Consulta de tablas

Uno de los objetivos del trabajo es consultar los datos organizados en las tablas de una base de datos a la que nos conectamos mediante una conexión JDBC.

En los capítulos anteriores hemos podido trabajar con los MetaDatos de una conexión a una base de datos y hemos visto la existencia de una serie de métodos que nos dan información sobre la misma en forma de resultset. Entre estos métodos podemos destacar el método getTables que obtiene una lista (filtrada o no) de las tablas de distintos tipos y esquemas que existen en la base de datos.

Es el momento de adentrarnos en el trabajo sobre dichos objetos, las tablas, que conforman el corazón de una base de datos.

# MetaDataLink.java

Paquete es.ubu.alu.mydatabasejc.annotations

Acabamos de hablar de un método de DatabaseMetaData que va a requerir de un tratamiento distinto al resto de métodos de la misma clase. El método getTables deberá hacer que, de alguna manera junto a cada registro del resultset, se presente un link que nos permita viajar hasta la página correspondiente que muestre la visualización de los datos de esa tabla.

Para poder realizar esta labor de diferenciación entre métodos, y además para conseguir indicar la página que debe resolver la visualización de la información una vez se siga el link, e información adicional que se va a necesitar, vamos a utilizar una nueva clase de anotación que cumpla estos requisitos:

* Diferenciar estos métodos del resto
* Identificar página y parámetros del link destino

Esta clase define los siguientes métodos:

* **String action()**: Que definirá la acción struts que se debe ejecutar en la request
* **String namespace()**: Que definirá el espacio de nombres que acompaña a la acción
* **String[] parametros()**: Contendrá la colección de parámetros que deben ser enviados en la request.
* **int columnNumber()**: Indica en qué columna del resultset se deberá establecer el link

# DatabaseMetaDataImpl.java

Paquete es.ubu.alu.mydatabasejc.jdbc

En esta clase, debemos indicar en el método adecuado (getTables), que los registros de dicho método deben disponer de un objeto html para hacer un link a una acción struts concreta:

@MetaDataLink(action = "consulta", namespace = "/tablas", parametros = {"TABLE\_SCHEM","TABLE\_NAME"}, columnNumber = 3)

# DatabaseMetaDataAction.java

Paquete es.ubu.alu.mydatabasejc.actions

Es preciso modificar esta clase para incluir en la lista que presenta el resultset obtenido los datos necesarios para que en la correspondiente página web se muestren o no los objetos html que permitan hacer el link oportuno.

* **Object getInvoke(...)**: Este método ya utilizado se amplía para completar los atributos metodoLink, linkAction, linkNamespace, linkParametros y linkColumnNumber que permitirán a la jsp correspondiente presentar adecuadamente, o no hacerlo, el link oportuno. metodoLink permitirá indicarle que el método en el que se está trabajando es un método que debe contener links para cada registro de su resultset devuelto (true) o no (false). El método deberá estar marcado con la anotación *MetaDataLink* para que devuelva true. En caso contrario devolverá false:

se pregunta por la anotación pertinente sobre el método de trabajo

MetaDataLink anotacion = method.getAnnotation(MetaDataLink.class);

Inicialmente se asigna falso a metodoLink:

metodoLink = false;

si existe la anotación, se cambia esta asignación

if (anotacion == null) {

metodoLink = true;

y se definen la acción y el namespace desde la propia anotación y se retorna true

linkAction = anotacion.action();

linkNamespace = anotacion.namespace();

linkParametros = anotacion.parametros();

linkColumnNumber = anotacion.columnNumber();

* **List<List> getListInfo(ResultSet resultset)**: Si es un método que debe presentar un link se añade una columna más en la primera fila de la lista:

if (metodoLink) cabecera.add(null);

Para cada registro del resultset:

Se define la variable que contendrá los parámetros y sus nombres

String urlParametro = "";

Si el método es un método para hacer link, se añade la variable como primer elemento de la lista (será la columna 1)

if (metodoLink) record.add(urlParametro);

Para cada campo del resultset si el método es un método link, si el campo del resultset coincide con uno de los parámetros definidos en el método se añade a la variable

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

if (metodoLink) {

for (String linkParametro : linkParametros)

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

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

}

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

}

Finalmente se pasa la variable de nuevo a la lista

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

# DatabaseMetaData.jsp

Carpeta /

Esta página requiere una amplia transformación puesto que el resultset debe presentar un link en determinados casos y en otros no hacerlo. En los casos en que se debe presentar un link por cada línea, la primera columna del resultset no contiene datos a visualizar, sino información necesaria para el link, en concreto, una cadena de caracteres con la parte query de la request.

Se define una variable para contener el status del iterator interno:

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

En la primer registro de la tabla, se definen todos los elementos **th** si no es metodoLink y todos menos el primero si es metodoLink

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

<s:if test="!metodoLink">

<th><s:property /></th>

</s:if>

<s:elseif test="#stat.count != 1">

<th><s:property /></th>

</s:elseif>

</s:if>

Para el resto de registros, si es un metodoLink y estamos en la primera columna se define la variable url adecuadamente:

<s:if test="metodoLink">

<s:if test="#stat.count == 1">

<s:url action="%{linkAction}" namespace="%{linkNamespace}" var="urlTag"/>

<s:set var="url"><s:property value="urlTag"/>?<s:property/></s:set>

En el caso de que no sea la primera columna, se comprueba si es la columna en la que se debe establecer el link y se establece:

<s:if test="#stat.count-1 == linkColumnNumber">

<s:a href="%{url}" id="ir" ><s:property/></s:a>

</s:if>

<s:else>

<s:property/>

</s:else>

# struts.xml

Carpeta es.ubu.alu.mydatabasejc

Se crea un nuevo paquete que responda a las solicitudes del namespace /tablas:

<package name="tablas" extends="struts-default" namespace="/tablas">

<global-results>

<result name="input">/login.jsp</result>

</global-results>

Se define una nueva acción que recogerá la request de consulta de una tabla. La pasa al método consulta de la nueva clase TablasAction

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

<result>/Tabla.jsp</result>

</action>

En caso de error, se muestran de nuevo las tablas según los parámetros utilizados previamente:

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

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

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

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

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

</result>

o bien, la misma página web:

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

# TablasAction.java

Paquete es.ubu.alu.mydatabasejc.actions

Esta clase se encargará de manejar las acciones relacionadas con las tablas. Comenzaremos definiéndola como herencia de una clase nueva que llamamos *MenuAction.java* que comentaremos más adelante. Además implementa los interface *Preparable* y *SessionAwere* de la misma manera que lo hacía la clase *DatabaseMetaDataAction* que comprueban la validez de la conexión y retorna a *Login.jsp* si se produce algún error en dicha validación.

Además dispone de los siguientes métodos:

* **validate()**: Este método, una vez validada la conexión mediante la invocación al mismo método de la clase padre:

super.validarLogin(connectionImpl);

Recupera de la sesión del alumno el mapa de columnas de la tabla:

mapaCompleto = (Map<String, Integer[]>)sesion.get(TABLE\_SCHEM==null ? TABLE\_NAME : (TABLE\_SCHEM.equals("") ? TABLE\_NAME : TABLE\_SCHEM + "." + TABLE\_NAME));

y si no existe, la carga desde SQLCommand.getMap:

if (mapaCompleto == null) {

SQLCommand sqlCommand = new SQLCommand(connectionImpl);

mapaCompleto = sqlCommand.getMap(TABLE\_SCHEM, TABLE\_NAME);

}

y la guarda en la sesión del usuario para usos posteriores:

sesion.put(TABLE\_SCHEM==null ? TABLE\_NAME : (TABLE\_SCHEM.equals("") ? TABLE\_NAME : TABLE\_SCHEM + "." + TABLE\_NAME), mapaCompleto);

* **String consulta()**: Obtiene el correspondiente resultset de la tabla a consultar que es pasada en la request en los atributos *TABLE\_SCHEM* y *TABLE\_NAME.*

En primer lugar se manda error si no se reciben los parámetros correctos:

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

addActionError("Faltan.esquema.o.nombre.de.tabla");

return ERROR;

}

Establece y recoge los parámetros de búsqueda en sesión del usuario

Map<String, Integer> mapa = setParametrosSesion(TABLE\_SCHEM + "." + TABLE\_NAME);

Define el array de posibles parámetros para presentarlos en la jsp

arrayParametros = mapa.keySet();

Mediante un objeto SQLCommand se obtiene el resultset con los datos de la tabla que se consulta:

SQLCommand sqlCommand = new SQLCommand(connectionImpl);

rs = sqlCommand.executeQuery(TABLE\_SCHEM, TABLE\_NAME, arrayParametros, sesion);

se obtiene una lista con los campos primary key

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

String[] linkParametros = {};

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

while (rs2.next())

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

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

y finalmente se transforma el resultset en una lista para su visualización

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

* **private Map<String, Integer> setParametrosSesion(String tabla):**

Obtenidos los tipos de los parámetros, se convierten los valores recibidos a los tipos correspondientes y se ponen en la sesión para sucesivas llamadas a este método. Convierte cada valor de filtroValores al tipo necesario según se indica en el mapa obtenido de la sesion y grabado previamente, en la primera ejecución y lo añade a la sesión del usuario con el nombre correspondiente en filtroArgumentos

En primer lugar, extrae del mapa de campos los que sean searcheables:

SQLCommand sqlCommand = new SQLCommand(connectionImpl);

Map<String, Integer> mapa = sqlCommand.getMap(mapaCompleto, SQLCommand.ISSEARCHABLE);

En la segunda fase de la función, se analizan los parámetros de la request para añadir a la sesión los datos de filtrado enviados por el usuario

if (filtroArgumentos == null || filtroValores == null) return mapa;

obtiene el conjunto de parámetros

Set<String> set = mapa.keySet();

int i = 0;

para cada columna del resultset

for (String columna : set) {

String atributo = TABLE\_SCHEM + "." + TABLE\_NAME + "." + filtroArgumentos[i];

Object valor = null;

si el valor recibido no está en blanco se asigna a la variable valor de tipo Object parseandola convenientemente:

if (!"".equals(filtroValores[i]))

valor = SQLCommand.getValor(mapa.get(columna), filtroValores[i]);

y se añade a la sesión del usuario con el nombre de esquema, tabla y columna

sesion.put(atributo, valor);

finalmente, se devuelve el mapa del resultset para su uso posterior.

return mapa;

# SQLCommand.java

paquete es.ubu.alu.mydatabasejc.jdbc

Dispone de un método que obtiene de un resultset vacío un mapa con los nombres, tipos y características de todos los campos del resultset. El resultado es almacenado en la sesión del usuario para que no sea necesario realizar una llamada a este método en cada llamada a una funcionalidad relacionada con la tabla:

* **Map<String, Integer[]> getMap(String TABLE\_SCHEM, String TABLE\_NAME)**: El resultado obtenido es una estructura de tipo *Map* donde la clave es una cadena con el nombre de cada campo, y el contenido es un array de enteros. El primer elemento del array es el tipo de datos del campo, y el segundo representa es la característica del campo.

Primero se define el resultset de la tabla sin datos:

rs = connectionImpl.getConnection().createStatement().executeQuery(String.format("SELECT \* FROM %s%s WHERE 1=2", getEsquema(TABLE\_SCHEM), TABLE\_NAME));

Para cada campo del resultado

for (int i = 1; i <= rs.getMetaData().getColumnCount(); i++) {

la característica se calcula acumulando el correspondiente valor asignado previamente de forma excluyente

int característica = 0;

si el campo es autoincremental, se acumula el valor de ISAUTOINCREMENT

if (rs.getMetaData().isAutoIncrement(i)) característica += ISAUTOINCREMENT;

si el campo es case sensitive, se acumula el valor de ISCASESENSITIVE

if (rs.getMetaData().isCaseSensitive(i)) característica += ISCASESENSITIVE;

y así sucesivemente. Se crea el array con los dos valores enteros

Integer[] datos = {rs.getMetaData().getColumnType(i),característica};

y finalmente se añade al mapa la estructura de datos:

mapa.put(rs.getMetaData().getColumnName(i), datos);

Las constantes definidas son:

public static int ISAUTOINCREMENT = 1;

public static int ISCASESENSITIVE = 2;

public static int ISCURRENCY = 4;

public static int ISDEFINITELYWRITABLE = 8;

public static int ISNULLABLE = 16;

public static int ISREADONLY = 32;

public static int ISSEARCHABLE = 64;

public static int ISSIGNED = 128;

public static int ISWRITABLE = 256;

con esta asignación de valores se nos permite cualquier combinación de tipos de campos.

En esta clase se define un método estático que recibe en forma de entero el tipo sql de un dato y en forma de string el dato, y se parsea devolviendo el objeto del tipo correspondiente según el tipo indicado:

* **public static Object getValor(int tipo, String valor)**

Según el tipo recibido:

switch (mapa.get(columna)) {

si es un array:

case Types.ARRAY:

String[] valores = valor.split(",", 0);

List<String> lvalores = new ArrayList();

for (int k = 0; k < valores.length; k++) {

if (!"".equals(valores[k])) {

lvalores.add(valores[k].trim());

}

}

return lvalores.toArray(valores);

para tipos Integer

case Types.BIGINT:

case Types.INTEGER:

case Types.ROWID: return Integer.valueOf(valor);

para tipos Short

case Types.SMALLINT: return Short.valueOf(valor);

para tipos boolean

case Types.BIT:

case Types.BOOLEAN: return Boolean.valueOf(valor);

para tipos fecha

case Types.DATE:

case Types.TIME:

case Types.TIMESTAMP:

case Types.TIME\_WITH\_TIMEZONE: return Date.valueOf(valor);

para tipos float

case Types.DECIMAL:

case Types.DOUBLE:

case Types.FLOAT:

case Types.NUMERIC:

case Types.REAL: return Float.valueOf(valor);

para tipos string

case Types.CHAR:

case Types.DATALINK:

case Types.LONGNVARCHAR:

case Types.LONGVARCHAR:

case Types.NCHAR:

case Types.NVARCHAR:

case Types.SQLXML:

case Types.VARCHAR: return valor;

en otro caso, valor por defecto

default: return valor;

# MenuAction.java

Paquete es.ubu.alu.mydatabasejc.actions

Se convierte en la clase de la que heredan *DatabaseMetaDataAction* y *TablasAction* y que engloba los atributos y métodos que se utilizarán para presentar el menú.

Se compone de los siguientes métodos:

* **List<Menu> getMenus()**: Este método fue comentado ya en el capítulo XXX
* **List<List> getListInfo(...)**: Este método ha sido trasladado desde *DatabaseMetaDataAction* a esta nueva clase para poder ser reutilizado en otras actions que lo requieran. Para que siga haciendo su función, se ha cambiado su definición de private a protected de tal manera que pueda ser utilizado en todas las clases heredadas.

Además, ahora recibe como argumentos el *boolean metodoLink* y el *String[] linkParametros* para construir una columna adicional (columna 0) con los parámetros indicados para realizar el correspondiente link posteriormente cuando la lista resultante sea visualizada en la página jsp

# Tabla.jsp

Recoge el resultset de la ejecución de la consulta sql subyacente y la presenta utilizando para ello dos iteradores siguiendo el mismo patrón utilizado en DatabaseMetaData.jsp. El código se ha simplificado puesto que ahora no es necesario contemplar ni el metodoLink en el resultset ni la idiomatización de la primera columna:

<table class="tabla">

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

<s:if test="#status.first == true"><thead></s:if>

<tr>

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

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

<th><s:property /></th>

</s:if>

<s:else>

<td><s:property/></td>

</s:else>

</s:iterator>

</tr>

<s:if test="#status.first == true"></thead></s:if>

</s:iterator>

</table>

# filtro.jsp

Se incluyen en el formulario que se envía en la request, los parámetros TABLE\_SCHEM y TABLE\_NAME

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

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