra un objeto de base de datos. La clase dispone de los siguientes atributos:

* **public ConnectionImpl connectionImpl:** Es el objeto que representa la conexión con la base de datos
* **public String cadenaWhere**: cadena utilizada para la parte where de la sentencia SQL
* **public List<Object> listaParametrosWhere**: Lista de valores que deben asignarse a cada uno de los campos que intervienen en la parte where de la sentencia. Por ejemplo, si una cláusula where es del tipo "mes = ?", *cadenaWhere* contendrá esta cadena de texto y *listaParametrosWhere* tendrá el valor del mes que se desea filtrar (un valor entre 1 y 12, por ejemplo)
* **public String cadenaSet**: Es el mismo concepto que el indicado para la cláusula where pero aplicado a la parte SET del comando SQL UPDATE
* **public List<Object> listaParametrosSet**: Es el mismo concepto que el indicado para *listaParametrosWhere* pero aplicado a los valores que se deben establecer para los campos en un comando SQL UPDATE
* **public String sql**: Contendrá la cadena SQL a ser ejecutada
* **public String cadenaInsert**: Contiene la lista de campos separada por comas que serán incluidos en el comando SQL INSERT.
* **public String cadenaInsert2**: Contiene la lista de interrogaciones análogas a la lista *cadenaInsert* que formarán la parte VALUES del comando SQL INSERT correspondiente
* **public List<Object> listaParametrosInsert**: Contiene la lista de valores que serán aplicados a los correspondientes marcadores definidos en *cadenaInsert2*.

Además, en relación con la eliminación de registros, necesitamos los siguientes métodos y constructores de clase:

* **SQLCommand(ConnectionImpl connectionImpl)**: El constructor inicializa los correspondientes atributos de la clase.
* **int executeUpdate(String esquema, String tabla, Map<String, Integer[]> mapa, int operacion, String[] pkArgumentos, String[] pkValores, String[] campos, String[] valores)**

Es el método que, propiamente dicho, intenta eliminar el registro. Esquema y tabla reciben el nombre del esquema y la tabla. *operacion*, puede recibir el valor *OPERACION\_DELETE* que establecerá la variable *sql* como "DELETE %s%s %s" cuyos parámetros son sustituidos por *getEsquema(esquema)*, *tabla* y *cadenaWhere*, respectivamente.

La asignación de parámetros a la sentencia sql se realiza mediante la funcionalidad de un *PreparedStatement*:

ps = connectionImpl.getConnection().prepareStatement(sql);

se asignan los parámetros

int parameterIndex = 1;

la parte where

for (Object o : listaParametrosWhere) {

ps.setObject(parameterIndex++, o);

}

return ps.executeUpdate();

El establecimiento de *cadenaWhere* y de *listaParametrosWhere* se realiza al comienzo del método con la llamada:

setSQLPartList(mapa, pkArgumentos, pkValores, OPERACION\_WHERE);

* **private void setSQLPartList(Map<String, Integer[]> mapa, String[] pkArgumentos, String[] pkValores, int tipo)**: Este método es el encargado de establecer correctamente tanto *cadenaWhere* como *listaParametrosWhere*. Como se utiliza en otras circunstancias, se recibe una cadena tipo que determina cómo trabajará el método. En el caso de la eliminación, el tipo es *OPERACION\_WHERE*:

if (tipo == OPERACION\_WHERE) {

Para cada cadena recibida en *pkArgumentos*

for (String columna : pkArgumentos) {

Se obtiene el correspondiente valor de *pkValores* del tipo adecuado calculado por la función getValor pasándole el tipo necesario desde el mapa recibido como argumento:

Object o = getValor(mapa.get(columna)[0], pkValores[n++]);

Si no es el primer campo, se enlazan con la partícula *AND*:

if (listaParametrosWhere.size() != 0) {

cadenaWhere = cadenaWhere + "AND ";

Y si es el primer campo, se incia la cadena con "where":

} else {

cadenaWhere = "where ";

}

Se completa la cadena añadiendo el nombre del campo y el valor posible:

cadenaWhere = cadenaWhere + columna + " = ? ";

Y se añade el valor a la lista de parámetros:

listaParametrosWhere.add(o);

Ante cualquier error se genera una excepción:

catch (SQLException ex) {

throw new SQLCommandException(this, ex);

## SQLCommandException.java

Paquete es.ubu.alu.mydatabasejc.exceptions

Para manejar los errores producidos en la clase SQLCommand anterior, se crea esta clase que dispone del siguiente constructor:

* **public SQLCommandException(SQLCommand aThis, SQLException ex)**. El constructor inicia la clase de la que hereda (*Throwable*)

super(ex);

e imprime en el log la información correspondiente al comando que provocó el error (error propiamente dicho, comando SQL, valores SET y valores WHERE):

logger.error("SQL error: {}\nSQL: {}\nvalores SET: {}\nvalores WHERE: {}\nvalores INSERT: {}",

ex.getMessage(), aThis.sql, aThis.listaParametrosSet,

aThis.listaParametrosWhere, aThis.listaParametrosInsert);

# Modificación

El proceso para la modificación de un registro, así como el de la inserción, y a diferencia de la eliminación de un registro, requiere de la presentación al usuario de un formulario en el que el usuario introducir los datos correspondientes, para modificar o para insertar el registro. En el caso concreto de la modificación, no solo necesitamos obtener del usuario los valores nuevos para los campos, además necesitamos conocer los campos de la clave primaria y sus valores previos a la posible modificación.

Comenzamos analizando la petición que se realiza para una modificación que, al igual que ocurría en el proceso de eliminación, debe indicar el esquema, la tabla y los campos y valores que conforman e identifican la primary key del registro que se desea modificar. Por ejemplo:

tablas/editar.action?TABLE\_SCHEM=JHA&TABLE\_NAME=EDIFICIO&TABLE\_TYPE=TABLE&pkArgumentos=NOMBRE&pkValores=SALAMANCA&pkArgumentos=NOMBRE&pkValores=ALAMEDILLA

## struts.xml

Como en todas las acciones, hay que definir el método y el destino de la acción:

<action name="editar" class="es.ubu.alu.mydatabasejc.actions.TablasAction" method="editar">

<result>/Form.jsp</result>

<result name="error">/Form.jsp</result>

</action>

Para la acción de guardar una modificación:

<action name="editarGuardar" class="es.ubu.alu.mydatabasejc.actions.TablasAction" method="editarGuardar">

<result name="consulta" type="redirectAction">

<param name="actionName">consulta</param>

<param name="TABLE\_SCHEM">%{TABLE\_SCHEM}</param>

<param name="TABLE\_NAME">%{TABLE\_NAME}</param>

<param name="TABLE\_TYPE">%{TABLE\_TYPE}</param>

</result>

</action>

## TablasAction.java

El primer método que se utiliza es:

* **editar()**: Tras la correspondiente validación, similar a la indicada en el método *borrar()*, se obtiene el resultset con los datos del registro a modificar iniciando un objeto de la clase SQLCommand

SQLCommand sqlComand = new SQLCommand(connectionImpl);

y se resuelve el resultset mediante el método *executeQuery* correspondiente:

ResultSet rs = sqlComand.executeQuery(TABLE\_SCHEM, TABLE\_NAME,

pkArgumentos, pkValores,

ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_READ\_ONLY);

pasándole esquema, tabla, campos y valores de la primary key y dos flags que permiten hacer un scroll sobre el resultset (lo explicaremos a continuación) y un flag que indica que el resultset será de solo lectura.

Finalmente, se obtiene la lista con los datos mediante el método *getListInfoReverse* a partir del resultset obtenido:

listInfo = getListInfoReverse(rs);

Como último paso, se carga el mapa de campos editables de tal forma que aparecerán en la página web deshabilitados aquellos campos que no sean editables:

mapaEditables = sqlCommand.getMap(mapaCompleto, SQLCommand.ISDEFINITELYWRITABLE + SQLCommand.ISWRITABLE);

Si no hay error, el retorno llevará a presentar al usuario el formulario con los datos actuales del registro:

return SUCCESS;

* **getPkValores(int i)**: obtiene el elemento i del array *pkValores*:

return pkValores == null ? null : (i>=pkValores.length ? null : pkValores[i]);

* **editarGuardar()**: Este método ejecuta la actualización de los datos en la base de datos. En primer lugar valida la información recibida.

En primer lugar la tabla:

if (TABLE\_NAME == null || "".equals(TABLE\_NAME)) {

sesion.put("ACTION\_ERROR", "Faltan.esquema.o.nombre.de.tabla");

return "tablas";

}

Luego los datos de la clave principal:

if (pkArgumentos == null || pkArgumentos.length == 0) {

sesion.put("ACTION\_ERROR", "Falta.primary.key");

return "consulta";

}

Finalmente los campos y valores a actualizar:

if (formCampos == null || formCampos.length == 0 || formValores == null || formValores.length == 0) {

sesion.put("ACTION\_ERROR", "Faltan.datos.para.hacer.la.operacion");

return "consulta";

}

Una vez validada, se inicia el SQLCommand:

SQLCommand sqlComand = new SQLCommand(connectionImpl);

y se ejecuta el executeUpdate correspondiente:

if (sqlComand.executeUpdate(TABLE\_SCHEM, TABLE\_NAME, mapaCompleto, OPERACION\_UPDATE, pkArgumentos, pkValores, formCampos, formValores)==0) {

Si no se actualiza ningún registro, mensaje:

sesion.put(ACTION\_MESSAGE, "No.se.ha.actualizado.ningún.registro");

Si se actualiza algún registro, mensaje

} else {

sesion.put(ACTION\_MESSAGE, "El.registro.ha.sido.actualizado");

}

y si se produce algún error, mensaje

sesion.put(ACTION\_ERROR, ex.getLocalizedMessage());

y en cualquier caso, retorno a consulta

return "consulta";

* **boolean isEditDisabled(String campo)**: Esta función devuelve true si la edición no es posible para el campo indicado o falso si es posible modificar el valor del campo. Para cada campo editable, si coincide con el campo indicado, se retorna false (edición habitlitada)

for (String campoWritable : mapaEditable.keySet()) {

if (campoWritable.equalsIgnoreCase(campo))

return false;

}

Si no hay coincidencia se retorna true (edición deshabitilatada)

return true;

## Form.jsp

Carpeta /

Esta pantalla presenta los datos disponibles bajo listInfo en formato formulario. En principio, listInfo solo debería tener un único registro, aunque este hecho no es controlado y si dispone de más registros y el resultado sería, la lista con los nombres de campo en la primera columna y a continuación un conjunto de campos text de html presentando los valores de cada registro por columnas.

Para realizar correctamente la distribución de los datos por columnas, en vez de por filas, la lista listInfo debe estar adecuadamente construida de tal forma que en la primera fila tenga el nombre y valores del primer campo de cada registro del resultset, en la segunda fila tenga el nombre y valores del segundo campo de cada registro del resultset y así sucesivamente hasta completar los c campos del resultset asociado.

La distribución se realiza dentro de un formulario html:

<s:form name="filtro" method="POST" theme="simple" id="filtro" action="editarGuardar">

que indica que la acción a realizar tras la request es *editarGuardar* que se vió en el punto anterior

Los campos y valores se organizan en forma de tabla html

<table class="filtro">

donde cada fila se corresponde con una fila de la lista listInfo

<s:iterator value="listInfo" var="record" status="status">

y a cada fila se le corresponde un fila de la tabla html

<tr>

Dentro de cada fila, un segundo iterador recorre los componentes de cada fila

<s:iterator value="#record" status="stat">

Si es el primer elemento, contiene el nombre del campo del resultset y se presenta tal cual acompañado de un campo oculto de nombre *formCampos* con el nombre del campo. También almacenamos el nombre del campo en la variable *nombreCampo*:

<s:set var="campo"><s:property escapeHtml="false"/></s:set>

<s:if test="#stat.first == true">

<td>

<s:text name="%{campo}"/>:

<s:hidden name="formCampos" value="%{campo}"/>

<s:set var="nombreCampo" value="%{campo}"/>

</td>

</s:if>

Si no se trata del primer registro, se contiene el valor del resultset para ese campo que se presentarán dentro de un campo formulario con el nombre *formValores.* También se establece la posibilidad de edición del campo mediante la función *isEditDisabled* comentada en el punto anterior:

<td><s:textfield name="formValores" value="%{campo}" disabled="%{isEditDisabled(#nombreCampo)}"/></td>

Finalmente, se añaden al formulario el esquema y la tabla

<s:hidden name="TABLE\_SCHEM"/>

<s:hidden name="TABLE\_NAME"/>

<s:hidden name="TABLE\_TYPE"/>

Junto con los campos ocultos que conforman los campos y valores de la clave principal para lo que se utiliza la función *getPkValores* comentada en el punto anterior:

<s:iterator value="pkArgumentos" status="stat">

<s:set var="pk"><s:property/></s:set>

<s:hidden name="pkArgumentos" value="%{pk}"/>

<s:hidden name="pkValores" value="%{getPkValores(#stat.count-1)}"/>

</s:iterator>

Y un botón de envío del formulario:

<s:submit name="guardar" value="Guardar"/>

## SQLCommand.java

En el método TablasAction.editar se hace referencia al método SQLCommand.getMap que devuelve un mapa con los campos de un mapa completo que contiene todos los nombres de campo más una estructura de doble entero en la que el primero es el tipo de datos del campo y el segundo es la característica del campo.

* **Map<String, Integer> getMap(Map<String, Integer[]> mapa, int tipo)**: La función construye un mapa de campos y sus tipos según se adapten o no al *tipoCampo* indicado.

Map<String, Integer> retorno = new HashMap<String, Integer>();

para cada columna del mapa completo

for (String columna : mapa.keySet()) {

si el segundo entero combinado con el tipo en binario es distinto de cero, el campo tiene la característica solicitada y se añade al mapa retorno:

if ((mapa.get(columna)[1] & tipo) != 0)

retorno.put(columna, mapa.get(columna)[0]);

Finalmente se retorna el mapa obtenido:

return retorno;

Como se ha indicado en el método *TablasAction.editarGuardar()* se realiza una llamada al método

sqlComand.executeUpdate(TABLE\_SCHEM, TABLE\_NAME, mapaCompleto, SQLCommmand.OPERACION\_UPDATE, pkArgumentos, pkValores, formCampos, formValores)

al que se le pasan el nombre del esquema y tabla, la cadena "update", los campos y valores que conforman la primary key (*pkArgumentos* y *pkValores*) y los campos y valores del registro que se deben actualizar (*formCampos* y *formvalores*).

* **executeUpdate**: El método comienza con el establecimiento de las variables globales *cadenaSet, cadenaWhere, listaParametrosSet y listaParametrosWhere*:

setSQLPartList(mapa, pkArgumentos, pkValores, campos, valores);

Se establece la variable con el comando SQL:

sql = String.format("UPDATE %s%s SET %s %s",

getEsquema(esquema),

tabla,

cadenaSet,

cadenaWhere);

Y se asigna el comando a un PreparedStatement asignandole posteriormente los valores de los campos a modificar y de la primary key:

ps = connectionImpl.getConnection().prepareStatement(sql);

se asignan los parámetros

int parameterIndex = 1;

en primer lugar la parte update, no habrá parámetros si no hay parte update

for (Object o : listaParametrosSet) {

ps.setObject(parameterIndex++, o);

}

y luego la parte where

for (Object o : listaParametrosWhere) {

ps.setObject(parameterIndex++, o);

}

Al final se retorna el código *executeUpdate* del *preparedStatement*:

return ps.executeUpdate();

Si hay algún error se genera una excepción *SQLCommandException* como ya se comentó en el caso de una operación de borrado de datos.

* **setSQLPartList(Map<String, Integer[]> mapa, String[] pkArgumentos, String[] pkValores, String[] campos, String[] valores)**: Este método establece los valores de las variables globales *cadenaSet*, *cadenaWhere*, *listaParametrosSet* y *listaParametrosWhere*.

En primer lugar establece la *cadenaSet* y *listaParametrosSet* obteniendo el valor mediante la función getValor y el tipo de datos sql obtenido del mapa de columnas:

for (String columna : campos) {

Object o = getValor(mapa.get(columna)[0], valores[n++]);

if (listaParametrosSet.size() != 0) {

cadenaSet = cadenaSet + ", ";

}

cadenaSet = cadenaSet + columna + " = ? ";

listaParametrosSet.add(o);

}

A continuación establece la *cadenaWhere* y *listaParametrosWhere*:

for (String columna : pkArgumentos) {

Object o = pkValores[n++];

if (listaParametrosWhere.size() != 0) {

cadenaWhere = cadenaWhere + "AND ";

} else {

cadenaWhere = "where ";

}

cadenaWhere = cadenaWhere + columna + " = ? ";

listaParametrosWhere.add(o);

}

# Inserción

La operación de inserción de un registro presenta al usuario un formulario con los campos de la tabla para que se completen y finalmente, mediante una operación sql insert se creará el correspondiente registro en la tabla de la base de datos.

Se comienza con una petición al sistema que debe obedecer a la siguiente estructura:

tablas/insertar.action?TABLE\_SCHEM=JHA&TABLE\_NAME=CONTRATO&TABLE\_TYPE=TABLE

Se puede observar que no es necesario enviar los parámetros correspondientes a la clave primaria, puesto que no se va a realizar ninguna modificación o eliminación de datos.

## struts.xml

Se definen las acciones que se van a manejar. En primer lugar, la acción que recoge la petición de inserción que llevará al programa a presentar el formulario FormAlta.jsp:

<action name="insertar" class="es.ubu.alu.mydatabasejc.actions.TablasAction" method="insertar">

<result>/FormAlta.jsp</result>

<result name="error">/FormAlta.jsp</result>

</action>

En una segunda acción se realizará el guardado de los datos enviados por el usuario en un nuevo registro y la posterior presentación de la consulta de la tabla mediante una redirección a la acción *consulta* explicada en el capítulo anterior:

<action name="insertarGuardar" class="es.ubu.alu.mydatabasejc.actions.TablasAction" method="insertarGuardar">

<result name="consulta" type="redirectAction">

<param name="actionName">consulta</param>

<param name="TABLE\_SCHEM">%{TABLE\_SCHEM}</param><param name="TABLE\_NAME">%{TABLE\_NAME}</param>

<param name="TABLE\_TYPE">%{TABLE\_TYPE}</param>

</result>

</action>

## TablasAction.java

En esta clase se inicia la acción de inserción mediante la llamada al método insertar():

* **String insertar()**: Tras validar los datos:

if (TABLE\_NAME == null || "".equals(TABLE\_NAME)) {

sesion.put(ACTION\_ERROR, "Faltan.esquema.o.nombre.de.tabla");

return "tablas";

}

se obtiene el mapa de campos con las características adecuadas para una inserción, *Autoincremental* y *modificable*:

SQLCommand sqlCommand = new SQLCommand(connectionImpl);

arrayParametros = sqlCommand.getMap(mapaCompleto, SQLCommand.ISAUTOINCREMENT + SQLCommand.ISDEFINITELYWRITABLE + SQLCommand.ISWRITABLE).keySet();

El método getMap ya ha sido explicado en el punto anterior, modificación de registros.

* **String insertarGuardar()**: Después de que el usuario completa el formulario, la petición es recogida por este método que, inicialmente, comprueba la validez de los datos:

if (TABLE\_NAME == null || "".equals(TABLE\_NAME)) {

sesion.put(ACTION\_ERROR, "Faltan.esquema.o.nombre.de.tabla");

return "tablas";

}

y aquí la validación de los campos del formulario:

if (formCampos == null || formCampos.length == 0 || formValores == null || formValores.length == 0) {

sesion.put(ACTION\_ERROR, "Faltan.datos.para.hacer.la.operacion");

return "consulta";

}

La operación de inserción en la base de datos es similar a la tratada en la operación de modificación:

SQLCommand sqlComand = new SQLCommand(connectionImpl);

if (sqlComand.executeUpdate(TABLE\_SCHEM, TABLE\_NAME, mapaCompleto, formCampos, formValores)==0) {

sesion.put(ACTION\_MESSAGE, "No.se.ha.insertado.ningún.registro");

} else {

sesion.put(ACTION\_MESSAGE, "El.registro.ha.sido.insertado");

}

A la función executeUpdate no es preciso pasarle el tipo de operación puesto que por la sobrecarga de parámetros en la llamada, en la que envían solo el mapa de datos, sin incluir el mapa de la primary key, queda claro que se trata de una operación de inserción

## SQLCommand.java

En esta clase se crea el siguiente método para completar la operación de inserción:

* **int executeUpdate(String esquema, String tabla, Map<String, Integer[]> mapa, String[] campos, String[] valores)**.

En primer lugar se informan los campos *listaParametrosInsert*, *cadenaInsert* y *cadenaInsert2* con la siguiente llamada:

setSQLPartList(mapa, campos, valores, OPERACION\_INSERT);

A continuación se define la instrucción sql a realizar:

sql = String.format("INSERT INTO %s%s (%s) VALUES (%s)",getEsquema(esquema),tabla, cadenaInsert, cadenaInsert2);

Se define un *PreparedStatement*, se asingan los valores correspondientes y se ejecuta el *executeUpdate* del *ps*:

ps = connectionImpl.getConnection().prepareStatement(sql);

int parameterIndex = 1;

for (Object o : listaParametrosInsert) {

ps.setObject(parameterIndex++, o);

}

retorno = ps.executeUpdate();

* **setSQLPartList(Map<String, Integer[]> mapa, String[] pkArgumentos, String[] pkValores, int tipo)**: A partir de un array de campos y de otro de valores, en función del tipo, se completan las variables necesarias para la consiguiente instrucción sql. En el caso de inserción:

if (tipo == OPERACION\_INSERT) {

Para cada columna del array de campos

for (String columna : pkArgumentos) {

Se define el objeto con el valor correspondiente convertido al tipo correcto obtenido del mapa de columnas mediante la función getValor:

Object o = getValor(mapa.get(columna)[0], pkValores[n++]);

si el valor del objeto es nulo o vacío, no se incluirá el campo en las variables globales:

if (o==null) continue;

if ("".equals(String.valueOf(o))) continue;

Se completan las variables *cadenaInsert*, con los nombres de campo y *cadenaInsert2* con las interrogaciones oportunas

if (listaParametrosInsert.size() != 0) {

cadenaInsert = cadenaInsert + ", ";

cadenaInsert2 = cadenaInsert2 + ", ";

}

cadenaInsert = cadenaInsert + columna;

cadenaInsert2 = cadenaInsert2 + "?";

y se añade el valor a la *listaParametrosInsert*

listaParametrosInsert.add(o);

## FormAlta.jsp

Carpeta /

Se trata de una pantalla similar a la de edición aunque es más sencilla puesto que no es necesario presentar valor alguno y tampoco es necesario deshabilitar los campos no editables puesto que inicialmente solo se presentarán aquellos que sí lo son.

Los datos serán enviados mediante un formulario html a la acción insertarGuardar ya vista:

<s:form name="filtro" method="POST" theme="simple" id="filtro" action="insertarGuardar">

...

</s:form>

Dentro, una tabla html organiza los campos visibles

<table class="filtro">

...

</table>

y mediante un iterador se presentan los campos de la tabla:

<s:iterator value="arrayParametros" status="stat">

<tr>

..

</tr>

</s:iterator>

Se define la etiqueta y el campo oculto que devolverá al programa el nombre del campo:

<s:set var="campo"><s:property escapeHtml="false"/></s:set>

<td>

<s:text name="%{campo}" />:

<s:hidden name="formCampos" value="%{campo}"/>

</td>

y en una segunda columna el campo para que el usuario introduzca el dato:

<td><s:textfield name="formValores" value="%{getParameter(#stat.count-1)}"/></td>

Se completa el formulario html con un conjunto de campos ocultos que aportan información adicional necesaria para el programa:

<s:hidden name="TABLE\_SCHEM"/>

<s:hidden name="TABLE\_NAME"/>

<s:hidden name="TABLE\_TYPE"/>

<s:submit name="guardar" value="Guardar"/>