UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Máster Online en Ingeniería Informática

TFM del Máster Online en Ingeniería Informática

Creación de componentes para el mantenimiento web de datos en sistemas de gestión de bases de datos relaciones

Presentado por José Ignacio Huidobro del Arco

en Universidad de Burgos - 11 de enero de 2019

Tutor: María Belén Vaquerizo García

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Máster Online en Ingeniería Informática

Da. María Belén Vaquerizo García, profesora del departamento de Ingeniería Civil, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. José Ignacio Huidobro del Arco, con DNI 07872311D, ha realizado el Trabajo final de Máster en Ingeniería Informática titulado Creación de componentes para el mantenimiento web de datos en sistemas de gestión de bases de datos relaciones.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección de la que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 11 de enero de 2019

Vo. Bo. del Tutor:

D. María Belén Vaquerizo García

Resumen

Falta por desarrollar

Descriptores

Falta por desarrollar

Abstract

Falta por desarrollar en inglés

Keywords

Falta por desarrollar en inglés

Índice general

Introducción 1

Objetivos del proyecto 2

Conceptos teóricos 3

3.1. La API de Java javaee-web-api-7.0 3

3.2. La API de Java JDBC 12

Técnicas y herramientas 32

4.1. Apache Tomcat 32

4.2. Apache Struts 2 55

4.3. Maven 65

4.4. NetBeans IDE 8.2 66

Aspectos relevantes del desarrollo del proyecto 69

5.1. Conexión a la base de datos 69

5.2. Presentación de la base de datos 80

5.3. Presentación de DatabaseMetaData 92

5.4. Filtro y ordenación de resultset de DatabaseMetaData 102

5.5. Consulta de tablas 110

5.6. Modificación de datos 127

Trabajos relacionados 161

6.1. Herramientas de administración de bases de datos 161

6.2. phpMyAdmin 172

6.3. Adminer 172

6.4. Oracle Enterprise Manager 173

6.5. Herramientas de escritorio 173

6.6. Conclusiones 177

Conclusiones y Líneas de trabajo futuras 180

Bibliografía 181

Índice de figuras

Figura 6.1: API JDBC para aplicaciones..... 15
Figura 6.2: API del controlador JDBC..... 15
Figura 6.3: API del controlador JDBC..... 15
Figura 6.4: API nativa..... 16
Figura 6.5: Controlador Java puro para Database Middleware..... 16
Figura 6.6: Controlador Java puro directo a la base de datos..... 17
Figura 8.1: redirección http - https: redirección http - https..... 70
Figura 8.2: Esquema general de inicio en aplicación..... 72
Figura 8.3: Validación de login..... 74
Figura 8.4: Procesamiento de login..... 75
Figura 8.5: Presentación de la pantalla inicial..... 81
Figura 8.6Lógica de inicial.sp..... 87
Figura 8.7: Lógica de menu.jsp..... 88
Figura 8.8: Métodos resultset e info de DatabaseMetaDataAction.java..... 93
Figura 8.9: Lógica de consulta de una tabla..... 111
Figura 8.10: getInvoke para obtener resultsets o listas..... 114

Figura 8.11: getListInfo. Obtención de lista a partir de resultset 116

Figura 8.12: Lógica de la eliminación de un registro 134

Figura 8.13: Lógica de edición de registro 142

Figura 8.14: Lógica para guardar un registro editado 149

Figura 8.15: Lógica de inserción de un registro 153

Figura 8.16: Lógica de guardado en la inserción de un registro..... 155

Índice de tablas

Tabla 6.1: Variables implícitas en páginas JSP 8

Tabla 6.2: Directivas en páginas JSP 10

Tabla 8.1: JDBC Technology Core 19

Tabla 8.2: Paquete opcional de JDBC 20

Tabla 7.1: Estado del arte 165

Tabla 7.2: Interfaz de usuario de herramientas 167

Tabla 7.3: Características de herramientas basadas en navegador 171

Introducción

El trabajo consiste en la realización de una aplicación web que permita conectar con una base de datos relacional a través del estándar JDBC. Una vez conseguida la conexión, la aplicación debe ser capaz de inspeccionar la base de datos llegando incluso a realizar las operaciones básicas de consulta, modificación, eliminación e inserción de registros en las tablas de la misma.

Objetivos del proyecto

El objetivo es plantear y desarrollar los principales pasos para la construcción de una aplicación Web dinámica cuyo fin sea el mantenimiento de Bases de Datos remota, via internet, ofreciendo para ello, al usuario, un entorno amigable e intuitivo que le permitirá efectuar cualquier operación básica de mantenimiento sobre una base de datos relacional (inserciones, borrados, modificaciones, consultas,..). La página web constará de una ayuda interactiva en línea, a la que el usuario tendrá la oportunidad de acceder desde cualquiera de las páginas de la aplicación, y que le será útil tanto para poder resolver dudas sobre el manejo de la aplicacion, como en cuanto a la explicación de diversos conceptos teóricos que ayuden en la comprensión de sus operaciones. La aplicación ha de diseñarse para poder administrar cualquier tipo de bases de datos: SqlServer, MySql,.. e incluso Access, ac-cediendo y tratando los metadatos de la base de datos, para permitir al sistema a través de mecanismos JDBC obtener la información necesaria para la realización de las operaciones de mantenimiento que decida el usuario. En resúmen, dicho sistema constará de distintos elementos que permitirán la realización de operaciones básicas sobre cualquier base de datos relacional (inserciones, borrados, modificaciones, consultas,..) de un modo sencillo para el usuario, con la posibilidad de aportar ayuda interactiva para guiar al usuario, paso a paso, de forma dinámica en la realización de cualquiera de estas operaciones.

Conceptos teóricos

3.1. La API de Java javaee-web-api-7.0

La API Java-web fue presentada en la versión 6 de Java EE con un perfil que optimiza radicalmente la plataforma y permite la creación de nuevos servidores de aplicaciones ligeros y ágiles para el desarrollo de aplicaciones web. La diferencia real, sin embargo, con la API de javaee-api no parece tan grande y se reducen a un soporte de JMS y una reducción en el soporte de xml. [1]

Si nos centramos, pues, en la API de Java EE, recibe distintos nombres como por ejemplo **Java Platform Enterprise Edition, Java EE, J2EE, Java Empresarial.** Se trata de una plataforma de programación para desarrollar y ejecutar software de aplicaciones en lenguaje Java. Se ejecuta sobre un servidor de aplicaciones.

Java EE tiene varias especificaciones de API, como por ejemplo, JDBC, RMI, e-mail, JMS, Servicios Web, XML, etc. Además también contiene algunas especificaciones para componentes que incluyen Enterprise JavaBeans, servlets, portlets o JavaServer Pages. Todo ello permite crear una aplicación de empresa, portable entre plataformas, escalable e integrable con tecnologías anteriores.

Servidores de aplicaciones Java EE certificados

[2] La lista relativa a la versión 5 es:

- JOnAS
- JBoss
- Sun Java System Application Server Platform Edition 9.0

- Oracle WebLogic Application Server 10.0
- Servidor de Aplicaciones SAP NetWeaver
- JEUS 6
- Apache Geronimo 2.0
- IBM WebSphere Application Server.
- Oracle Containers for Java EE 11.
- GlassFish
- Apache OpenEJB via Apache Geronimo.

A continuación nombraremos y describiremos brevemente las APIs generales que se han utilizado en el presente trabajo

java.sql

Debido a su importancia en el presente trabajo, será tratado en un capítulo aparte

Java Servlet

[3] Los servlets con utilizados comunmente para estender 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. Equivalen a otras tecnologías de contenido dinámico web como PHP y ASP.NET

El uso más comun es generar páginas web de forma dinámica a partir de los parámetros de la petición que envíe el navegador web.

Ciclo de vida

Cuando un servidor carga un servlet, ejecuta el método `init` del servlet. El proceso de inicialización debe completarse antes de poder manejar peticiones de los clientes, y antes de que el servlet sea destruido.

Aunque muchos servlets se ejecutan en servidores multi-thread, los servlets no tienen problemas de concurrencia durante su inicialización. El servidor llama sólo una vez al método `init` al crear la instancia del servlet, y no lo llamará de nuevo a menos que vuelva a recargar el servlet. El servidor no puede recargar un servlet sin primero haber destruido el servlet llamando al método `destroy`.

Después de la inicialización, el servlet puede dar servicio a las peticiones de los clientes. Estas peticiones serán atendidas por la misma instancia del servlet, por lo que hay que tener cuidado al acceder a variables compartidas, ya que podrían darse problemas de sincronización entre requerimientos simultáneos.

Los servlets se ejecutan hasta que el servidor los destruye, por cierre del servidor o bien a petición del administrador del sistema. Cuando un servidor destruye un servlet, ejecuta el método `destroy` del propio servlet. Este método sólo se ejecuta una vez y puede ser llamado cuando aún queden respuestas en proceso, por lo que hay que tener la atención de esperarlas. El servidor no ejecutará de nuevo el servlet hasta haberlo cargado e inicializado de nuevo.

Interacción con el cliente

[4]

El Interface Servlet. La abstracción central en el API Servlet es el interface Servlet. Todos los servlets implementan este interface, bien directamente o, más comúnmente, extendiendo una clase que lo implemente como `HttpServlet`

Cuando un servlet acepta una llamada de un cliente, recibe dos objetos.

- Un ServletRequest, que encapsula la comunicación desde el cliente al servidor.
- Un ServletResponse, que encapsula la comunicación de vuelta desde el servlet hacia el cliente.

ServletRequest y ServletResponse son interfaces definidos en el paquete javax.servlet.

El Interface ServletRequest. El Interface `ServletRequest` permite al servlet acceder a :

- Información como los nombres de los parámetros pasados por el cliente, el protocolo (esquema) que está siendo utilizado por el cliente, y los nombres del host remote que ha realizado la petición y la del server que la ha recibido.

- El stream de entrada, ServletInputStream. Los Servlets utilizan este stream para obtener los datos desde los clientes que utilizan protocolos como los métodos **POST** y **PUT** del **HTTP**.

Los interfaces que extienden el interface ServletRequest permiten al servlet recibir más datos específicos del protocolo. Por ejemplo, el interfaz HttpServletRequest contiene métodos para acceder a información de cabecera específica HTTP.

El Interface ServletResponse. El Interface `ServletResponse` le da al servlet los métodos para responder al cliente.

- Permite al servlet seleccionar la longitud del contenido y el tipo MIME de la respuesta.
- Proporciona un stream de salida, ServletOutputStream, y un Writer a través del cual el servlet puede responder datos.

Los interfaces que extienden el interface ServletResponse le dan a los servlets más capacidades específicas del protocolo. Por ejemplo, el interfaz HttpServletResponse contiene métodos que permiten al servlet manipular información de cabecera específica HTTP.

Capacidades Adicionales de los Servlets HTTP. Las clases e interfaces descritos anteriormente constituyen un servlet básico. Los servlets HTTP tienen algunos objetos adicionales que proporcionan capacidades de seguimiento de sesión. El desarrollador de servlets pueden utilizar esos APIs para mantener el estado entre el servlet y el cliente persistiendo a través de múltiples conexiones durante un periodo de tiempo. Los servlets HTTP también tienen objetos que proporcionan cookies. El API cookie se utiliza para guardar datos dentro del cliente y recuperar esos datos.

JavaServer Pages

JavaServer Pages (JSP) [5] es una tecnología que ayuda a los desarrolladores de software a crear páginas web dinámicas basadas en HTML y XML, entre otros tipos de documentos. JSP es similar a PHP, pero usa el lenguaje de programación Java.

Para desplegar y correr JavaServer Pages, se requiere un servidor web compatible con contenedores servlet como Apache Tomcat o Jetty.

TagLibs -> JSP -> Servidor Aplicaciones (Servlets) -> Cliente (Navegador)

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.

La principal ventaja de JSP frente a otros lenguajes es que el lenguaje Java es un lenguaje de propósito general que excede el mundo web y que es apto para crear clases que manejen lógica de negocio y acceso a datos de una manera prolija. Esto permite separar en niveles las aplicaciones web, dejando la parte encargada de generar el documento HTML en el archivo JSP.

Otra ventaja es que JSP hereda la portabilidad de Java, y es posible ejecutar las aplicaciones en múltiples plataformas sin cambios. Es común incluso que los desarrolladores trabajen en una plataforma y que la aplicación termine siendo ejecutada en otra.

Los servlets y Java Server Pages (JSPs) son dos métodos de creación de páginas web dinámicas en servidor usando el lenguaje Java.

Para empezar, los JSPs y servlets se ejecutan en una máquina virtual Java, lo cual permite que, en principio, se puedan usar en cualquier tipo de ordenador, siempre que exista una máquina virtual Java para él. Cada servlet (o JSP, a partir de ahora lo usaremos de forma indistinta) se ejecuta en su propio hilo, es decir, en su propio contexto; pero no se comienza a ejecutar cada vez que recibe una petición, sino que persiste de una petición a la siguiente, de forma que no se pierde tiempo en invocarlo (cargar programa + intérprete). Su persistencia le permite también hacer una serie de cosas de forma más eficiente: conexión a bases de datos y manejo de sesiones, por ejemplo.

Las JSPs son en realidad una forma alternativa de crear servlets ya que el código JSP se traduce a código de servlet Java la primera vez que se le invoca y

en adelante es el código del nuevo servlet el que se ejecuta produciendo como salida el código HTML que compone la página web de respuesta.

JSP puede ser visto como una abstracción de alto nivel de los servlets Java. Las JavaServer Pages son traducidas a servlets en tiempo real; cada servlet es guardado en caché y rehusado hasta que la JSP original es modificada. Dicho en otras palabras un JSP son páginas java para un ambiente web.

Sintaxis

A continuación se detallan las cuestiones relativas a la sintaxis JSP más relevantes

Variables implícitas

Las páginas JSP incluyen ciertas variables privilegiadas sin necesidad de declararlas ni configurarlas. Se puede ver la estrecha relación de JSP con Servlet

Variable	Clase
pageContext	javax.servlet.jsp.PageContext
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
session	javax.servlet.http.HttpSession
config	javax.servlet.ServletConfig
application	javax.servlet.ServletContext
out	javax.servlet.jsp.JspWriter
page	java.lang.Object
exception	java.lang.Exception

Tabla 3.1: Variables implícitas en páginas JSP

Directivas. Son etiquetas a partir de las cuales se genera información que puede ser utilizada por el motor de JSP. No producen una salida visible al usuario sino que configura cómo se ejecutará la página JSP.

Su sintaxis es:

```
<%@ directiva atributo="valor" %>
```

Las directivas disponibles son:

- `include`: Incluye el contenido de un fichero en la página mediante el atributo file. `<%@ include file="cabecera.html" %>`
- `taglib`: Importa bibliotecas de etiquetas (Tag Libraries) `<%@ taglib uri="/tags/struts-html" prefix="html" %>`
- `page`: Especifica atributos relacionados con la página a procesar. Los atributos son:

Atributo	Sintaxis	Utilización
import	`<%@ page import="class; class" %>`	Importa clases y paquetes Java para ser utilizadas dentro del fichero JSP.
session	`<%@ page session="false" %>`	Especifica si utiliza los datos contenidos en sesión; por defecto "true".
contentType	`<%@ page contentType="class; class" %>`	Especifica el tipo MIME del objeto "response"; por defecto "text/html; charset=ISO-8859-1".
buffer	`<%@ page buffer="12KB" %>`	Buffer utilizado por el objeto writer "out"; puede tomar el valor de "none"; por defecto "8KB".
errorPage	`<%@ page errorPage="/path_to_error_page" %>`	Especifica la ruta de la página de error que será invocada en caso de producirse una excepción durante la ejecución de este fichero JSP.

isErrorPage	`<%@ page isErrorPage="true" %>`	Determina si este fichero JSP es una página que maneja excepciones. Únicamente a este tipo de páginas pueden acceder a la variable implícita "exception", que contiene la excepción que provocó la llamada a la página de error.

Tabla 3.2: Directivas en páginas JSP

Otros

- Declaraciones: Nos permiten declarar variables, funciones y datos estáticos:
  `<%! int maxAlumnosClase = 30; %>`
- Scriptles: Son partes de código Java incrustadas entre los elementos estáticos de la página:
  `<% ... código Java ... %>`
- Expresiones: se evalúan dentro de la servlet y no deben acabar en ";":
  `<%= maxAlumnosClase + 1%>`

Etiquetas. Permiten simplificar el código y dar mayor funcionalidad. Entre las ventajas a considerar por el uso de etiquetas destacamos la facilidad de aprendizaje, de mantenimiento, el fomento de la modularidad y la reutilización y la simplificación del código y la reducción del número de líneas necesarias.

Las etiquetas están definidas en las bibliotecas de etiquetas, identificadas en el fichero descriptor de la aplicación (web.xml). Aunque existe una gran cantidad de bibliotecas de etiquetas para JSP, a continuación mostraremos someramente las etiquetas pertenecientes a la propia especificación JSP que proporcionan una funcionalidad básica.

Un primer grupo de etiquetas proporciona funcionalidad a nivel de la página de una manera muy simple:

- `<jsp:forward>`, redirige la request a otra URL
- `<jsp:include>`, incluye el texto de un fichero dentro de la página
- `<jsp:plugin>`, descarga un plugin de Java (una applet o un Bean).

Un segundo grupo permite manipular componentes `JavaBean` sin conocimientos de Java.

- `<jsp:useBean>`, permite manipular un Bean (si no existe, se creará el Bean), especificando su ámbito (scope), la clase y el tipo.
- `<jsp:getProperty>`, obtiene la propiedad especificada de un bean previamente declarado y la escribe en el objeto response.
- `<jsp:setProperty>`, establece el valor de una propiedad de un bean previamente declarado.

Otros grupos importantes de etiquetas utilizables en páginas JSP son:

- Etiquetas JSTL: Son proporcionadas por Sun dentro de la distribución de JSTL.
  - core: iteraciones, condicionales, manipulación de URL y otras funciones generales.
  - xml: para la manipulación de XML y para XML-Transformation.
  - sql: para gestionar conexiones a bases de datos.
  - i18n: para la internacionalización y formateo de las cadenas de caracteres como cifras.
- Etiquetas Struts TagLib: Distribuidas por Apache para funcionar junto con el Framework de Struts. Las veremos en más detalle cuando nos centremos en este framework que será ampliamente utilizado en el trabajo.

3.2. La API de Java JDBC

Introducción

[6] Se trata del conjunto de funciones y procedimientos, clases y objetos, cuya función es proporcionar acceso a datos universales desde el lenguaje de programación Java pudiendo acceder a prácticamente cualquier fuente de datos incluyendo bases de datos relacionales, hojas de cálculo o archivos planos.

Esta tecnología proporciona una base común sobre la que se pueden construir interfaces alternativos o herramientas complejas.

La API de JDBC se compone de dos paquetes java:

- java.sql
- javax.sql

Para utilizar la API con un sistema de administración de base de datos particular, es necesario un controlador basado en tecnología JDBC que medie entre JDBC y la base de datos. El controlador puede estar escrito íntegramente en lenguaje Java o combinado con los métodos nativos de la interfaz nativa de Java (JNI).

A continuación se indican las compañías que han respaldado la API de JDBC y han creado o están creando productos basados en ella [7]:

- Agave Software Design
- Asgard Software
- Atinav, Inc.
- Automation Technology, Inc.
- BEA
- Borland
- Bulletproof
- Caribou Lake Software
- Daffodil Software
- DataMirror
- Dharma Systems Inc.
- eFORCE
- Empress Software Inc.

- Esker Software
- Fujitsu Siemens Computers
- Gupta Technologies
- Hewlett-Packard
- HP Integrity NonStop computing
- Hit Software
- Hummingbird
- IBM's Database 2 (DB2)
- IBM's Informix Software Inc.
- IDS Software
- i-net software
- InterSystems Corporation
- iWay Software
- JNetDirect
- Minisoft
- mysql
- NEON Systems
- New Atlanta Communications
- Novell
- OpenLink Software
- Open Text Corporation
- Oracle Corporation
- Pervasive Software
- POET Software
- postgreSQL
- Progress Software
- Quest Software
- Recital Corporation
- RogueWave Software Inc.
- SAS Institute Inc.
- Siemens AG
- Simba Technologies, Inc.
- Sun Software
- Sunopsis S.A.
- Sybase Inc.
- Symantec
- Thunderstone

- Thought Inc.
- Trifox, Inc.
- Yard Software GmbH

Resumen de JDBC

[8] Este trabajo de fin de máster no pretende centrarse exclusivamente en cuestiones relativas a JDBC. Aunque constituye la base sobre la que se construye la aplicación existen otras tecnologías que se aplican conjuntamente para obtener el producto necesario.

Este capítulo se desarrolla a modo de resumen de lo que forma parte del API de JDBC, sus características y posibilidades y cómo utilizarlo.

Como se trata de una API de Java, esto nos permitirá poder utilizar el producto desarrollado, en cualquier plataforma dotada de una JVM (máquina virtual Java) independientemente del sistema operativo subyacente a la misma.

La API de JDBC hace posible tres cosas:

- Establecer una conexión con una base de datos o fuente de datos basada en tablas
- Enviar sentencias SQL
- Procesar los resultados obtenidos de tales sentencias

Arquitectura

La API tiene dos conjuntos principales de interfaces:

- La API JDBC para aplicaciones. Las aplicaciones y applets pueden acceder a las bases de datos a través de su correspondiente API

Figura 3.1: API JDBC para aplicaciones

- La API del controlador JDBC de nivel inferior para controladores.

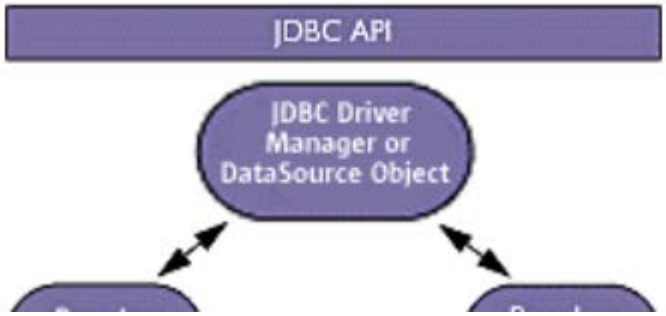


Figura 3.2: API del controlador JDBC

Esta API proporciona a los diseñadores de controladores cuatro formas de acceder a la API de JDBC:

- Tipo 1: Puente JDBC-ODBC, más el controlador ODBC. Esta combinación proporciona el acceso JDBC a través de controladores ODBC. Para uso experimental o si no hay otro

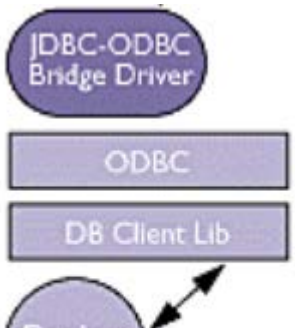


Figura 3.3: API del controlador JDBC

controlador disponible.

- Tipo 2: API nativa en parte del controlador habilitado para la tecnología Java

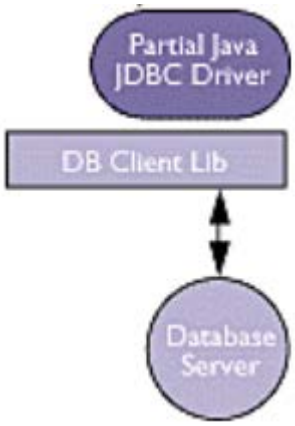
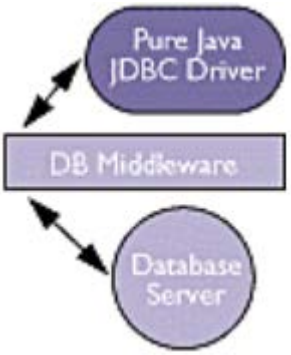


Figura 3.4: API nativa

- Tipo 3: Controlador Java puro para Database Middleware. Este tipo de controlador traduce las llamadas JDBC al protocolo del proveedor de middleware que luego se traduce a un protocolo DBMS por un servidor de middleware de tal forma que el middleware proporciona conectividad a muchas bases de datos diferentes
- Tipo 4: Controlador Java puro directo a la base de datos: el controlador convierte las llamadas JDBC al protocolo de red

Figura 3.5: Controlador Java puro para Database Middleware



utilizado directamente por los sistemas de base de datos proporcionando una solución práctica para el acceso a la intranet.

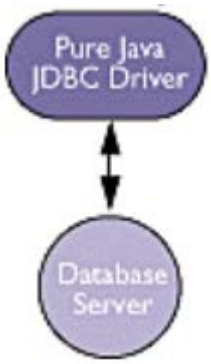


Figura 3.6: Controlador Java puro directo a la base de datos

Ventajas de la tecnología JDBC

- Aprovechamiento de datos existentes: Con esta tecnología, las empresas no están bloqueadas en ninguna arquitectura propietaria pudiendo seguir utilizando sus bases de datos incluso si se almacenan en diferentes sistemas.
- Desarrollo simplificado: junto con la API de Java, se consigue que el desarrollo de aplicaciones sea fácil y económico ocultando la complejidad de muchas tareas de acceso a datos
- No precisa configuración de red: Toda la información necesaria para establecer una conexión está completamente definida por la URL de JDBC o por un objeto DataSource registrado.

Características clave

- Acceso completo a los metadatos: El acceso a los metadatos permite el desarrollo de aplicaciones sofisticadas con comprensión de las capacidades subyacentes de una conexión de base de datos.
- Sin instalación: No se requiere una instalación especial del controlador basado en tecnología JDBC.
- Conexión de base de datos identificada por URL: se aprovechan las ventajas de las URL estándar de internet para identificar las conexiones de base de datos. Existe una forma incluso mejor que es utilizando un objeto DataSource que hace que el código sea aún más portable.
- Incluida en la plataforma Java 2: como parte central de dicha plataforma, está disponible en cualquier lugar donde se encuentre ella.
- Requisitos: Los mismos requisitos hardware que para una plataforma Java 2 (ya sea Java 2 SDK, J2SE o J2EE) más una base de datos SQL y un controlador basado en tecnología JDBC para esa base de datos

Características

[9] A parte de las características clave indicadas en el apartado anterior, a continuación detallaremos a modo de resumen las características técnicas más importante y los beneficios obtenidos por dichas características:

Puesto que existen dos paquetes principales, explicaremos las características de cada de uno de ellos

JDBC Technology Core (java.sql)

Característica	Beneficio
Mejoras en el conjunto de resultados	Facilidad de programación
- Conjunto de resultados desplazable	Posibilidad de mover el cursor de un conjunto de resultados a una fila específica. Esta característica es utilizada por las herramientas GUI y para la actualización programática.

- Conjunto de resultados actualizable.	Capacidad para utilizar comandos de lenguaje de programación Java en lugar de SQL
Nuevos tipos de datos compatibles	Mejora del rendimiento (capacidad para manipular objetos grandes como BLOB y CLOB sin traerlos al cliente desde el servidor DB)
Actualizaciones por lotes	Mejora del rendimiento (el envío de múltiples actualizaciones a la base de datos para su procesamiento como un lote puede ser mucho más eficiente que enviar instrucciones de actualización por separado)
Puntos de salvaguardia	Posibilidad de revertir transacciones a donde se establece un punto de salvaguarda

Tabla 3.3: JDBC Technology Core

Paquete opcional de JDBC (javax.sql)

Característica	Beneficio
Soporte JNDI	Facilidad de implementación (le da independencia al controlador JDBC, hace que las aplicaciones JDBC sean más fáciles de administrar)
Agrupación de conexiones	Mejora del rendimiento (una agrupación de conexiones es un caché de conexiones de base de datos que se mantiene en la memoria, para que las conexiones puedan reutilizarse)
Transacciones distribuidas	Importante para implementar un sistema de procesamiento de transacciones distribuidas.

JavaBeans (Objetos RowSet)	Envíe datos a través de una red a clientes ligeros, como navegadores web, computadoras portátiles, PDA, etc. Acceda a cualquier fuente de datos tabular, incluso hojas de cálculo o archivos planos Haga que los conjuntos de resultados sean desplazables o actualizables cuando el controlador JDBC no admita la capacidad de desplazamiento y la capacidad de actualización
Referencia a JDBC Rowset	Encapsule un controlador como un componente JavaBeans para usar en una GUI
Agrupación de estados de cuenta	Mejora del rendimiento (agrupando los estados de cuenta y agrupando las conexiones)

Tabla 3.4: Paquete opcional de JDBC

La interfaz de acceso a bases de datos

[10] La arquitectura JDBC se basa en un conjunto de clases Java que permiten conectar con bases de datos, crear y ejecutar sentencias SQL o recuperar y modificar la información almacenada en una base de datos.

A continuación se describen los distintos tipos de operaciones

java.sql.Driver y java.sql.DriverManager

La interfaz `java.sql.Connection` representa una conexión con una base de datos. En una aplicación podemos obtener una o más conexiones para una o más bases de datos utilizando drivers JDBC. Cada driver implementa la interfaz `java.sql.Driver` y uno de los métodos que define esta interfaz es el método `connect()` que permite establecer una conexión y obtener un objeto `Connection`.

Pero en lugar de acceder directamente a clases que implementan `java.sql.Driver`, el enfoque estándar es registrar cada driver con `java.sql.DriverManager` y utilizar los métodos proporcionados en esta clase para obtener conexiones.

Las URL de JDBC. La noción de una URL en JDBC es muy similar al modo típico de utilizar las URL. Las URL de JDBC proporcionan un modo de identificar un driver de base de datos. Una URL de JDBC representa un driver y la información adicional necesaria para localizar una base de datos y conectar a ella. Su sintaxis es la siguiente:

```
jdbc:<subprotocol>:<subname>
```

Existen tres partes separadas por dos puntos:

- Protocolo: En la sintaxis anterior, jdbc es el protocolo. Éste es el único protocolo permitido en JDBC.
- Subprotocolo: Utilizado para identificar el driver que utiliza la API JDBC para acceder al servidor de bases de datos. Este nombre depende de cada fabricante.
- Subnombre: La sintaxis del subnombre es específica del driver.

Por ejemplo, para una base de datos MySQL llamada “Bank”', la URL al que debe conectar es:

```
jdbc:mysql:Bank
```

Alternativamente, si estuviéramos utilizando Oracle mediante el puente JDBC-ODBC, nuestra URL sería:

```
jdbc:odbc:Bank
```

Como puede ver, las URL de JDBC son lo suficientemente flexibles como para especificar información específica del driver en el subnombre.

Clase DriverManager. El propósito de la clase `java.sql.DriverManager` (gestor de drivers) es proporcionar una capa de acceso común encima de diferentes drivers de base de datos utilizados en una aplicación. En este enfoque, en lugar de utilizar clases

de implementación `Driver` directamente, las aplicaciones utilizan la clase `DriverManager` para obtener conexiones. Esta clase ofrece tres métodos estáticos para obtener conexiones.

Sin embargo, `DriverManager` requiere que cada driver que necesite la aplicación sea registrado antes de su uso, de modo que el `DriverManager` sepa que está ahí.

El enfoque JDBC para el registro de un driver de base de datos puede parecer oscuro al principio. Fíjese en el siguiente fragmento de código que carga el driver de base de datos de MySQL:

```
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
    // Driver no encontrado
}
```

En tiempo de ejecución, el `ClassLoader` localiza y carga la clase `com.mysql.jdbc.Driver` desde la ruta de clases utilizando el cargador de clase de autoarranque. Mientras carga una clase, el cargador de clase ejecuta cualquier código estático de inicialización para la clase. En JDBC, se requiere que cada proveedor de driver registre una instancia del driver con la clase `java.sql.DriverManager` durante esta inicialización estática. Este registro tiene lugar automáticamente cuando el usuario carga la clase del driver (utilizando la llamada `Class.forName()`).

Una vez que el driver ha sido registrado con el `java.sql.DriverManager`, podemos utilizar sus métodos estáticos para obtener conexiones. El gestor de drivers tiene tres variantes del método estático `getConnection()` utilizado para establecer conexiones. El gestor de drivers delega estas llamadas en el método `connect()` de la interfaz `java.sql.Driver`.

Dependiendo del tipo de driver y del servidor de base de datos, una conexión puede conllevar una conexión de red física al servidor de base de datos o a un proxy de conexión. Las bases de datos integradas no requieren conexión física. Exista o no una conexión física, el objeto de

conexión es el único objeto que utiliza una conexión para comunicar con la base de datos. Toda comunicación debe tener lugar dentro del contexto de una o más conexiones.

Consideremos ahora los diferentes métodos para obtener una conexión:

- `public static Connection getConnection(String url)`
  `java.sql.DriverManager` recupera el driver apropiado del conjunto de drivers registrados. El URL de la base de datos está especificado en la forma de `jdbc:subprotocol:subname`. Para poder obtener una conexión a la base de datos es necesario que se introduzcan correctamente los parámetros de autenticación requeridos por el servidor de bases de datos.
- `public static Connection getConnection(String url, java.util.Properties info)`

Este método requiere un URL y un objeto `java.util.Properties`. El objeto `Properties` contiene cada parámetro requerido para la base de datos especificada. La lista de propiedades difiere entre bases de datos. Dos propiedades comúnmente utilizadas para una base de datos son `autocommit=true` y `create=false`. Podemos especificar estas propiedades junto con el URL como `jdbc:subprotocol:subname;autocommit=true;create=true` o podemos establecer estas propiedades utilizando el objeto `Properties` y pasar dicho objeto como parámetro en el anterior método `getConnection()`.

```
String url = "jdbc:mysql:Bank";
Properties p = new Properties();
p.put("autocommit", "true");
p.put("create", "true");
Connection connection =
DriverManager.getConnection(url, p);
```

En caso de que no se adjunten todas las propiedades requeridas para el acceso, se generará una excepción en tiempo de ejecución.

- La tercera variante toma como argumentos además del URL, el nombre del usuario y la contraseña. Fíjese en el siguiente ejemplo, utiliza un driver MySQL, y requiere un nombre de usuario y una contraseña para obtener una conexión:

```
String url = "jdbc:mysql:Bank";
String user = "root";
String password = "nadiuska";
Connection connection =
DriverManager.getConnection(url, user,
password);
```

Observe que todos estos métodos están sincronizados, lo que supone que sólo puede haber un hilo accediendo a los mismos en cada momento. Estos métodos lanzan una excepción `SQLException` si el driver no consigue obtener una conexión.

Interfaz `Driver`. Cada driver debe implementar la interfaz `java.sql.Driver`. En MySQL, la clase `com.mysql.jdbc.Driver` implementa la interfaz `java.sql.Driver`.

La clase `DriverManager` utiliza los métodos definidos en esta interfaz. En general, las aplicaciones cliente no necesitan acceder directamente a la clase `Driver` puesto que se accederá a la misma a través de la API JDBC. Esta API enviará las peticiones al Driver, que será, quién en último término, acceda a la base de datos.

java.sql.Connection

Para comunicar con una base de datos utilizando JDBC, debemos en primer lugar establecer una conexión con la base de datos a través del driver JDBC apropiado. El API JDBC especifica la conexión en la `interfaz java.sql.Connection`.

La interfaz `java.sql.Connection` representa una conexión con una base de datos. Es una interfaz porque la implementación de una conexión depende de

la red, del protocolo y del vendedor. El API JDBC ofrece dos vías diferentes para obtener conexiones. La primera utiliza la clase `java.sql.DriverManager` y es adecuada para acceder a bases de datos desde programas cliente escritos en Java. El segundo enfoque se basa en el acceso a bases de datos desde aplicaciones J2EE (Java 2 Enterprise Edition).

El siguiente código muestra un ejemplo de conexión JDBC a una base de datos MySQL:

```
Connection connection;
String url              = "jdbc:mysql:Bank";
String login            = "root";
String password         = "nadiuska";

try {
     connection = DriverManager
          .getConnection(url, login, password);

     // Acceso a datos utilizando el objeto de conexión
     ...
} catch (SQLException sqle) {
     // Tratar el error aquí
} finally {
     try {
          connection.close();
     } catch (SQLException e) {
          // Tratar el error aquí
     }
}
En este ejemplo, la clase DriverManager intenta
establecer una conexión con la base de datos Bank
utilizando el driver JDBC que proporciona MySQL. Para
poder acceder al RDBMS MySQL es necesario introducir un
login y un password válidos.
```

En el API JDBC, hay varios métodos que pueden lanzar la excepción `SQLException`. En este ejemplo, la conexión se cierra al final del bloque finally, de modo que los recursos del sistema puedan ser liberados independientemente del éxito o fracaso de cualquier operación de base de datos.

java.sql.Statement

Antes de poder ejecutar una sentencia SQL, es necesario obtener un objeto de tipo `Statement`. Una vez creado dicho objeto, podrá ser utilizado para ejecutar cualquier operación contra la base de datos.

El siguiente método crea un objeto `Statement`, que podemos utilizar para enviar instrucciones SQL a la base de datos.

```
Statement Connection.createStatement() throws
SQLException
```

La finalidad de un objeto `Statement` es ejecutar una instrucción SQL que puede o no devolver resultados. Para ello, la interfaz `Statement` dispone de los siguiente métodos:

- `executeQuery()`, para sentencias SQL que recuperen datos de un único objeto ResultSet.
- `executeUpdate()`, para realizar actualizaciones que no devuelvan un `ResultSet`. Por ejemplo, sentencias DML SQL (Data Manipulation Language) como `INSERT`, `UPDATE` y `DELETE`, o sentencias DDL SQL (Data Definition Language) como `CREATE TABLE`, `DROP TABLE` y `ALTER TABLE`. El valor que devuelve `executeUpdate()` es un entero (conocido como la cantidad de actualizaciones) que indica el número de filas que se vieron afectadas. Las sentencias que no operan en filas, como `CREATE TABLE` o `DROP TABLE`, devuelven el valor cero.

java.sql.ResultSet

El objeto `Statement` devuelve un objeto `java.sql.ResultSet` que encápsula los resultados de la ejecución de una sentencia `SELECT`. Ésta interfaz es implementada por los vendedores de drivers. Dispone de métodos que permiten al usuario navegar por los diferentes registros que se obtienen como resultado de la consulta.

El siguiente método, `executeQuery`, definido en la interfaz `java.sql.Statement` le permite ejecutar las instrucciones `SELECT`:

```
public ResultSet executeQuery (String sql) throws
SQLException
```

La interfaz `java.sql.ResultSet` ofrece varios métodos para recuperar los datos que se obtienen de realizar una consulta:

```
getBoolean()
getInt()
getShort()
getByte()
getDate()
getDouble()
getfloat()
```

Todos estos métodos requieren el nombre de la columna (tipo `String`) o el índice de la columna (tipo `int`) como argumento. La sintaxis para las dos variantes del método `getString()` es la siguiente:

```
public String getString(int columnIndex) throws
SQLException
public String getString(String columnName) throws
SQLException
```

La interfaz `ResultSet` también permite conocer la estructura del bloque de resultados. El método `getMetaData()` ayuda a recuperar un objeto `java.sql.ResultSetMetaData` que tiene varios métodos para describir el bloque de resultados, algunos de los cuales se enumeran a continuación:

```
getTableName()
getColumnCount()
getColumnName()
getColumnType()
```

Tomando un bloque de resultados, podemos utilizar el método `getColumnCount()` para obtener el número de columnas de dicho bloque. Conocido el número de columnas, podemos obtener la información de tipo asociada a cada una de ellas.

Los tipos de columna son devueltos como números enteros. Por ejemplo, todas las columnas de tipo `VARCHAR` retornarán el entero 12, las del tipo `DATE`, 91. Estos tipos son constantes definidas en la interfaz `java.sql.Types`.

Transacciones

Si hay una propiedad que distingue una base de datos de un sistema de archivos, esa propiedad es la capacidad de soportar transacciones. Si está

escribiendo en un archivo y el sistema operativo cae, es probable que el archivo se corrompa. Si está escribiendo en un archivo de base de datos, utilizando correctamente las transacciones, se asegura que, o bien el proceso se completará con éxito, o bien la base de datos volverá al estado en el que se encontraba antes de comenzar a escribir en ella.

Cuando múltiples instrucciones son ejecutadas en una única transacción, todas las operaciones pueden ser realizadas (convertidas en permanentes en la base de datos) o descartadas (es decir, se deshacen los cambios aplicados a la base de datos).

Cuando se crea un objeto `Connection`, éste está configurado para realizar automáticamente cada transacción. Esto significa que cada vez que se ejecuta una instrucción, se realiza en la base de datos y no puede ser deshecha. Los siguientes métodos en la interfaz `Connection` son utilizados para gestionar las transacciones en la base de datos:

```
void setAutoCommit(boolean autoCommit) throws
SQLException
void commit() throws SQLException
void rollback() throws SQLException
```

Para iniciar una transacción, invocamos `setAutoCommit(false)`. Esto nos otorga el control sobre lo que se realiza y cuándo se realiza. Una llamada al método `commit()` realizará todas las instrucciones emitidas desde la última vez que se invocó el método `commit()`. Por el contrario, una llamada `rollback()` deshará todos los cambios realizados desde el último `commit()`. Sin embargo, una vez se ha emitido una instrucción `commit()`, esas transacciones no pueden deshacerse con `rollback()`.

java.sql.DataBaseMetaData y java.sql.ResultSetMetaData

[11] En ocasiones necesitamos hacer aplicaciones en Java que trabajen contra una base de datos desconocida, de la que no sabemos qué tablas tiene, ni qué columnas cada tabla. Es, por ejemplo, el caso de aplicaciones que permiten visualizar el contenido (o incluso modificar el mismo) de una base de datos cualquiera de una forma genérica, precisamente el caso en el que nos encontramos en este trabajo de fin de máster.

En java, las clases `DataBaseMetaData` y `ResultSetMetaData` permiten, respectivamente, analizar la estructura de una base de datos (qué tablas tiene, que columnas cada tabla, de qué tipos, etc) o de un `ResultSet` de una consulta, para averiguar cuántas columnas tiene dicho `ResultSet`, de qué columnas de base de datos proceden, de qué tipo son, etc.

No vamos a detallar toda la posible información que se puede obtener a través de estas dos clases ni a ser demasiado exhaustivos.

java.sql.DataBaseMetaData. Para ver qué catálogos, esquemas, tablas, tipos de tablas, columnas de las tablas y demás de una base de datos, tenemos la clase `DataBaseMetaData`. Una vez establecida la conexión, podemos obtener la instancia correspondiente de dicha clase con el método `getDataBaseMetaData()` de la `Connection`,

```
DataBaseMetaData metaDatos =
conexion.getMetaData();
```

En `metaDatos` tendremos la instancia de `DataBaseMetaData`.

Mirando en la API de `DataBaseMetaData` veremos que hay montones de métodos para interrogar a la base de datos sobre montones de cosas:

```
ResultSet rs = metaDatos.getTables(null, null, "%",
null);
```

En SQL el caracter que indica "todo" es %, equivalente al * a la hora de listar ficheros. Esto nos dará todas las tablas del catálogo y esquema actual.

En el `ResultSet` obtenido tendremos una fila por cada tabla que cumpla los patrones que hemos puesto en los parámetros. Las columnas de ese `ResultSet` podemos verlas en la API del método `DataBaseMetaData.getTables()`.

Un posible siguiente paso sería, para cada tabla, obtener las columnas que la componen: su nombre y tipo. Para ello, podemos usar el método `getColumns()` de `DataBaseMetaData`.

```
ResultSet rs = metaDatos.getColumns(catalogo, null,
tabla, null);
```

El contenido del `ResultSet` será una fila por cada columna de la tabla. Las columnas del `ResultSet` se pueden ver en la API del método `DataBaseMetaData.getColumns()`.

Por supuesto, hay muchos más métodos en `DataBaseMetaData` para obtener muchas más cosas sobre la base de datos. Gran parte del trabajo se basa en la ejecución de estos métodos y la observación de los resultados obtenidos.

ResultSetMetaData. Si no tenemos muy claro qué campos nos devuelve una consulta (quizás porque la consulta la ha escrito directamente el usuario desde una caja de texto), o queremos hacer algún trozo de código general capaz, por ejemplo, de meter cualquier `ResultSet` en un `<table> html`, nos resultará de utilidad la clase `ResultSetMetaData`.

```
Statement st = conexion.createStatement();
ResultSet rs = st.executeQuery("select * from
person");
```

A partir del `ResultSet`, podemos obtener el `ResultSetMetaData`, al que podremos interrogar sobre lo que queramos.

```
ResultSetMetaData rsmd = rs.getMetaData();
```

Vamos a preguntarle, primero, cuántas columnas tiene el `ResultSet`. Y luego, en un bucle, preguntaremos por el nombre de cada columna, a qué tabla pertenece y de qué tipo es. El bucle puede ser como este

```
for (int i = 1; i <= numeroColumnas; i++) {
   System.out.println("columna=" +
rsmd.getTableName(i) + "."
      + rsmd.getColumnName(i) + " --> "
      + rsmd.getColumnTypeName(i));
}
```

y podría darnos un resultado similar a éste

```
columna=person.PERSON_ID --> BIGINT
columna=person.age --> INT
columna=person.firstname --> VARCHAR
columna=person.lastname --> VARCHAR
```

Técnicas y herramientas

4.1. Apache Tomcat

[12] Apache Tomcat (también llamado Jakarta Tomcat o simplemente Tomcat) funciona como un contenedor de servlets desarrollado bajo el proyecto Jakarta en la Apache Software Foundation. Tomcat implementa las especificaciones de los servlets y de JavaServer Pages (JSP) de Oracle Corporation (aunque creado por Sun Microsystems).

Tomcat es desarrollado y actualizado por miembros de la Apache Software Foundation y voluntarios independientes. Los usuarios disponen de libre acceso a su código fuente y a su forma binaria en los términos establecidos en la Apache Software License. Las primeras distribuciones de Tomcat fueron las versiones 3.0.x. A partir de la versión 4.0, Jakarta Tomcat utiliza el contenedor de servlets Catalina.

Las versiones más recientes son las 9.x, que implementan las especificaciones de Servlet 4.0 y de JSP 2.3.

Tomcat es un contenedor web con soporte de servlets y JSPs. Tomcat no es un servidor de aplicaciones, como JBoss o JOnAS. Incluye el compilador Jasper, que compila JSPs convirtiéndolas en servlets. El motor de servlets de Tomcat a menudo se presenta en combinación con el servidor web Apache.

Tomcat puede funcionar como servidor web por sí mismo. En sus inicios existió la percepción de que el uso de Tomcat de forma autónoma era sólo recomendable para entornos de desarrollo y entornos con requisitos mínimos de velocidad y gestión de transacciones. Hoy en día ya no existe esa percepción y

Tomcat es usado como servidor web autónomo en entornos con alto nivel de tráfico y alta disponibilidad.

Dado que Tomcat fue escrito en Java, funciona en cualquier sistema operativo que disponga de la máquina virtual Java.

La jerarquía de directorios de instalación de Tomcat incluye:

- bin - arranque, cierre, y otros scripts y ejecutables.
- common - clases comunes que pueden utilizar Catalina y las aplicaciones web.
- conf - ficheros XML y los correspondientes DTD para la configuración de Tomcat.
- logs - logs de Catalina y de las aplicaciones.
- server - clases utilizadas solamente por Catalina.
- shared - clases compartidas por todas las aplicaciones web.
- webapps - directorio que contiene las aplicaciones web.
- work - almacenamiento temporal de ficheros y directorios.

Configuración de tomcat

[13] Introducción

Hay varias formas de configurar Tomcat para que se ejecute en diferentes plataformas. La documentación principal para esto es un archivo llamado `RUNNING.txt`. Se recomienda consultar ese archivo para más detalles.

Windows

La instalación de Tomcat en Windows se puede hacer fácilmente usando el instalador de Windows. Su interfaz y funcionalidad es similar a la de otros instaladores basados en asistentes, con solo algunos elementos de interés.

- **Instalación como servicio**: Tomcat se instalará como un servicio de Windows sin importar qué configuración se seleccione. El uso de la casilla de verificación en la página de componentes establece el servicio como inicio "automático", de modo que Tomcat se inicia automáticamente cuando se inicia Windows. Para una seguridad óptima,

el servicio debe ejecutarse como un usuario separado, con permisos reducidos.

- **Ubicación de Java**: el instalador proporcionará un JRE predeterminado que se utilizará para ejecutar el servicio. El instalador utiliza el registro para determinar la ruta base de un JRE Java 6 o posterior, incluido el JRE instalado como parte del JDK completo. Cuando se ejecuta en un sistema operativo de 64 bits, el instalador buscará primero un JRE de 64 bits y solo buscará un JRE de 32 bits si no se encuentra un JRE de 64 bits. No es obligatorio utilizar el JRE predeterminado detectado por el instalador. Se puede usar cualquier JRE de Java 6 o posterior instalado (32 bits o 64 bits).
- **Icono de bandeja**: cuando Tomcat se ejecuta como un servicio, no habrá ningún icono de bandeja presente cuando se ejecuta Tomcat. Tenga en cuenta que al elegir ejecutar Tomcat al final de la instalación, el icono de la bandeja se utilizará incluso si Tomcat se instaló como un servicio.
- **Valores predeterminados**: los valores predeterminados utilizados por el instalador pueden anularse mediante el uso del argumento de línea de comandos `/C=<config file>`. El archivo de configuración utiliza el formato `name=value` con cada par en una línea separada. Los nombres de las opciones de configuración disponibles son:
    - `JavaHome`
    - `TomcatPortShutdown`
    - `TomcatPortHttp`
    - `TomcatPortAjp`
    - `TomcatMenuEntriesEnable`
    - `TomcatShortcutAllUsers`
    - `TomcatServiceDefaultName`
    - `TomcatServiceName`
    - `TomcatServiceFileName`
    - `TomcatServiceManagerFileName`
    - `TomcatAdminEnable`
    - `TomcatAdminUsername`
    - `TomcatAdminPassword`
    - `TomcatAdminRoles`

El instalador creará accesos directos que le permitirán iniciar y configurar Tomcat. Es importante tener en cuenta que la aplicación web de administración de Tomcat solo se puede utilizar cuando Tomcat se está ejecutando.

Unix daemon

Tomcat se puede ejecutar como un daemon utilizando la herramienta jsvc del proyecto commons-daemon. Los archivos fuente de archivos comprimidos para jsvc se incluyen con los binarios de Tomcat y deben compilarse. La construcción de jsvc requiere un compilador C ANSI (como GCC), GNU Autoconf y un JDK.

Antes de ejecutar el script, la variable de entorno `JAVA_HOME` debe establecerse en la ruta base del JDK. Alternativamente, al llamar al `./configurescript`, la ruta del JDK puede especificarse usando el parámetro `--with-java`, como `./configure --with-java=/usr/java`.

El uso de los siguientes comandos debe dar como resultado un binario jsvc compilado, ubicado en la carpeta `$CATALINA_HOME/bin`. Esto supone que se utiliza GNU TAR y que `CATALINA_HOME` es una variable de entorno que apunta a la ruta base de la instalación de Tomcat.

Tenga en cuenta que debe usar GNU make (`gmake`) en lugar de la marca BSD nativa en los sistemas FreeBSD.

Es posible que también deba especificar `-jvm server` si la JVM utiliza de forma predeterminada una VM de servidor en lugar de una VM de cliente.

Se permite ejecutar Tomcat como un usuario no privilegiado y al mismo tiempo ser capaz de usar puertos privilegiados. Tenga en cuenta que si usa esta opción e inicia Tomcat como root, deberá deshabilitar la verificación `org.apache.catalina.security.SecurityListener` que impide que Tomcat se inicie cuando se ejecute como root.

El archivo `$CATALINA_HOME/bin/daemon.sh` se puede usar como plantilla para iniciar Tomcat automáticamente en el momento del arranque desde `/etc/init.d` con jsvc.

Tenga en cuenta que el archivo JAR de Commons-Daemon debe estar en su classpath de tiempo de ejecución para ejecutar Tomcat de esta manera. El archivo JAR de Commons-Daemon se encuentra en la entrada Class-Path del archivo `bootstrap.jar`, pero si obtiene una `ClassNotFoundException` o

`NoClassDefFoundError` para una clase de Commons-Daemon, agregue el Commons-Daemon JAR al argumento -cp al iniciar jsvc .

Implementación de la aplicación web de Tomcat

[14] Implementación es el término utilizado para el proceso de instalación de una aplicación web en el servidor Tomcat.

La implementación de la aplicación web se puede realizar de varias maneras dentro del servidor Tomcat.

- Inactivamente: la aplicación web está configurada antes de que se inicie Tomcat
- Dinámicamente: manipulando directamente las aplicaciones web ya implementadas o de forma remota mediante la aplicación web Tomcat Manager

El `Tomcat Manager` es una aplicación web que puede ser utilizado de forma interactiva (a través de interfaz gráfica de usuario HTML) o mediante programación (a través de la API basada en URL) para desplegar y gestionar aplicaciones web.

Hay varias formas de realizar la implementación que dependen de la aplicación web de Manager. Apache Tomcat proporciona tareas para la herramienta de construcción Apache Ant. El proyecto `Apache Tomcat Maven Plugin` proporciona integración con Apache Maven. También hay una herramienta llamada `Client Deployer`, que se puede usar desde una línea de comandos y proporciona una funcionalidad adicional como compilar y validar aplicaciones web, así como empaquetar aplicaciones web en archivos de recursos de aplicaciones web (WAR).

Instalación

No se requiere ninguna instalación para la implementación estática de aplicaciones web, ya que Tomcat la proporciona de forma inmediata. Tampoco se requiere ninguna instalación para las funciones de implementación con Tomcat Manager, aunque se requiere alguna configuración. Sin embargo, se requiere una instalación si desea utilizar el Tomcat Client Deployer (TCD).

El TCD no está empaquetado con la distribución central de Tomcat y, por lo tanto, debe descargarse por separado desde el área de Descargas. La descarga se suele etiquetar como apache-tomcat-7.0.x-deployer .

Queda fuera del ámbito de este trabajo la instalación y utilización del TCD.

Contextos

Al hablar sobre la implementación de aplicaciones web, se necesita comprender el concepto de contexto. Un contexto es lo que Tomcat llama una aplicación web.

Para configurar un contexto dentro de Tomcat se requiere un descriptor de contexto. Un descriptor de contexto es simplemente un archivo XML que contiene la configuración relacionada con Tomcat para un contexto, por ejemplo, recursos de nombres o la configuración del administrador de sesión. En versiones anteriores de Tomcat, el contenido de una configuración del Descriptor de Contexto a menudo se almacenaba en el archivo de configuración principal de Tomcat server.xml, pero ahora está desaconsejado (aunque actualmente todavía funciona).

Los Descriptores de contexto no solo ayudan a Tomcat a saber cómo configurar Contextos, sino que otras herramientas como Tomcat Manager y TCD a menudo utilizan estos Descriptores de contexto para realizar sus funciones correctamente.

Las ubicaciones de los descriptores de contexto son:

1. `$CATALINA_BASE/conf/[enginename]/[hostname]/[webappname].xml`
2. `$CATALINA_BASE/webapps/[nombre de la aplicación web]/META-INF/context.xml`

Los archivos en (1) se denominan `[webappname].xml`, pero los archivos en (2) se denominan `context.xml`. Si no se proporciona un Descriptor de contexto para un Contexto, Tomcat configura el Contexto usando valores predeterminados.

App Tomcat Manager

[15] En muchos entornos de producción, es muy útil tener la capacidad de implementar una nueva aplicación web o anular la implementación de una existente, sin tener que cerrar y reiniciar todo el contenedor. Además, puede solicitar que una aplicación existente se vuelva a cargar.

Para admitir estas capacidades, Tomcat incluye una aplicación web (instalada de forma predeterminada en la ruta de contexto `/manager`) que admite las siguientes funciones:

- Implementar una nueva aplicación web desde el contenido cargado de un archivo WAR.
- Implementar una nueva aplicación web, en una ruta de contexto específica, desde el sistema de archivos del servidor.
- Enumerar las aplicaciones web actualmente implementadas, así como las sesiones que están actualmente activas para esas aplicaciones web.
- Recargar una aplicación web existente, para reflejar los cambios en los contenidos de `/WEB-INF/classes/WEB-INF/lib`.
- Listar los valores de las propiedades del sistema operativo y JVM.
- Enumerar los recursos JNDI globales disponibles.
- Iniciar una aplicación detenida (volviéndola a estar disponible).
- Detener una aplicación existente (para que no esté disponible), sin desinstalarla.
- Despliegar una aplicación web implementada y eliminar su directorio de base de documentos (a menos que se haya implementado desde el sistema de archivos).

Una instalación predeterminada de Tomcat incluye el Tomcat Manager.

Si tiene Tomcat configurado para admitir varios hosts virtuales (sitios web), deberá configurar un Manager para cada uno.

Hay tres formas de usar la aplicación web Manager.

- Como una aplicación con una interfaz de usuario que utiliza en su navegador. He aquí un ejemplo de URL donde se puede sustituir localhost con su sitio web nombre de host: `http://localhost:8080/manager/html`.

- Una versión mínima que utiliza solo solicitudes HTTP que es adecuada para que la utilicen los administradores de sistemas con la configuración de scripts. Los comandos se dan como parte del URI de solicitud, y las respuestas son en forma de texto simple que se puede analizar y procesar fácilmente.
- Un conjunto conveniente de definiciones de tareas para la herramienta de compilación Ant (versión 1.4 o posterior).

Fuentes de datos JDBC

[16] A continuación, se presentan algunas configuraciones de ejemplo que se han publicado en tomcat-user para las bases de datos populares y algunos consejos generales para el uso de la base de datos.

DriverManager

`java.sql.DriverManager` soporta el mecanismo proveedor de servicios. Esta característica es que todos los controladores JDBC disponibles que se anuncian al proporcionar un archivo `META-INF/services/java.sql.Driver` se descubren, cargan y registran automáticamente, lo que le libera de la necesidad de cargar el controlador de la base de datos explícitamente antes de crear una conexión JDBC. Sin embargo, la implementación está rota fundamentalmente en todas las versiones de Java para un entorno de contenedor de servlet. El problema es que `java.sql.DriverManager` buscará los controladores una sola vez.

El JRE Memory Leak Prevention Listener que se incluye con Apache Tomcat resuelve esto activando el escaneo del controlador durante el inicio de Tomcat. Esto está habilitado por defecto. Esto significa que solo las bibliotecas visibles para el cargador de clases común y sus padres serán analizadas en busca de controladores de base de datos. Esto incluye los controladores en `$CATALINA_HOME/lib`, `$CATALINA_BASE/lib` la ruta de clase y (donde JRE lo admite) el directorio aprobado. Controladores empaquetados en aplicaciones web (en `WEB-INF/lib`) y en el cargador de clases compartidas (donde esté configurado) no estarán visibles y no se cargarán automáticamente. Si está considerando deshabilitar esta función, tenga en cuenta que la primera aplicación web que utiliza JDBC activará el análisis, lo que provocará fallas

cuando se vuelva a cargar esta aplicación web y para otras aplicaciones web que se basen en esta función.

Por lo tanto, las aplicaciones web que tienen controladores de base de datos en su directorio `WEB-INF/lib` no pueden confiar en el mecanismo del proveedor de servicios y deben registrar los controladores explícitamente.

La lista de controladores en `java.sql.DriverManager` también es una fuente conocida de fugas de memoria. Todos los controladores registrados por una aplicación web deben anular su registro cuando la aplicación web se detiene. Tomcat intentará descubrir y anular automáticamente el registro de cualquier controlador JDBC cargado por el cargador de clases de la aplicación web cuando la aplicación web se detenga. Sin embargo, se espera que las aplicaciones hagan esto por sí mismas a través de un archivo `ServletContextListener`.

Pool de conexiones a base de datos

La implementación del pool de conexiones de base de datos predeterminada en Apache Tomcat se basa en las bibliotecas del proyecto Apache Commons aunque en el presente trabajo no se utilizarán. Se utilizan las siguientes bibliotecas:

- `Commons DBCP 1.x`
- `Commons Pool 1.x`

Estas bibliotecas se encuentran en un solo JAR en `$CATALINA_HOME/lib/tomcat-dbcp.jar`. Sin embargo, solo se han incluido las clases necesarias para la agrupación de conexiones y se ha cambiado el nombre de los paquetes para evitar interferir con las aplicaciones.

DBCP 1.4 proporciona soporte para JDBC 4.0.

Problemas comunes

A continuación se indican algunos problemas comunes encontrados con una aplicación web que usa una base de datos y consejos para resolverlos

Fallos de conexión de base de datos intermitentes. Tomcat se ejecuta dentro de una JVM. La JVM realiza periódicamente la recolección de basura (GC) para eliminar objetos java que ya no se utilizan. Cuando la

JVM realiza la ejecución GC del código dentro de Tomcat se congela. Si el tiempo máximo configurado para el establecimiento de una conexión de base de datos es menor que el tiempo que tardó la recolección de basura, puede obtener un error de conexión de la base de datos.

Para recopilar datos sobre la duración de la recolección de basura, agregue el argumento `-verbose:gc` a su variable de entorno `CATALINA_OPTS` inicie Tomcat. Cuando gc verbose esté habilitado, su archivo de registro `$CATALINA_BASE/logs/catalina.out` incluirá datos para cada recolección de basura, incluido el tiempo que tomó.

Cuando su JVM está sintonizada correctamente el 99% del tiempo, un GC tardará menos de un segundo. El resto sólo tomará unos segundos. En raras ocasiones, es posible que alguna vez debería un GC tomar más de 10 segundos.

Asegúrese de que el tiempo de espera de conexión de la base de datos esté configurado en 10-15 segundos. Para el DBCP 1.x, establezca esto usando el parámetro `maxWait`

Conexión aleatoria, excepciones cerradas. Esto puede ocurrir cuando una solicitud obtiene una conexión de base de datos del grupo de conexiones y la cierra dos veces. Cuando se usa un grupo de conexiones, al cerrar la conexión solo se devuelve al grupo para que otra solicitud la reutilice, no cierra la conexión. Y Tomcat usa múltiples hilos para manejar las solicitudes concurrentes. Aquí hay un ejemplo de la secuencia de eventos que podrían causar este error en Tomcat:

- La solicitud 1 que se ejecuta en el subproceso 1 obtiene una conexión db.
- Solicitud 1 cierra la conexión db.
- La JVM cambia el hilo en ejecución a Thread 2
- La solicitud 2 que se ejecuta en Thread 2 obtiene una conexión db. (La misma conexión que se acaba de cerrar por Solicitud 1).
- La JVM cambia el hilo en ejecución de nuevo a Hilo 1

- La solicitud 1 cierra la conexión db por segunda vez en un bloque final.
- La JVM cambia el hilo en ejecución de nuevo a Thread 2
- Solicitud 2: El subproceso 2 intenta usar la conexión db pero falla porque la Solicitud 1 lo cerró.

Este es un ejemplo de código escrito correctamente para usar una conexión de base de datos:

```
Connection conn = null;
  try {
    conn = ... consigue la conexión ...
    ... realiza las operaciones con la conexion ...
    conn.close(); // Cierra la conexión
    conn = null;  // Asegurarse de que no se cierre
dos veces
  } catch (SQLException e) {
    ... tramiento de errores ...
  } finally {
    // Asegurese siempre de que la conexión se
cierra
    if (conn != null) {
      try { conn.close(); } catch (SQLException e)
{ ; }
      conn = null;
    }
  }
```

Context vs GlobalNamingResources. Tener en cuenta que aunque las instrucciones anteriores colocan las declaraciones JNDI en un elemento de contexto, es posible y algunas veces deseable colocar estas declaraciones en la sección GlobalNamingResources del archivo de configuración del servidor. Un recurso ubicado en la sección GlobalNamingResources se compartirá entre los Contextos del servidor.

El cargador de clases

[17] Al igual que muchas aplicaciones de servidor, Tomcat instala una variedad de cargadores de clases (es decir, clases que implementan

`java.lang.ClassLoader`) para permitir que diferentes porciones del contenedor, y las aplicaciones web que se ejecutan en el contenedor, tengan acceso a diferentes repositorios de clases y recursos disponibles. Este mecanismo se utiliza para proporcionar la funcionalidad definida en la Especificación de Servlet, versión 2.4.

En un entorno Java, los cargadores de clases se organizan en un árbol padre-hijo. Normalmente, cuando se le pide a un cargador de clases que cargue una clase o recurso en particular, primero delega la solicitud a un cargador de clases principal, y luego busca en sus propios repositorios solo si los cargadores de clases principales no pueden encontrar la clase o recurso solicitado. Tener en cuenta que el modelo para los cargadores de clases de aplicaciones web difiere ligeramente de este, como se explica a continuación, pero los principios fundamentales son los mismos.

Cuando se inicia Tomcat, crea un conjunto de cargadores de clases que se organizan en las siguientes relaciones padre-hijo, donde el cargador de clase padre está por encima del cargador de clase hijo:

```
   Bootstrap
      |
    System
      |
    Common
    /     \
Webapp1  Webapp2 ...
```

Las características de cada uno de estos cargadores de clases, incluida la fuente de las clases y los recursos que hacen visibles, se describen someramente en la siguiente sección.

Definiciones del cargador de clases

Como se indica en el diagrama anterior, Tomcat crea los siguientes cargadores de clases a medida que se inicializan:

- **Bootstrap**: este cargador de clases contiene las clases de tiempo de ejecución básicas proporcionadas por la Máquina Virtual de Java, más

cualquier clase de archivos JAR presentes en el directorio de Extensiones del Sistema (`$JAVA_HOME/jre/lib/ext`).

- **Sistema**: este cargador de clases normalmente se inicializa desde el contenido de la variable de entorno `CLASSPATH`. Todas estas clases son visibles tanto para las clases internas de Tomcat como para las aplicaciones web. Sin embargo, los scripts de inicio estándar de Tomcat ignoran totalmente el contenido de la propia variable de entorno `CLASSPATH` y, en su lugar, crean el cargador de clases del sistema a partir de los siguientes repositorios:
    - `$CATALINA_HOME/bin/bootstrap.jar`: contiene el método main () que se utiliza para inicializar el servidor Tomcat y las clases de implementación del cargador de clases de las que depende.
    - `$CATALINA_BASE/bin/tomcat-juli.jar` Estas incluyen clases de mejora de API `java.util.logging`, conocidas como JULI de Tomcat, y una copia renombrada de paquete de la biblioteca de Apache Commons Logging utilizada internamente por Tomcat.
    - Si `tomcat-juli.jar` está presente en `$CATALINA_BASE/bin`, se usa en lugar de la que se encuentra en `$CATALINA_HOME/bin`. Es útil en ciertas configuraciones de registro.
    - `$CATALINA_HOME/bin/commons-daemon.jar` - Las clases del proyecto Apache Commons Daemon . Este archivo JAR no está presente en el CLASSPATH construido por catalina.bat o .sh scripts, pero se hace referencia desde el archivo de manifiesto de bootstrap.jar.
- **Common**: este cargador de clases contiene clases adicionales que se hacen visibles tanto para las clases internas de Tomcat como para todas las aplicaciones web.

  Normalmente, las clases de aplicación NO se deben colocar aquí. Las ubicaciones buscadas por este cargador de clases están definidas por la propiedad `common.loader` en `$CATALINA_BASE/conf/catalina.properties`. La configuración predeterminada buscará las siguientes ubicaciones en el orden en que aparecen:

    - Clases y recursos desempaquetados en `$CATALINA_BASE/lib`
    - Archivos JAR en `$CATALINA_BASE/lib`
    - Clases y recursos desempaquetados en `$CATALINA_HOME/lib`
    - Archivos JAR en `$CATALINA_HOME/lib`
- **WebappX**: se crea un cargador de clases para cada aplicación web que se implementa en una sola instancia de Tomcat. Todas las clases y recursos desempaquetados en el directorio `/WEB-INF/classes` de su aplicación web, más las clases y los recursos en archivos JAR en el directorio `/WEB-INF/lib` de su aplicación web, se hacen visibles para esta aplicación web, pero no para otras.

Como se mencionó anteriormente, el cargador de clases de aplicaciones web difiere del modelo de delegación de Java predeterminado. Cuando se procesa una solicitud para cargar una clase desde el cargador de clases WebappX de la aplicación web , este cargador de clases buscará primero en los repositorios locales, en lugar de delegar antes de mirar. Hay excepciones. Las clases que forman parte de las clases base de JRE no se pueden anular. Para algunas clases (como los componentes del analizador XML en J2SE 1.4+), la función respaldada por Java puede usarse hasta Java 8. Por último, cualquier archivo JAR que contenga clases de la API del Servlet será explícitamente ignorado por el cargador de clases. No incluir tales archivos JARs en la aplicación web. Todos los demás cargadores de clases en Tomcat siguen el patrón de delegación habitual.

Por lo tanto, desde la perspectiva de una aplicación web, la clase o la carga de recursos se ve en los siguientes repositorios, en este orden:

- Clases de arranque de la JVM.
- /WEB-INF/classes de la aplicación web
- /WEB-INF/lib/*.jar de la aplicación web
- Clases de cargador de clases del sistema (descritas anteriormente)
- Clases de cargador de clases comunes (descritas anteriormente)

Si el cargador de clases de aplicaciones web está configurado con `<Loader delegate="true"/>`, el orden se convierte en:

- Clases de arranque de la JVM.
- Clases de cargador de clases del sistema (descritas anteriormente)

- Clases de cargador de clases comunes (descritas anteriormente)
- /WEB-INF/clases de la aplicación web
- /WEB-INF/lib/*.jar de la aplicación web

También se puede configurar una jerarquía de cargador de clases más compleja. Vea el diagrama a continuación. De forma predeterminada, los cargadores de clase Servidor y Compartidos no están definidos y se utiliza la jerarquía simplificada que se muestra arriba. Esta jerarquía más compleja se puede usar al definir valores para las propiedades `server.loader` y `shared.loader` en `conf/catalina.properties`.

```
 Bootstrap
    |
   System
    |
   Common
   /      \
Server  Shared
           /    \
  Webapp1  Webapp2 ...
```

El cargador de clases del servidor solo es visible para las partes internas de Tomcat y es completamente invisible para las aplicaciones web.

El cargador de clases compartidas es visible para todas las aplicaciones web y se puede usar para compartir código en todas las aplicaciones web. Sin embargo, cualquier actualización de este código compartido requerirá un reinicio de Tomcat.

Conectores

[18] Elegir un conector para usar con Tomcat puede ser difícil. Esta página mostrará una lista de los conectores compatibles con esta versión de Tomcat y, con suerte, le ayudará a tomar la decisión correcta de acuerdo con las necesidades.

HTTP

El conector HTTP está configurado de forma predeterminada con Tomcat y está listo para usar. Este conector presenta la latencia más baja y el mejor rendimiento general.

Para la agrupación en clústeres, se debe instalar un equilibrador de carga HTTP con soporte para la adherencia de las sesiones web para dirigir el tráfico a los servidores Tomcat. Tomcat admite mod_proxy (en Apache HTTP Server 2.x, e incluido de forma predeterminada en Apache HTTP Server 2.2) como el equilibrador de carga. Cabe señalar que el rendimiento del proxy HTTP suele ser inferior al rendimiento de AJP, por lo que a menudo es preferible la agrupación de AJP.

AJP

Cuando se usa un solo servidor, el rendimiento es en la mayoría de los casos significativamente peor que en un Tomcat independiente con su conector HTTP predeterminado, incluso si una gran parte de la aplicación web está hecha de archivos estáticos. Si se necesita la integración con el servidor web nativo por cualquier motivo, un conector AJP proporcionará un rendimiento más rápido que el HTTP proxy. La agrupación AJP es la más eficiente desde la perspectiva de Tomcat. De lo contrario, es funcionalmente equivalente al agrupamiento en clúster HTTP.

Los conectores nativos compatibles con esta versión de Tomcat son:

- JK 1.2.x con cualquiera de los servidores soportados
- mod_proxy en Apache HTTP Server 2.x (incluido de forma predeterminada en Apache HTTP Server 2.2), con AJP habilitado

Otros conectores nativos compatibles con AJP pueden funcionar, pero ya no son compatibles

Consideraciones de seguridad

[19] Tomcat está configurado para ser razonablemente seguro para la mayoría de los casos de uso de forma predeterminada. Algunos entornos pueden requerir más o menos configuraciones seguras. Este apartado proporciona un

punto de referencia para las opciones de configuración que pueden afectar la seguridad y ofrece algunos comentarios sobre el impacto esperado de cambiar esas opciones. La intención es proporcionar una lista de opciones de configuración que deben considerarse al evaluar la seguridad de una instalación de Tomcat.

Configuraciones no Tomcat

La configuración de Tomcat no debe ser la única línea de defensa. Los otros componentes del sistema (sistema operativo, red, base de datos, etc.) también deben estar protegidos.

Tomcat no debe ejecutarse bajo el usuario root. Cree un usuario dedicado para el proceso Tomcat y proporcione a ese usuario los permisos mínimos necesarios para el sistema operativo. Por ejemplo, no debería ser posible iniciar sesión de forma remota utilizando el usuario de Tomcat.

Los permisos de archivos también deben estar adecuadamente restringidos. La configuración estándar es que todos los archivos de Tomcat sean propiedad de root con el grupo Tomcat y mientras el propietario tenga privilegios de lectura / escritura , el grupo solo tiene lectura y el mundo no tiene permisos. Las excepciones son los registros, temp y directorio de trabajo que son propiedad del usuario Tomcat en lugar de la raíz. Esto significa que incluso si un atacante compromete el proceso de Tomcat, no puede cambiar la configuración de Tomcat, implementar nuevas aplicaciones web o modificar las aplicaciones web existentes. El proceso de Tomcat se ejecuta con una umask de 007 para mantener estos permisos.

En el nivel de red, considerar usar un firewall para limitar las conexiones entrantes y salientes a solo aquellas conexiones que se espera que estén presentes.

Aplicaciones web por defecto

Tomcat se envía con una serie de aplicaciones web que están habilitadas de forma predeterminada. Las vulnerabilidades se han descubierto en estas aplicaciones en el pasado. Las aplicaciones que no son necesarias deben eliminarse para que el sistema no corra riesgos si se descubre otra vulnerabilidad.

RAÍZ. La aplicación web ROOT presenta un riesgo de seguridad muy bajo, pero incluye la versión de Tomcat que se está utilizando. Normalmente, la aplicación web ROOT se debe eliminar de una instancia de Tomcat de acceso público, no por razones de seguridad, sino para que se muestre a los usuarios una página predeterminada más apropiada.

Documentation. La aplicación web de documentación presenta un riesgo de seguridad muy bajo, pero identifica la versión de Tomcat que se está utilizando. Normalmente debería eliminarse de una instancia de Tomcat de acceso público.

Examples. La aplicación web de ejemplos siempre debe eliminarse de cualquier instalación sensible a la seguridad. Si bien la aplicación web de ejemplos no contiene ninguna vulnerabilidad conocida, se sabe que contiene características que un atacante puede usar junto con una vulnerabilidad en otra aplicación implementada en la instancia de Tomcat para obtener información adicional que de lo contrario no estaría disponible.

Manager. La aplicación Manager permite el despliegue remoto de aplicaciones web y es atacada frecuentemente por los atacantes debido al uso generalizado de contraseñas débiles e instancias de Tomcat de acceso público con la aplicación Manager habilitada. La aplicación Manager no es accesible de forma predeterminada ya que ningún usuario está configurado con el acceso necesario. Si la aplicación Manager está habilitada, debe seguirse la guía en la sección Protección de aplicaciones de administración.

Host Manager. La aplicación Host Manager permite la creación y administración de hosts virtuales, incluida la habilitación de la aplicación Manager para un host virtual. La aplicación Host Manager no es accesible de forma predeterminada ya que ningún usuario está configurado con el acceso necesario. Si la aplicación Host Manager está habilitada, debe seguirse la guía en la sección Protección de aplicaciones de administración.

Asegurando aplicaciones de gestión. Al implementar una aplicación web que proporciona funciones de administración para la instancia de Tomcat, se deben seguir las siguientes pautas:

Asegúrese de que todos los usuarios autorizados para acceder a la aplicación de administración tengan contraseñas seguras.

No elimine el uso de `LockOutRealm` que evita los ataques de fuerza bruta contra las contraseñas de los usuarios.

Descomente el `RemoteAddrValve` en el `/META-INF/context.xml` que limita el acceso a localhost. Si se requiere acceso remoto, limítelo a direcciones IP específicas usando esta válvula.

Security manager

Al habilitar el administrador de seguridad, las aplicaciones web se ejecutan en un entorno limitado, lo que limita significativamente la capacidad de una aplicación web para realizar acciones maliciosas, como llamar a `System.exit()`, establecer conexiones de red o acceder al sistema de archivos fuera de los directorios raíz y temporales de la aplicación web. Sin embargo, debe tenerse en cuenta que hay algunas acciones maliciosas, como desencadenar un alto consumo de CPU a través de un bucle infinito, que el administrador de seguridad no puede evitar.

Habilitar el administrador de seguridad generalmente se hace para limitar el impacto potencial, en caso de que un atacante encuentre una manera de comprometer una aplicación web de confianza. También se puede usar un administrador de seguridad para reducir los riesgos de ejecutar aplicaciones web que no son de confianza (por ejemplo, en entornos de alojamiento), pero se debe tener en cuenta que el administrador de seguridad solo reduce los riesgos de ejecutar aplicaciones web que no son de confianza, no las elimina. Si ejecuta múltiples aplicaciones web que no son de confianza, se recomienda que cada aplicación web se despliegue en una instancia de Tomcat separada (y, idealmente, en hosts separados) para reducir la capacidad de una aplicación web maliciosa que afecta la disponibilidad de otras aplicaciones.

Tomcat se prueba con el administrador de seguridad habilitado; pero la mayoría de los usuarios de Tomcat no se ejecutan con un administrador de seguridad, por lo que Tomcat no está tan bien probado en esta configuración. Se han producido y continúan existiendo errores informados que se activan al ejecutarse bajo un administrador de seguridad.

Es probable que las restricciones impuestas por un administrador de seguridad rompan la mayoría de las aplicaciones si el administrador de seguridad está habilitado. El administrador de seguridad no debe utilizarse sin pruebas exhaustivas. Idealmente, el uso de un administrador de seguridad debe introducirse al inicio del ciclo de desarrollo, ya que puede llevar mucho tiempo rastrear y solucionar los problemas causados por la habilitación de un administrador de seguridad para una aplicación madura.

Al habilitar el administrador de seguridad cambia los valores predeterminados para las siguientes configuraciones:

El valor predeterminado para el atributo `deployXML` del elemento Host se cambia a false.

server.xml

El server.xml predeterminado contiene una gran cantidad de comentarios, incluidas algunas definiciones de componentes de ejemplo que están comentadas. La eliminación de estos comentarios facilita considerablemente la lectura y la comprensión de server.xml.

Si un tipo de componente no está en la lista, entonces no hay configuraciones para ese tipo que afecten directamente la seguridad.

Server. Establecer el atributo `port` a -1 deshabilita el puerto de apagado.

Si el puerto de apagado no está deshabilitado, se debe configurar una contraseña segura para el cierre.

Connectors. Por defecto, se configuran un conector HTTP y un conector AJP. Los conectores que no se utilizarán deben eliminarse de server.xml. A continuación, se consideran solo los más relevantes de cara al trabajo actual.

El atributo `address` se puede usar para controlar en qué dirección IP el conector escucha las conexiones. De forma predeterminada, el conector escucha en todas las direcciones IP configuradas.

Los atributos `SSLEnabled`, `scheme` y `secure` pueden ser establecidos independientemente. Normalmente se utilizan cuando

Tomcat se encuentra detrás de un proxy inverso y el proxy se conecta a Tomcat a través de HTTP o HTTPS. Permiten que Tomcat vea los atributos SSL de las conexiones entre el cliente y el proxy en lugar del proxy y Tomcat. Por ejemplo, el cliente puede conectarse al proxy a través de HTTPS, pero el proxy se conecta a Tomcat mediante HTTP. Si es necesario que Tomcat pueda distinguir entre conexiones seguras y no seguras recibidas por un proxy, el proxy debe usar conectores separados para pasar las solicitudes seguras y no seguras a Tomcat.

Host. El elemento host controla la implementación. La implementación automática permite una administración más sencilla, pero también facilita la implementación de una aplicación maliciosa por parte de un atacante. La implementación automática está controlada por los atributos `autoDeploy` y `deployOnStartup`. Si ambos son `false`, solo se desplegarán los Contextos definidos en server.xml y cualquier cambio requerirá un reinicio de Tomcat.

Context. Esto se aplica a los elementos de contexto en todos los lugares donde se pueden definir: archivo `server.xml`, `context.xml`, por host `context.xml.default`, archivo de contexto de la aplicación web en el directorio de configuración por host o dentro de la aplicación web.

Valves. Se recomienda encarecidamente que se configure un AccessLogValve. La configuración predeterminada de Tomcat incluye un `AccessLogValve`. Normalmente, estos se configuran por host, pero también se pueden configurar por motor o por contexto, según sea necesario.

Cualquier aplicación administrativa debe estar protegida por un `RemoteAddrValve`. El atributo `allow` debe utilizarse para limitar el acceso a un conjunto de hosts de confianza conocidos.

El `ErrorReportValve` predeterminado incluye el número de versión de Tomcat en la respuesta enviada a los clientes. Para evitar esto, se puede configurar el manejo personalizado de errores dentro de cada aplicación web. Alternativamente, puede configurar explícitamente un

`ErrorReportValve` y establecer su atributo `showServerInfo` en `false`.

Realms. El `MemoryRealm` no está diseñado para su uso en producción ya que cualquier cambio en tomcat-users.xml requiere un reinicio de Tomcat para que tenga efecto.

El `JDBCRealm` no se recomienda para uso de producción ya que es un solo hilo para todas las opciones de autenticación y autorización. Utilice el `DataSourceRealm` en su lugar.

`UserDatabaseRealm` no está diseñado para instalaciones a gran escala. Está destinado a entornos relativamente estáticos a pequeña escala.

Por defecto, los reinos no implementan ninguna forma de bloqueo de cuenta. Esto significa que los ataques de fuerza bruta pueden tener éxito. Para evitar un ataque de fuerza bruta, el reino elegido debe estar envuelto en un `LockOutRealm`.

Manager. El componente manager se utiliza para generar identificadores de sesión.

La clase utilizada para generar identificadores de sesión aleatorios se puede cambiar con el atributo `randomClass`.

La longitud de la ID de sesión puede cambiarse con el atributo `sessionIdLength`.

web.xml

Esto se aplica al archivo `conf/web.xml` y archivos predeterminados `WEB-INF/web.xml` en las aplicaciones web si definen los componentes mencionados aquí.

El DefaultServlet se configura con **readonly** establecido en `true`. Cambiar esto permite a los clientes eliminar o modificar recursos estáticos en el servidor y cargar nuevos recursos. Esto normalmente no se debe cambiar sin requerir autenticación.

El DefaultServlet se configura con los **listings** establecidos en `false`. Esto no se debe a que permitir la inclusión de listas de directorios sea inseguro, sino porque la generación de listas de directorios con miles de archivos puede consumir una CPU significativa que conduce a un ataque de DdS.

El DefaultServlet está configurado con **showServerInfo** configurado en `true`. Cuando los listados de directorios están habilitados, el número de versión de Tomcat se incluye en la respuesta enviada a los clientes. Para evitar esto, puede configurar explícitamente un DefaultServlet y establecer su atributo **showServerInfo** en `falso`.

El Servlet CGI está deshabilitado por defecto.

FailedRequestFilter se puede configurar y utilizar para rechazar solicitudes que tuvieron errores durante el análisis de parámetros de solicitud. Sin el filtro, el comportamiento predeterminado es ignorar parámetros no válidos o excesivos.

HttpHeaderSecurityFilter se puede usar para agregar encabezados a las respuestas para mejorar la seguridad. Si los clientes acceden directamente a Tomcat, es probable que desee habilitar este filtro y todos los encabezados que establece, a menos que su aplicación ya los esté configurando. Si se accede a Tomcat a través de un proxy inverso, entonces la configuración de este filtro debe coordinarse con los encabezados que establece el proxy inverso.

General

La autenticación BASIC y FORM pasan los nombres de usuario y las contraseñas en texto sin cifrar. Las aplicaciones web que utilizan estos mecanismos de autenticación con clientes que se conectan a través de redes que no son de confianza deben usar SSL.

La cookie de sesión para una sesión con un usuario autenticado es casi tan útil como la contraseña del usuario a un atacante y, en casi todas las circunstancias, debe tener el mismo nivel de protección que la propia contraseña. Por lo general, esto significa autenticarse a través de SSL y continuar usando SSL hasta que finalice la sesión.

4.2. Apache Struts 2

[20] Apache Struts 2 es un marco de aplicación web de código abierto para desarrollar aplicaciones web Java EE. Se utiliza y extiende el API Java Servlet para animar a los desarrolladores a adoptar una arquitectura basada en un modelo-vista-controlador (MVC). El marco framework se separó de Apache Struts con el objetivo de ofrecer mejoras al mismo tiempo que conserva la misma arquitectura general del marco Struts original. En diciembre de 2005, se anunció que el framework 2.2 se adoptaba como Apache Struts 2, que alcanzó su primer lanzamiento completo en febrero de 2007.

Las aplicaciones web basadas en JavaServer Pages a veces combinan código de base de datos, código de diseño de página y código de flujo de control. En la práctica, encontramos que a menos que estas preocupaciones se separen, las aplicaciones más grandes se vuelven difíciles de mantener.

Una forma de separar las preocupaciones en una aplicación de software es usar una arquitectura de Modelo-Vista-Controlador (MVC). El modelo representa el negocio o el código de la base de datos, la vista representa el código de diseño de la página y el controlador representa el código de navegación. El framework Struts está diseñado para ayudar a los desarrolladores a crear aplicaciones web que utilizan una arquitectura MVC.

El framework proporciona tres componentes clave:

1. Un controlador de "solicitud" proporcionado por el desarrollador de la aplicación que se asigna a un URI estándar.
2. Un controlador de "respuesta" que transfiere el control a otro recurso que completa la respuesta.
3. Una biblioteca de etiquetas que ayuda a los desarrolladores a crear aplicaciones interactivas basadas en formularios con páginas de servidor.

Tecnologías clave

[21] La documentación del framework está escrita para desarrolladores web activos y supone un conocimiento práctico sobre cómo se crean las aplicaciones web Java. Antes de comenzar, debemos comprender los conceptos básicos de varias tecnologías clave:

- HTTP y HTML
- El ciclo de solicitud / respuesta HTTP
- El lenguaje Java y los marcos de aplicación
- JavaScript, AJAX y SOAP
- Propiedades de archivos y ResourceBundles
- Servlets, filtros y contenedores web.
- Páginas JavaServer y bibliotecas de etiquetas JSP
- Lenguaje de marcado extensible (XML)
- JAAS
- Controlador de vista de modelo

Este punto define brevemente cada una de estas tecnologías, pero no las describe en detalle.

HTTP, HTML y agentes de usuario

La World Wide Web se creó a través del Protocolo de transferencia de hipertexto (HTTP) y el Lenguaje de marcado de hipertexto (HTML). Un agente de usuario, como un navegador web, utiliza HTTP para solicitar un documento HTML. El navegador entonces formatea y muestra el documento a su usuario. HTTP se utiliza para transportar más HTML, HTML desde el navegador para representar la vista.

El ciclo de solicitud / respuesta HTTP

Una parte muy importante de HTTP para el desarrollador web es el ciclo de solicitud / respuesta. Para usar HTTP tienes que hacer una solicitud. Un servidor HTTP, como un servidor web, está obligado a responder. Cuando construye su aplicación web, la diseña para que reaccione a una solicitud HTTP devolviendo una respuesta HTTP. Los frameworks a menudo abstraen gran parte de estas itnerioridades, pero es importante entender lo que está sucediendo detrás de la escena.

El lenguaje Java y los frameworks de aplicación

Struts está escrito en el lenguaje de programación Java. Java es un lenguaje orientado a objetos, y el framework hace un buen uso de muchas técnicas orientadas a objetos.

Reflection y Introspection. La reflexión es el proceso de determinar qué campos y métodos miembros están disponibles en un objeto. La introspección es una forma especializada de reflexión utilizada por la API de JavaBean. Mediante la introspección, podemos determinar a qué métodos de un objeto se pretende acceder mediante otros objetos. Los getters y los setters, por ejemplo.

Struts usa Introspection para convertir los parámetros HTTP en propiedades JavaBean y para rellenar los campos HTML de las propiedades JavaBean. Esta técnica hace que sea fácil "redondear" las propiedades entre los formularios HTML y JavaBeans.

Archivos Properties y ResourceBundles. Las aplicaciones Java, incluidas las aplicaciones web, a menudo se configuran utilizando archivos de propiedades. Los archivos de propiedades son la base de los ResourceBundles que utiliza el framework para proporcionar recursos de mensajes a una aplicación.

Los Java ResourceBundles usan uno o más archivos de propiedades para proporcionar mensajes internacionalizados a los usuarios según su configuración regional del navegador o sistema

Threads. Con Struts 1 se requería que se supiera mucho sobre cómo escribir código que pudiera ejecutarse en un entorno de múltiples subprocesos. Con Struts 2 esto ya no es necesario. En un entorno de Struts 2, cada acción causada por una solicitud HTTP es un objeto Java que se instancia para cada solicitud.

JavaScript, AJAX y SOAP

Con HTTP y HTML ya se pueden proporcionar páginas web estáticas. Hoy en día, con frecuencia esto ya no es suficiente y los usuarios de la aplicación pueden esperar una interfaz de usuario interactiva. Los desarrolladores web a menudo recurren a JavaScript para hacer las aplicaciones web más interesantes.

AJAX es una tecnología utilizada a menudo por los programadores de JavaScript para crear aplicaciones web que son tan interactivas y sensibles como las aplicaciones de escritorio. Es posible cargar partes de una vista del sitio web o solo los datos de la aplicación (Struts) en lugar de regenerar toda la vista.

Apache Struts proporciona complementos para trabajar fácilmente con AJAX e incluso con JavaScript. Donde Struts no puede proporcionar la funcionalidad necesaria, los terceros proporcionan extensiones para el comportamiento requerido.

Otra tecnología que puede mejorar el ciclo de solicitud / respuesta HTTP es SOAP. Usando SOAP, una aplicación puede acceder a los datos e invocar la lógica de negocios en otro servidor utilizando HTTP como capa de transferencia. El uso conjunto de AJAX y SOAP se está convirtiendo en una forma popular para que la página envíe solicitudes finamente integradas directamente a un servidor remoto, a la vez que mantiene una separación de preocupaciones entre la lógica empresarial y el marcado de la página. En nuestro caso, no necesitaremos hacer uso de la tecnología SOAP.

Servlets, filtros y contenedores web

Dado que Java es un lenguaje orientado a objetos, la plataforma Java Servlet se esfuerza por convertir HTTP en una forma orientada a objetos. Esta estrategia hace que sea más fácil para los desarrolladores de Java concentrarse en lo que necesitan que haga su aplicación, en lugar de la mecánica de HTTP.

Un servidor HTTP compatible con Java puede pasar una solicitud a un contenedor de servlets. El contenedor puede cumplir con la solicitud o puede pasar la solicitud al servidor HTTP. El contenedor decide si puede manejar la solicitud al verificar su lista de servlets. Si hay un servlet registrado para la solicitud, el contenedor pasa la solicitud al servlet.

Cuando llega una solicitud, el contenedor verifica si hay un servlet registrado para esa solicitud. Si hay una coincidencia, la solicitud se entrega al servlet. Si no, la solicitud se devuelve al servidor HTTP.

Es el trabajo del contenedor gestionar el ciclo de vida del servlet. El contenedor crea los servlets, invoca los servlets y, en última instancia, los desecha.

La antigua versión de Struts 1 se basaba en gran medida en los servlets y el buen conocimiento de la misma generalmente ayudaba mucho en el desarrollo de aplicaciones web.

Con Struts 2, solo necesitaremos un conocimiento básico de Servlets. Struts en realidad usa uno llamado `ServletFilter` para "hacer que las cosas funcionen". En general, no es necesario que escriba Servlets cuando use Struts 2. Todavía es útil saber qué son los Servlets, los Filtros y los Contenedores.

Los filtros le permiten componer un conjunto de componentes que procesarán una solicitud o respuesta. Los filtros se agregan en una cadena en la que cada filtro tiene la posibilidad de procesar la solicitud y la respuesta antes y después de que sea procesada por los filtros subsiguientes (y el servlet al que finalmente se llama).

Sesiones #{session}

Una de las características clave de HTTP es que es sin estado. En otras palabras, no hay nada integrado en HTTP que identifique una solicitud posterior del mismo usuario como relacionada con una solicitud previa de ese usuario. Esto hace que la creación de una aplicación que quiera entablar una conversación con el usuario a través de varias solicitudes sea algo difícil.

Para aliviar esta dificultad, la API servlet proporciona un concepto programático llamado `session`, representado como un objeto que implementa la interface `javax.servlet.http.HttpSession`. El contenedor de servlets utilizará una de dos técnicas (cookies o reescritura de URL) para garantizar que la próxima solicitud del mismo usuario incluya el ID de sesión para esta sesión, de modo que la información de estado guardada en la sesión pueda asociarse con varias solicitudes. Esta información de estado se almacena en atributos de sesión (en JSP, se conocen como "beans de alcance de sesión").

Para evitar ocupar recursos indefinidamente cuando un usuario no puede completar una interacción, las sesiones tienen un intervalo de tiempo de espera configurable. Si el intervalo de tiempo entre dos solicitudes supera este intervalo, se agotará el tiempo de espera de la sesión y se eliminarán todos los atributos de la sesión. Definir un tiempo de espera de sesión predeterminado en el descriptor de implementación de la aplicación web.

Es importante saber que los datos de la sesión ocupan con mayor frecuencia la memoria RAM de su servidor. Dependiendo de su contenedor, puede tener diferentes opciones para evitar esto.

Struts 2 proporciona formas fáciles de crear y acceder a una sesión.

Aplicaciones web

Al igual que un servidor HTTP se puede usar para alojar varios sitios web distintos, un contenedor de servlets se puede usar para hospedar más de una aplicación web. La plataforma de servlets de Java proporciona un mecanismo bien definido para organizar y desplegar aplicaciones web. Cada aplicación se ejecuta en su propio espacio de nombres para que puedan desarrollarse e implementarse por separado. Una aplicación web se puede ensamblar en un archivo de aplicación web o archivo WAR. El WAR único puede ser cargado en el servidor y desplegado automáticamente.

Seguridad

Un detalle que se puede configurar en el descriptor de implementación de la aplicación web es la seguridad administrada por contenedor. La seguridad declarativa se puede utilizar para proteger las solicitudes de URI que coincidan con determinados patrones. La seguridad pragmática se puede utilizar para ajustar la seguridad y tomar decisiones de autorización según la hora del día, los parámetros de una llamada o el estado interno de un componente web. También se puede utilizar para restringir la autenticación en función de la información en una base de datos.

JavaServer Pages, bibliotecas de etiquetas JSP y JavaServer Faces

Si escribe una aplicación web clásica, es posible que necesite un componente de visualización. Uno de los primeros de su tipo fue JSP.

Aunque aún son potentes y totalmente compatibles con Struts, las personas pueden preferir otras tecnologías como Velocity y Freemarker. Ambos son también ciudadanos de primera clase para Struts.

Todos tienen en común que comenzaría a escribir el marcado HTML y agregaría características dinámicas utilizando etiquetas JSP (lo mismo ocurre con Velocity y Freemarker). Aunque no se recomienda, JSP incluso admite la adición de Java simple al marcado.

Dicho esto, puede acceder fácilmente a su modelo de datos desde la vista. Struts 2 proporciona características para devolver JSON, que generalmente

alimenta las páginas web controladas por AJAX. Con eso, es fácil utilizar jQuery o AngularJS como capa frontal e incluso descartar JSP por completo, aunque en el presente trabajo no lo haremos.

Como se mencionó, además de Java Server Pages, hay otras tecnologías de presentación disponibles para Struts:

- Freemarker
- iText (PDF)
- JasperReports
- Velocity
- XSLT

Lenguaje de marcado extensible (XML)

Las características proporcionadas por el framework se basan en una serie de objetos que a veces se implementan utilizando un archivo de configuración escrito en Extensible Markup Language. XML también se utiliza para configurar aplicaciones web Java.

Afortunadamente Struts 2 reduce la necesidad de XML a casi cero, ya no es crucial escribir documentos XML largos para crear una aplicación de Struts.

Controlador de vista de modelo (MVC)

Las aplicaciones web basadas en JavaServer Pages a veces combinan código de base de datos, código de diseño de página y código de flujo de control. En la práctica, encontramos que a menos que estas ocupaciones se separen, las aplicaciones más grandes se vuelven difíciles de mantener.

Una forma de separar las ocupaciones en una aplicación de software es usar una arquitectura de Modelo-Vista-Controlador (MVC). El modelo representa el negocio o el código de la base de datos, la vista representa el código de diseño de la página y el controlador representa el código de navegación.

El término "MVC" se originó con el marco de SmallTalk Model-View-Controller. En Smalltalk MVC, la Vista se actualiza a sí misma desde el Modelo, a través del patrón "Observador". El patrón MVC original es como un bucle cerrado: la Vista habla con el Controlador, que habla con el Modelo, que habla con la Vista.

Sin embargo, un enlace directo entre el modelo y la vista no es práctico para las aplicaciones web, y modificamos la disposición MVC clásica para que se vea menos como un bucle y más como una herradura con el controlador en el medio.

En el patrón de diseño MVC / Modelo 2, el flujo de la aplicación está mediado por un Controlador central. El controlador delega las solicitudes, en nuestro caso, las solicitudes HTTP, a un controlador apropiado. Los controladores están vinculados a un modelo, y cada controlador actúa como un adaptador entre la solicitud y el modelo. El modelo representa, o encapsula, la lógica o estado comercial de una aplicación. Por lo general, el control se reenvía a través del Controlador a la Vista apropiada. El reenvío puede determinarse consultando un conjunto de asignaciones, generalmente cargadas desde una base de datos o archivo de configuración. Esto proporciona un acoplamiento débil entre la Vista y el Modelo, lo que puede hacer que las aplicaciones sean significativamente más fáciles de crear y mantener.

Si bien MVC es un paradigma conveniente, muchos trabajadores encuentran que las aplicaciones pueden utilizar más de tres capas. Por ejemplo, dentro del Modelo, a menudo hay capas de acceso a datos y de lógica de negocios distintas.

El framework proporciona la capa de control para un modelo 2 de aplicaciones web. Los desarrolladores pueden usar esta capa con otras tecnologías estándar para construir las capas de negocio, acceso a datos y presentación.

Cómo crear una aplicación web de struts2

[22]

Requisitos

Struts 2 requiere Servlet API 2.4 o superior, JSP 2.0 o superior y Java 7 o superior.

Uso de Maven para construir la aplicación

Una vez se ha creado una aplicación de tipo Maven en el IDE de Java, debemos modificar el fichero pom.xml (del contexto de Maven) para agregar los siguientes complementos Maven:

- maven-compiler-plugin 3.1 [23]: se utiliza para compilar las fuentes del proyecto. El compilador predeterminado es javax.tools.JavaCompiler. Mención especial requiere la opción `<compilerArgs><arg>-parameters</arg></compilerArgs>` Esta opción consigue que los nombres de los parámetros de los métodos de las clases se conserven en su versión compilada de cara a su utilización en la tecnología **reflection** que se comentará más adelante
- maven-war-plugin 2.3 [24]: este complemento es el responsable de recopilar todas las dependencias de artefactos, clases y recursos de la aplicación web y empaquetarlos en un archivo war
- maven-dependency-plugin 2.6 [25]: proporciona la capacidad de manipular artefactos. Puede copiar y desempaquetar artefactos de repositorios locales o remotos a una ubicación específica.

Agregar Struts2

Tras agregar los complementos Maven indicados, debemos agregar las bibliotecas de Struts 2 nuestro Class Path. Puesto que estamos utilizando Maven, nos basta con añadir el siguiente texto al fichero pom.xml

```
<dependency>
<groupId>org.apache.struts</groupId>
<artifactId>struts2-core</artifactId>
<version>2.3.34</version>
<type>jar</type>
</dependency>
```

De esta manera Maven obtendrá el `struts2-core.jar` y los otros archivos jar que struts2-core requiere

Añadir log

Para ver qué sucede debajo de la aplicación, vamos a usar log4j2 que añadiremos mediante las correspondientes dependencias Maven:

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.7</version>
    <type>jar</type>
</dependency>
<dependency>
```

```
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>2.7</version>
            <type>jar</type>
        </dependency>
```

y configuraremos mediante el fichero `log4j2.xml` ubicado en `src/main/resources`:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Appenders>
        <Console name="STDOUT" target="SYSTEM_OUT">
            <PatternLayout pattern="%d %-5p [%t] %C{2}
(%F:%L) - %m%n"/>
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="com.opensymphony.xwork2"
level="debug"/>
        <Logger name="org.apache.struts2" level="debug"/>
        <Root level="warn">
            <AppenderRef ref="STDOUT"/>
        </Root>
    </Loggers>
</Configuration>
```

Definir el filtro Struts2

Para habilitar el marco de trabajo de Struts 2 para que funcione con nuestra aplicación web, debemos agregar una clase de filtro Servlet y una asignación de filtro a `web.xml`. A continuación, se muestra el aspecto del `web.xml` que puede tener después de agregar el filtro y los nodos de mapeo del filtro. web.xml está en la carpeta `src/main/webapp/WEB-INF`.

```
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndE
xecuteFilter
        </filter-class>
        <init-param>
            <param-name>struts.devMode</param-name>
```

```
            <param-value>false</param-value>
        </init-param>
    </filter>
```

El valor del nodo `<url-patter>/*` significa que el filtro Struts 2 se aplicará a todas las URL de esta aplicación web.

Crear struts.xml

Struts 2 puede usar un archivo de configuración XML o anotaciones (o ambas) para especificar la relación entre una URL, una clase de Java y una página de vista (por ejemplo index.jsp). Para nuestra aplicación usaremos una configuración mínima de XML. El nombre del archivo es `struts.xml` y debe estar en la `src/main/resources`.

```
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration
2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.devMode" value="true" />
    <package name="basicstruts2" extends="struts-
default">
        <action name="index">
            <result>/index.jsp</result>
        </action>
    </package>
</struts>
```

Este archivo de configuración mínimo le dice al frame que si la URL termina en `index.action` redirecciona el navegador a `index.jsp`

4.3. Maven

[26] **Maven** es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Es similar en funcionalidad a Apache Ant, pero tiene un modelo de configuración de construcción más simple, basado en un formato XML. Estuvo integrado inicialmente dentro del proyecto Jakarta pero ahora ya es un proyecto de nivel superior de la Apache Software Foundation.

Maven utiliza un Project Object Model (POM) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Viene con objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado.

Una característica clave de Maven es que está listo para usar en red. El motor incluido en su núcleo puede dinámicamente descargar plugins de un repositorio, el mismo repositorio que provee acceso a muchas versiones de diferentes proyectos Open Source en Java, de Apache y otras organizaciones y desarrolladores. Este repositorio y su sucesor reorganizado, el repositorio Maven 2, pugnan por ser el mecanismo *de facto* de distribución de aplicaciones en Java, pero su adopción ha sido muy lenta. Maven provee soporte no solo para obtener archivos de su repositorio, sino también para subir artefactos al repositorio al final de la construcción de la aplicación, dejándola al acceso de todos los usuarios. Una caché local de artefactos actúa como la primera fuente para sincronizar la salida de los proyectos a un sistema local.

Maven está construido usando una arquitectura basada en plugins que permite que utilice cualquier aplicación controlable a través de la entrada estándar. En teoría, esto podría permitir a cualquiera escribir plugins para su interfaz con herramientas como compiladores, herramientas de pruebas unitarias, etcétera, para cualquier otro lenguaje. En realidad, el soporte y uso de lenguajes distintos de Java es mínimo. Actualmente existe un plugin para .Net Framework y es mantenido, y un plugin nativo para C/C++ fue alguna vez mantenido por Maven 1.

4.4. NetBeans IDE 8.2

[27] NetBeans es un entorno de desarrollo integrado libre, hecho principalmente para el lenguaje de programación Java. Existe además un número importante de módulos para extenderlo. NetBeans IDE1 es un producto libre y gratuito sin restricciones de uso.

Usando Maven en el IDE

[28] El soporte para Maven está totalmente integrado en NetBeans IDE. El desarrollo de un proyecto que utiliza el marco de Maven es casi idéntico al

desarrollo de un proyecto en Ant. Sin embargo, existen algunas diferencias relacionadas con la forma en que Maven construye proyectos y trabaja con dependencias.

Aspectos relevantes del desarrollo del proyecto

5.1. Conexión a la base de datos

Se trata de la primera funcionalidad que se debe incorporar a la aplicación: la posibilidad de obtener una conexión con una base de datos concreta. En este trabajo nos centraremos en la obtención directa de conexiones a través de la clase DriverManager, sin utilizar datasources definidos a nivel del servidor web ni pool de conexiones.

Según el API de JDBC, existen tres opciones de obtener conexiones a la base de datos a través del método `DriverManager.getConnection()`

- `getConnection(String url, Properties info)`
- `getConnection(String url, String user, String password)`
- `getConnection(String url)`

En todos los casos es necesario indicar la url que indica siempre el driver que se debe utilizar por parte del servidor para establecer la conexión con la base de datos.

La tercera opción, solo indicar url, no se va a utilizar, puesto que siempre se indicará al menos usuario y contraseña.

La aplicación deberá obtener una conexión a la base de datos cuando el usuario se identifica mediante usuario y contraseña. La conexión se almacenará en la sesión del servidor web y será utilizada en cada petición o consulta que se realice a la base de datos. Cuando la sesión web del usuario muera por

inactividad, el proceso correspondiente deberá cerrar la conexión establecida con la base de datos si aún estuviera activa.

Antes de poder utilizar una conexión recuperada de la sesión web del usuario deberá comprobarse que la conexión continúa activa. Si la conexión ya no estuviera activa, se intentará obtener una nueva conexión a la base de datos utilizando para ellos los datos de conexión almacenados en la sesión web del usuario.

Seguridad

La primera pantalla que se presenta al usuario consiste en la pantalla de identificación donde el usuario deberá introducir los datos necesarios para que se establezca la conexión con una base de datos.

Figura 5.1: redirección http - https: redirección http - https

Estos datos serán:

- url: será la url de la base de datos, de la forma jdbc.subprotocolo:subnombre
- usuario: nombre del usuario de base de datos
- password: contraseña del usuario de base de datos. La contraseña deberá ir oculta en un campo html tipo password

web.xml

Carpeta WEB-INF

Toda solicitud realizada para presentar esta página deberá ser redirigida al puerto SSL correspondiente utilizando a partir de entonces el protocolo HTTPS para cualquier comunicación entre cliente y servidor. Para conseguir esta operación, independientemente de la configuración del servidor, debemos

definir a nivel de aplicación, la configuración de seguridad de la siguiente manera:

```
<security-constraint>
    <display-name>Constraint1</display-name>
    <web-resource-collection>
        <web-resource-name>Todos</web-resource-name>
        <description/>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
        <description/>
        <transport-guarantee>
              CONFIDENTIAL
        </transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

de esta manera, todas las solicitudes que se realicen pasan a ser del tipo CONFIDENTIAL que en la práctica equivale a ser redirigidas al protocolo HTTPS

server.xml

En el desarrollo de la aplicación se ha utilizado un servidor apache tomcat y, aunque no es obligatorio su utilización y puede ser sustituido por otro servidor web, es necesaria la utilización de un motor java 1.8.

En la configuración del servidor, el fichero server.xml debe indicar que están abiertos los puertos de respuesta a los protocolos https puesto que todas las peticiones http realizadas son redirigidas a https:

```
<Connector port="8443"
protocol="org.apache.coyote.http11.Http11NioProtocol"
maxThreads="150" SSLEnabled="true" scheme="https"
secure="true"
clientAuth="false"
sslProtocol="TLS" />
```

Inicio

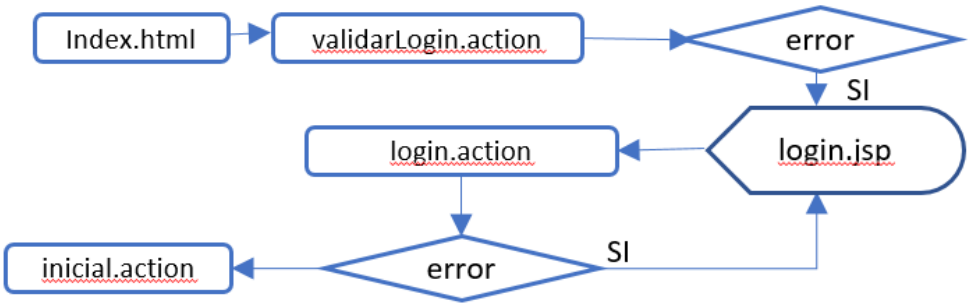


Figura 5.2: Esquema general de inicio en aplicación

struts.xml

Carpeta /WEB-INF/classes

Con el fin de poder proporcionar desde la primera pantalla un soporte idiomático, la visualización de esta primera pantalla será inicializada a través del framework struts. La acción que se inicia en index.html es validarLogin.action que es mapeada al método `LoginAction.validarLogin()`:

```
<action name="validarLogin"
      class="es.ubu.alu.mydatabasejc.actions.LoginAction"
      method="validarLogin">
   <result name="error">/login.jsp</result>
   <result type="redirectAction">
      <param name="actionName">inicial</param>
      <param name="namespace">/consulta</param>
      <param name="parse">false</param>
   </result>
</action>
```

Cuando, de resultas de la ejecución del método validarLogin, se devuelva “error”, se presentará de nuevo la pantalla login.jsp. Si, por el contrario, se devuelve “success” se deberá redirigir la acción a /consulta/inicial.action que presentará la información relativa a la conexión conseguida en la pantalla inicial de la aplicación.

login.jsp

Carpeta /

La solicitud HTTP se deberá realizar mediante el protocolo POST evitando que los datos enviados sean incluidos en la url enviada. En el formulario se incluirán los tres parámetros indicados así como la acción que debe intentar realizar el login:

```
<s:form action="login.action" method="POST"
        namespace="/">
   <s:textfield name="url" key="URL"/>
   <s:textfield name="usuario" key="Usuario"/>
   <s:textfield name="password" key="Contrasena"/>
   <s:submit key="Conectar"/>
</s:form>
```

El mensaje indicativo si existe algún problema aparecerá en la misma pantalla de identificación para que el usuario pueda hacer un nuevo intento de conexión. Para ello se dota al jsp de las líneas:

```
<s:actionerror/>
<s:actionmessage/>
```

que se transformarán en una lista html de mensajes cargados por el programa correspondiente

package.properties

Paquete es.ubu.alu.mydatabasejc

Se puede apreciar bajo el atributo **key** los textos que pasados a través de los ficheros **package.properties** nos permiten la internacionalización de la pantalla:

```
# Etiquetas estáticas en pantalla
Contrasena=Contraseña
```

A modo de ejemplo, el fichero **package_en.properties** daría soporte a navegadores configurados en el idioma inglés:

```
# Etiquetas estáticas en pantalla
Contrasena=Password
Usuario=User
Conectar=Connect
```

struts.xml

La acción login.action es redirigida a LoginAction.login() que establecerá la conexión física con la base de datos en función de los datos suministrados en el formulario y añadirá a la sesión web del usuario el objeto que contiene la conexión como tal y los datos adicionales de la misma por si hay que reestablecerla en el futuro. En struts.xml el mapeo de la acción y el retorno del proceso:

```
<action name="login"
      class="es.ubu.alu.mydatabasejc.actions.LoginAction"
      method="login">
   <result name="error">/login.jsp</result>
   <result>/inicio.jsp</result>
</action>
```

Identificación

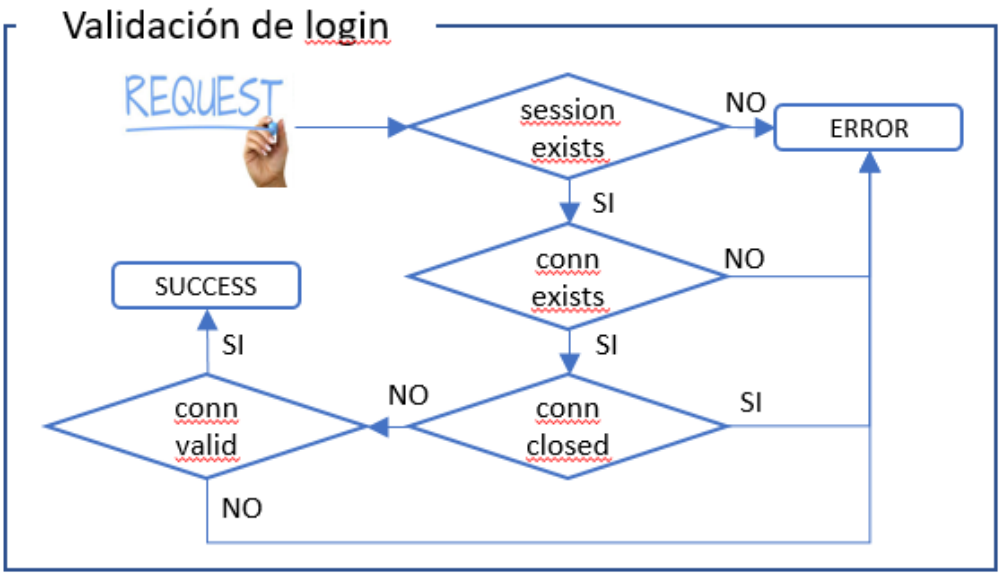


Figura 5.3: Validación de login

LoginAction.java

Paquete es.ubu.alu.mydatabasejc.actions

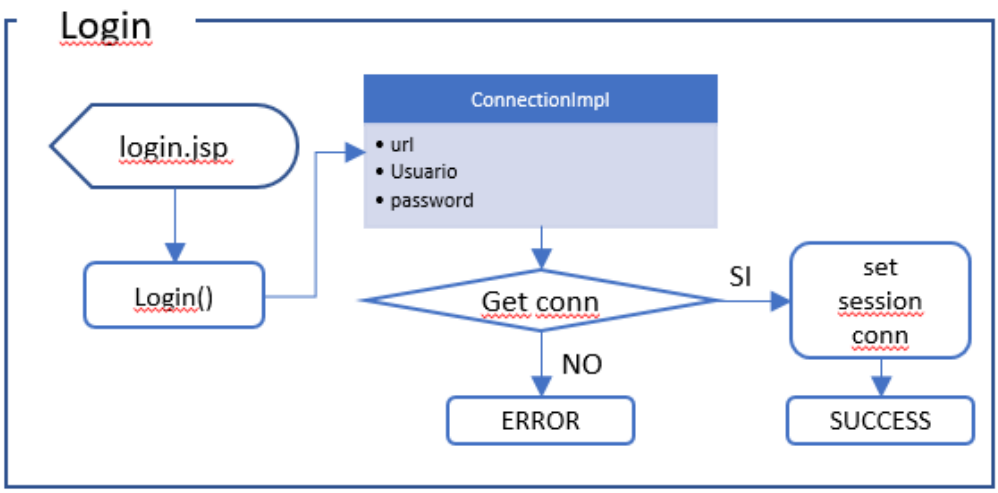


Figura 5.4: Procesamiento de login

Esta clase es la encargada de recoger las solicitudes del usuario relativas a los distintos procesos de login e indicar a struts que debe hacer a continuación.

Los datos del formulario HTML son recogidos en las propiedades declaradas en la clase java con el mismo nombre que se definió en los propios campos del formulario. Las propiedades deben estar dotadas de sus métodos gets y sets oportunos:

```
private String url;
private String usuario;
private String password;
```

La respuesta HTTP obtenida presentará la información inicial de la base de datos si se consigue establecer una conexión válida o un mensaje indicativo del problema encontrado si no se consigue obtener una conexión válida.

Como requisitos de datos, la url y el nombre del usuario son datos obligatorios. Para controlar la validez de url y usuario, se utiliza el método validate de la correspondiente clase action de java.

Los métodos más importantes definidos en esta clase son:

- **logear**(): Se utiliza para que el usuario pueda ir directamente a la pantalla para hacer login, aunque ya se encuentre logado en una base de datos.

- **login**(): Recoge url, usuario y clave del formulario html e intenta realizar una conexión a la base de datos. Si tiene éxito, se guardará en la sesión del usuario el objeto que contiene la conexión a la base de datos.

```
return login(url, usuario, password);
```

- **validarLogin**(): Comprueba que el usuario disponga de una sesión.

```
HttpServletRequest request =
ServletActionContext.getRequest();
if (request.getSession().isNew()) return ERROR;
```

A continuación, comprueba que el usuario disponga de una conexión a la base de datos ya establecida previamente y que sea válida.

```
ConnectionImpl connectionImpl = (ConnectionImpl)request
   .getSession().getAttribute(CONEXION);
```

Si dispone de una conexión pero no es válida se intenta realizar una nueva conexión con los mismos datos previos. Si no ha tenido previamente una conexión, se presenta al usuario el formulario para que ingrese url, usuario y clave.

```
if (connectionImpl==null) {
   addActionError(getText("Conexión no realizada"));
   return ERROR;
}
```

si la conexión se ha cerrado, se intenta establecer nuevamente

```
if (connectionImpl.isClosed())
   return login(
       connectionImpl.getUrl(),
       connectionImpl.getUsuario(),
       connectionImpl.getPassword());
```

si la conexión no es válida, se intenta establecer nuevamente

```
if (!connectionImpl.isValid(0))
   return login(
       connectionImpl.getUrl(),
       connectionImpl.getUsuario(),
       connectionImpl.getPassword());
```

validación de login correcta

```
return SUCCESS;
```

- **validate**(): Comprueba los valores del formulario html enviados. Se vuelve a la misma pantalla si alguno de los datos no es correctamente validado (url o usuario vacío). La siguiente línea, a modo de ejemplo, muestra cómo conseguir que se muestre un mensaje idiomatizado cuando se produce un fallo de validación:

  ```
  if ("".equals(url))
   addFieldError(
      "url", "La.URL.es.un.dato.obligatoria");
  ```

- **login(String url, String usuario, String password):** Se trata de un método **`private`**, llamado desde el método **`login()`** y desde el método **`validarLogin()`**. Este método es el responsable de intentar establecer finalmente la conexión con la base de datos. Si finalmente tiene éxito, también añadirá la instancia de ConnectionImpl a la sesión del usuario con el nombre “conexion” para ser recuperada en llamadas posteriores.

  obtiene la conexión, con control de errores

  ```
  ConnectionImpl connectionImpl = new ConnectionImpl(
    url, usuario, password);
  ```

  se añade el objeto en la sesión del usuario, si ya existía se cambia

```
HttpServletRequest request =
ServletActionContext.getRequest();
request.getSession().setAttribute(CONEXION,
connectionImpl);
```

ConnectionImpl.java

Paquete es.ubu.alu.mydatabasejc.jdbc

El objetivo de esta clase es contener la conexión JDBC oportuna con la base de datos, junto con el resto de los datos relevantes de la misma (url, usuario, clave).

También dispone de una serie de métodos para trabajar con la conexión JDBC contenida en ella. Los métodos más importantes son:

- **ConnectionImpl(String url, String usuario, String password):** Es el constructor de la clase. Inicializa el driver JDBC oportuno en función de la url recibida

  ```
  Class jdbcClass = Class.forName(driver);
  ```

  e instancia el objeto completando la conexión JDBC con la base de datos.

  ```
  connection = DriverManager.getConnection(
      url, usuario, password);
  ```

  Cualquier error es devuelto previo registro en el log de la aplicación.
- **finalize**(): Método sobreescrito que finaliza la conexión jdbc antes de que el objeto sea finalizado por el recolector de basura. De esta manera nos aseguramos que no queden conexiones con la base de datos abiertas innecesariamente
- **isClosed**(): true si la conexión con la base de datos está cerrada o no existe, false en caso contrario
- **isValid(int timeout):** true si la conexión con la base de datos es válida, false en caso contrario

ConnectionException.java

Paquete es.ubu.alu.mydatabasejc.exceptions

Esta clase hereda de la clase Throwable y permite registrar en el log definido, no solamente los errores producidos en operaciones de establecimiento y validación de conexión a la base de datos, si no también adjuntar en el mismo log información relevante de los datos del contexto utilizados para la conexión.

Por ejemplo, si no se puede inicializar el driver JDBC, se acompaña cuál es el driver que intenta inicializarse. Si no se consigue una conexión, se acompaña al mensaje de error la url y el usuario que intentaron realizarla.

5.2. Presentación de la base de datos

Se trata de la pantalla inicial de la aplicación tras la identificación y la obtención de la conexión de la base de datos.

En esta pantalla, en la parte superior, presentamos información general de la conexión (driver, url y usuario conectado) junto con una opción para desconectarnos de la base de datos (lo que nos llevará a la pantalla de login.

Existirá un faldón lateral izquierdo donde se presentará información relativa a la conexión agrupada en opciones de menú que se agruparán en dos tipos:

- Información simple
- Información tabulada

En la parte central de la pantalla se presentan las propiedades informativas de la conexión con la base de datos. Se trata de información obtenida de la conexión basada en los métodos disponibles en un objeto connection. Es, por lo tanto, información general correspondiente a cualquier tipo de conexión JDBC realizada con el programa.

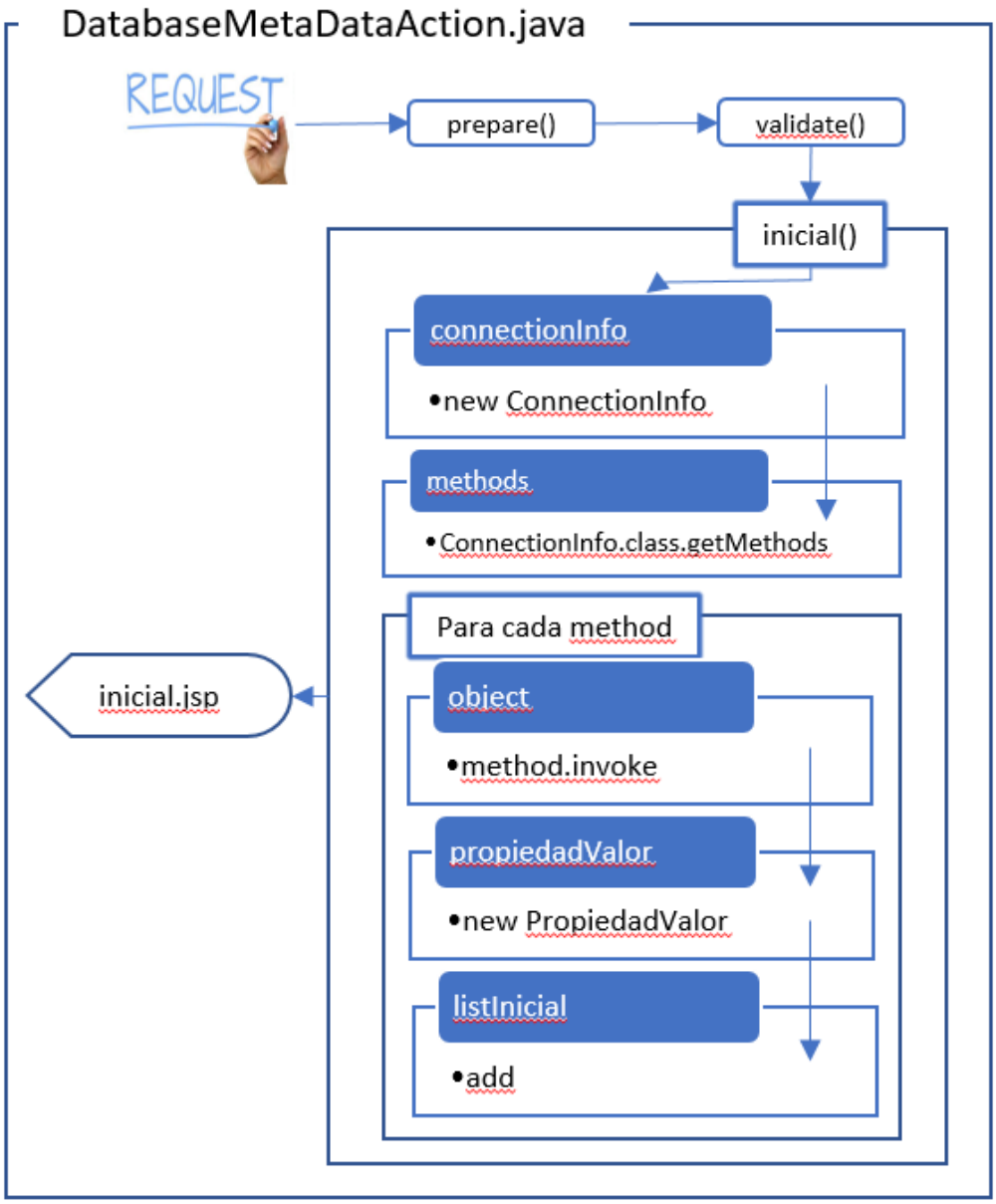


Figura 5.5: Presentación de la pantalla inicial

DatabaseMetaDataAction.java

Paquete es.ubu.alu.mydatabasejc.actions

Es la clase que proporciona la información necesaria para la presentación de la pantalla `inicial.jsp`

Puesto que esta clase va a hacer un uso intensivo de la sesión del usuario, implementamos los métodos del interface `SessionAware` de struts2 que mapea en una propiedad de tipo `Map<String, Object>` la sesión web del usuario:

```
public class ConsultaAction extends LoginAction
implements Preparable, SessionAware
...
private Map<String, Object> sesion;
...
@Override
public void setSession(Map<String, Object> map) {
  this.sesion = map;
}
```

De esta manera disponemos de todos los datos de la sesión del usuario bajo la propiedad `sesion`.

Dispone de los siguientes métodos:

- **prepare():** En un primer lugar, se cargará de la sesión del usuario el objeto con la conexión a la base de datos para lo que sobreescribe el método prepare() que será ejecutado de forma previa a cualquier otro método:

  ```
  connectionImpl = (ConnectionImpl) sesion
       .get("conexion");
  ```

- **validate()**: Recoge las validaciones que se deben realizar. Para ello se utiliza el método `LoginAction.validarLogin(ConnectionImpl)` para lo que es necesario heredar esta clase de la clase `LoginAction`:

  ```
  super.validarLogin(connectionImpl);
  ```

  Un error en este método volverá a la pantalla `login.jsp (struts.xml)`

- **inicial()**: Partiendo ya de una conexión válida, se obtienen los datos generales de la conexión. Para ello, se instancia un objeto de la clase ConnectionInfo

  ```
  ConnectionInfo connectionInfo =
       new ConnectionInfo(connectionImpl
          .getConnection());
  ```

y, mediante **Reflection** obtenemos todos los métodos de esta clase y los ejecutamos guardando tanto su nombre como su resultado en una lista que se visualizará en `inicial.jsp`

```
Method[] methods = ConnectionInfo
    .class.getMethods();
for (Method method : methods) {
    Object o = method
       .invoke(connectionInfo, new Object[]{});
    PropiedadValor p =
       new PropiedadValor(method.getName(), o);
    listInfo.add(p);
}
```

- **getMenus()**: Carga la lista de opciones de menú. Esta lista consiste en la relación de métodos del objeto `DatabaseMetaDataImpl` que devuelven un objeto del tipo `ResultSet` o un objeto de tipo `List<Object>`. El código que carga la lista de opciones de menú es similar al anterior cambiando la condición que debe cumplir el método por la siguiente:

```
if (method.getReturnType()==ResultSet.class ||
    method.getReturnType()==List.class)
```

Para cada método debemos obtener la lista de tipos de parámetros que posee puesto que la necesitamos posteriormente para identificar de forma única al método a invocar. En función del tipo de objeto devuelto por la función, la acción será “resultset” o “info” y se define el objeto `Menu` que se añadirá a la lista de menú:

```
Menu menu = new Menu(
   method.getReturnType()==ResultSet.class
       ? "resultset" : "info",
   method.getName(),
   method.getParameterTypes);
```

y con el objeto de no repetir cuando existe métodos con sobrecarga de parámetros:

```
if ((index = listMenu.indexOf(menu))==-1) {
    listMenu.add(menu);
}
```

ConnectionInfo.java

Paquete es.ubu.alu.mydatabasejc.jdbc

Contiene una propiedad de tipo `java.sql.Connection` y un conjunto de métodos que proporcionan información relativa a dicha conexión. El constructor recibe una conexión jdbc, se la asigna a la propiedad `connection` y, por cada método, realiza la correspondiente llamada al método equivalente del objeto `Connection`

PropiedadValor.java

Paquete es.ubu.alu.mydatabasejc

Es una clase que contiene los atributos propiedad y valor de tipo String y Object respectivamente que permitirá almacenar listas de este tipo para su presentación en pantalla.

Con el objeto de poder comparar dos objetos de esta clase, se sobreescribe el método `equals()` como sigue:

```
@Override
public boolean equals(Object obj) {
   if (!(obj instanceof PropiedadValor)) return false;
   if (obj==null) return false;
   if (this.propiedad==null &&
     ((PropiedadValor)obj).propiedad==null) return true;
   Return this.propiedad.equalsIgnoreCase(
      ((PropiedadValor)obj).propiedad);
}
```

Menu.java

Paquete es.ubu.alu.mydatabasejc

Define la estructura de cada opción de menú de la aplicación. Los atributos son:

- **action**: Es la acción que struts2 ejecutará cuando el usuario elija esta opción del menú

- **metodo**: Junto con la acción, se especifica el método que se debe ejecutar mediante Reflection
- **parametros**: Contiene en codificación Base64 el array de clases que conforman la llamada al método a invocar.

A destacar el siguiente constructor:

```
public Menu(String action, String metodo,
            Class[] parametros)
```

Convierte el array de clases parametros en una cadena codificada en base64 mediante las siguientes líneas:

```
ByteArrayOutputStream bs = new ByteArrayOutputStream();
ObjectOutputStream os = new ObjectOutputStream(bs);
os.writeObject(parametros);
String params = Base64.
encodeBase64URLSafeString(bs.toByteArray());
```

y finalmente lo asigna al atributo `parametros`

```
this.parametros = params;
```

DatabaseMetaDataImpl.java

Paquete es.ubu.alu.mydatabasejc.jdbc

Esta clase implementa el interface `java.sql.DatabaseMetaData` por lo que sobreescribe todos los métodos de dicho interface.

Solo dispone de una propiedad de tipo `DatabaseMetaData` que es inicializada en el constructor a través de un objeto de este tipo que se le ha de pasar por parámetro y que se obtendrá de la conexión a la base de datos mediante el método `getMetaData()`.

```
protected DatabaseMetaData metadata;
public DatabaseMetaDataImpl(DatabaseMetaData metadata){
   this.metadata = metadata;
}
```

Todos los métodos que sobreescriben los del interfaz implementado simplemente ejecutan el método correspondiente de la propiedad `metadata`. A modo de ejemplo vemos el siguiente:

```
@Override
public String getURL() throws SQLException {
   return metadata.getURL();
}
```

Además de los métodos del interfaz implementado, se proporcionan una serie de métodos que devuelven un objeto de tipo `List<List>` y que forman una agrupación de un conjunto de métodos simples que proporcionan información sobre usuarios, conexión, datos, sql, etc. Estos métodos devuelven todos ellos información en formato `List<List>` y son:

- **getBasicInfo()**: obtiene información básica.
- **getConnInfo**(): Información de la conexión
- **getDataBaseInfo**(): Información de la base de datos
- **getDataInfo**(): Información de los datos
- **getUserInfo**(): Información del usuario de base de datos

Si quisiéramos, podríamos añadir tantas funciones adicionales para agrupar los métodos simples de `DatabaseMetaDataImpl` en función del criterio que quisiéramos. La única restricción es que el objeto devuelto debe ser del tipo `List<List>`

inicial.jsp

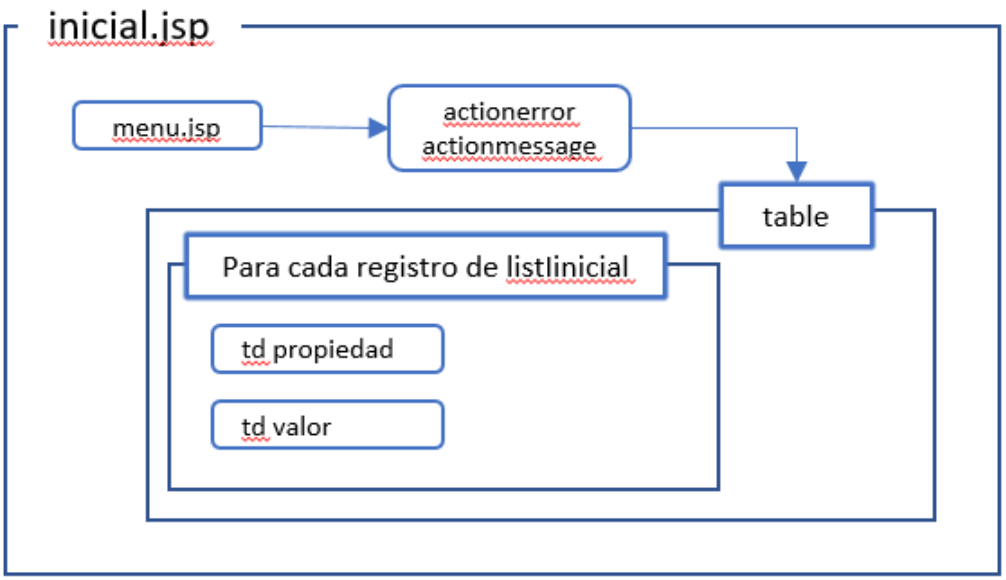


Figura 5.6Lógica de inicial.sp

En la cabecera, se presenta la información utilizada para establecer la conexión, obtenida del objeto connectionImpl:

```
<table>
   <tr>
      <td><s:property value="connectionImpl.url"/></td>
      <td>
          <s:property value="connectionImpl.driver"/>
       </td>
   </tr>
   <tr>
      <td>
          <s:property value="connectionImpl.usuario"/>
       </td>
      <td> </td>
   </tr>
</table>
```

En la parte central presenta la información de la conexión establecida. En la parte principal de la pantalla utiliza un iterador de struts para recorrer una estructura `List<PropiedadesValor>` y presentar tanto el nombre de la propiedad (el nombre del método pasado por la función `getText` que busca la traducción en `package.properties`) como el valor para cada propiedad

```
<table class="tabla">
   <tr>
      <th>Propiedad</th>
      <th>Valor</th>
   </tr>
   <s:iterator value="listInfo">
      <tr>
         <td>
            <s:property value="%{getText(propiedad)}"/>
         </td>
         <td><s:property value="valor"/></td>
      </tr>
   </s:iterator>
</table>
```

En el lateral izquierdo se despliega el menú de opciones para presentar distinto tipo de información. Para determinar el menu se utiliza una jsp que puede ser llamada desde cualquier jsp mediante el tag include:

```
<s:include value="/WEB-INF/jspf/menu.jsp"/>
```

Menu.jsp

Carpeta /WEB-INF/jsp

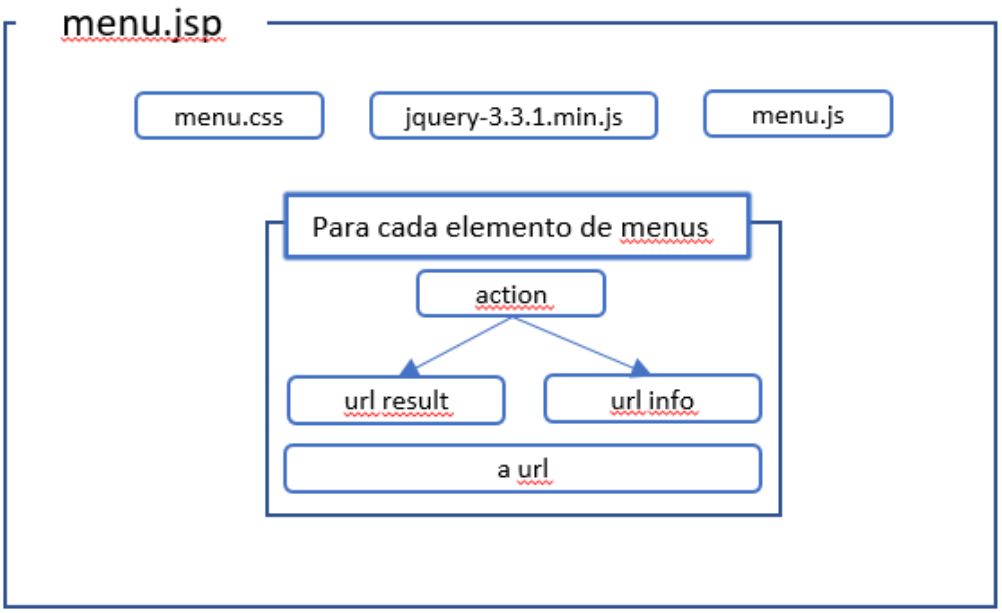


Figura 5.7: Lógica de menu.jsp

Inicialmente se muestra el nombre de la conexión:

```
<s:property value="connectionImpl.connection"/>
```

y a continuación se muestran los métodos disponibles del objeto `DatabaseMetaDataImpl` obtenido mediante el método `getMetaData()` del objeto `connection` que devuelvan objetos de tipo `ResultSet`. La información se visualiza en forma de lista html. Cada elemento de la lista es un link que llevará a visualizar la información del `ResultSet` correspondiente en la parte central de la pantalla:

```
<ol>
   <s:iterator value="menus">
      <li>
         <s:url action="%{action}" var="urlTag">
            <s:param name="metodo">
               <s:property value="metodo"/>
            </s:param>
            <s:param name="parametros">
               <s:property value="parametros"/>
            </s:param>
         </s:url>
         <s:a href="%{urlTag}">
            <s:property value="%{getText(metodo)}"/>
         </s:a>
      </li>
   </s:iterator>
</ol>
```

El iterador recorre `menus` (lista de objetos `Menu`) y para cada elemento compone la url destino del link acompañado como parámetros el contenido de la `propiedad` que es realmente el nombre del método de `DatabaseMetaDataImpl` que se deberá ejecutar y los tipos de parámetros que el método requiere codificados en `Base64` y bajo el nombre `parametros`.

El texto visible del link es el nombre del método pasado a través de la función `getText` que, como ya vimos, obtiene una pequeña descripción del grupo de ficheros `package.properties`

package.properties

Paquete es.ubu.alu.mydatabasejc

Como ya se comentó, en este fichero se encuentran las traducciones de mensajes y etiquetas de pantalla que se presentan por el navegador al usuario. Al pasar el mensaje o texto por la función `getText()` obligamos a struts a localizar el mensaje o texto en este fichero y presentar en su lugar el valor asignado en él. En lugar de presentar el nombre del método en pantalla, se le asigna una traducción sencilla que es la que utilizará en pantalla:

```
isReadOnly = Solo lectura
getSchema = Esquema
getAutoCommit = Auto commit
getCatalog = Catálogo
getClientInfo = Info cliente
getHoldability = Capacidad de retención
getNetworkTimeout = Timeout de red
getTransactionIsolation = Aislamiento de transacción

# Métodos de DatabaseMetaDataImpl
getBasicInfo = General
getConnInfo = Conexión
getDataBaseInfo = Base de datos
getDataInfo = Datos
getSQLInfo = SQL
getUserInfo = Usuario
getAttributes = Atributos
...
```

help.properties

Paquete es.ubu.alu.mydatabasejc

Para dotar a la aplicación de un sistema de ayuda interactivo, es necesario asociar mensajes o etiquetas con descripciones más amplias, que expliquen con más detalle los conceptos que se manejan. En relación con la información relativa a la conexión, esta información se ubica en este fichero, siguiendo el mismo planteamiento que tiene el fichero `package.properties`. Las propiedades son los nombres de los métodos y los valores son los textos de ayuda:

```
isReadOnly = esta base de datos está en modo de solo
lectura
getSchema = Recupera el nombre de esquema actual de esta
conexión
```

```
getAutoCommit = Recupera el modo de confirmación
automática actual para esta conexión
getCatalog = Recupera el nombre de catálogo actual de
esta conexión
getClientInfo = Devuelve una lista que contiene el nombre
y el valor actual de cada propiedad de información del
cliente admitida por el controlador
getHoldability = Recupera la capacidad de retención
actual de los objetos ResultSet creados con esta conexión
getNetworkTimeout = Recupera la cantidad de milisegundos
que el controlador esperará a que se complete una
solicitud de base de datos
getTransactionIsolation = Recupera el nivel de
aislamiento de transacción actual de esta conexión

# descripción de los métodos de DatabaseMetaDataImpl
allProceduresAreCallable = el usuario actual puede llamar
a todos los procedimientos devueltos por el método
getProcedures
allTablesAreSelectable = el usuario actual puede usar
todas las tablas devueltas por el método getTables en una
declaración SELECT
autoCommitFailureClosesAllResultSets = si un momento
SQLException autoCommit es true indica que todos los
ResultSets abiertos están cerrados, incluso los que se
pueden mantener
...
```

struts.xml

Se define el package `DatabaseMetaData` para dar respuesta al namespace `/DatabaseMetaData`.

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

Como norma general, cualquier error de entrada o validación en la clase action correspondiente retornará el control a `login.jsp`

```
<global-results>
<result name="input">/login.jsp</result>
</global-results>
```

La acción `inicial` se asocia a `DatabaseMetaDataAction.inicial()` y finalmente presenta la página `/inicial.jsp`

```
<action name="inicial"

class="es.ubu.alu.mydatabasejc.actions.DatabaseMetaDataAc
tion"
  method="inicial">
    <result>/inicial.jsp</result>
</action>
```

5.3. Presentación de DatabaseMetaData

Una de las funcionalidades más habituales consiste en representar en pantalla un ResultSet obtenido mediante la llamada a un método de la clase DatabaseMetaDataImpl. Un gran número de funciones de esta clase devuelve una estructura de datos de este tipo. En este capítulo comentamos los detalles de la aplicación en la llamada y presentación de este tipo de datos.

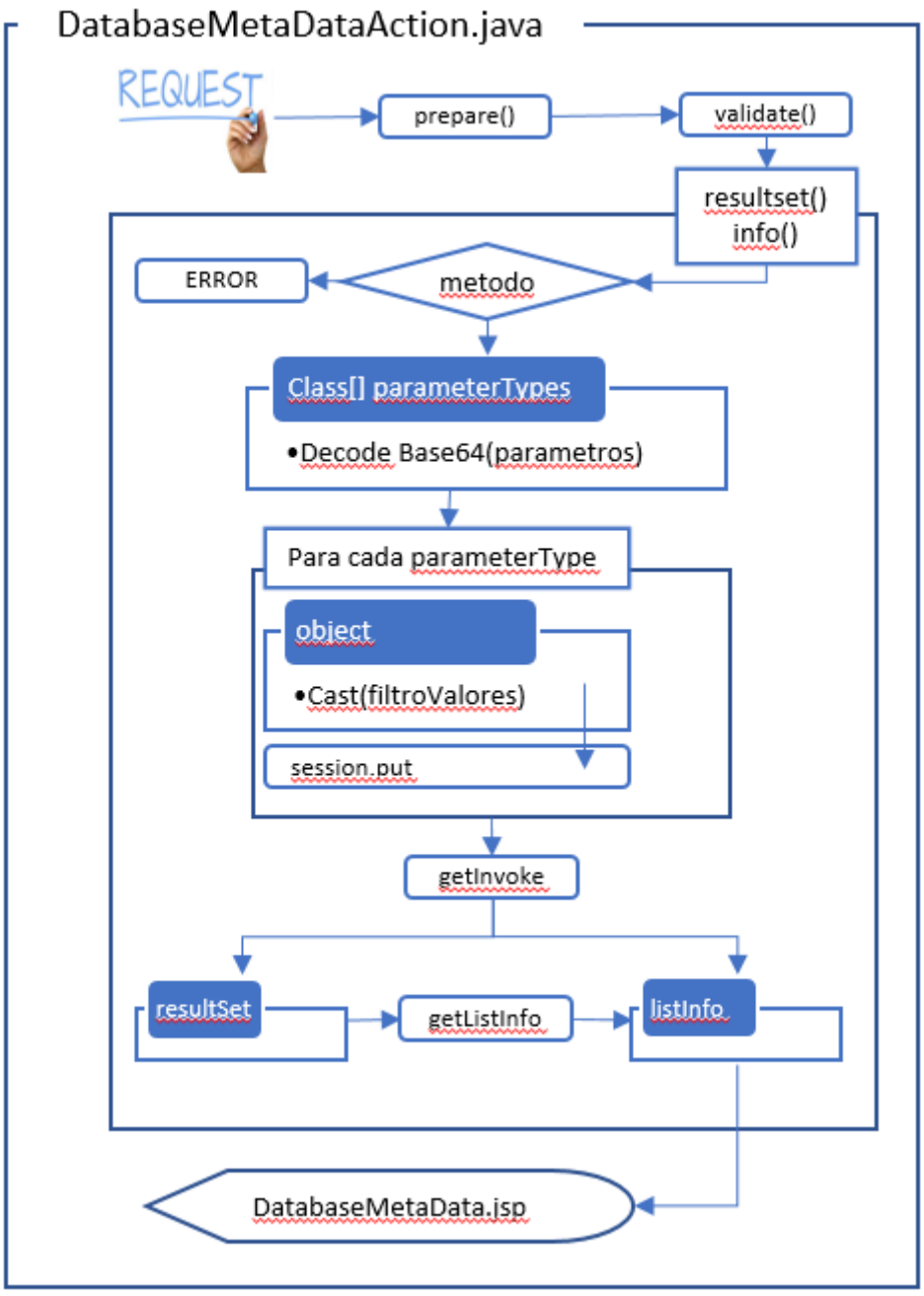


Figura 5.8: Métodos resultset e info de DatabaseMetaDataAction.java

menu.jsp

Una de las páginas típicas que realiza llamadas de este tipo es la pantalla menu.jsp:

```
<ol>
   <s:iterator value="menus">
      <li>
         <s:url action="%{action}" var="urlTag">
            <s:param name="metodo">
               <s:property value="metodo"/>
            </s:param>
            <s:param name="parametros">
               <s:property value="parametros"/>
            </s:param>
         </s:url>
         <s:a href="%{urlTag}">
   <s:property value="%{getText(metodo)}"/>
         </s:a>
      </li>
   </s:iterator>
</ol>
```

La lista de opciones de menú representada mediante una lista html consiste en una colección de links cuya acción es `resultset o List<List>` y se acompaña de un parámetro en la llamada con el nombre del método que se ha de ejecutar y a resultas de lo que se recibirá un resultset o un List<List>

struts.xml

Se mapea la acción `resultset` hacia el correspondiente método de la clase `DatabaseMetaDataAction`

```
<action name="resultset"  class="DatabaseMetaDataAction"
        method="resultset">
    <result>/DatabaseMetaData.jsp</result>
</action>
```

Una vez finalizada la acción se presentará la pantalla resultset.jsp

Igualmente, se mapea la acción “info”:

```
<action name="info"  class="DatabaseMetaDataAction"
        method="info">
    <result>/DatabaseMetaData.jsp</result>
</action>
```

DatabaseMetaData.jsp

Carpeta /

La página `DatabaseMetaData.jsp` es básicamente idéntica a la página `inicial.jsp` con la diferencia de que la parte principal de la página debe visualizar la información obtenida en el `List<List>`.

Comenzamos construyendo una tabla html con iterador para cada fila:

```
<table class="tabla">
   <s:iterator value="listInfo" var="record"
               status="status">
     <tr>
      ...
     </tr>
   </s:iterator>
</table>
```

Para cada fila, se define una variable `cuenta` que permitirá dar un formato especial a la primera columna:

```
<s:set var="cuenta" value="1"/>
```

A continuación, se construye un iterador para cada columna dentro de cada fila:

```
<s:iterator value="#record">
   ...
</s:iterator>
```

Dentro del iterador, se crea una variable `propiedad` que contendrá el valor de cada dato:

```
<s:set name="propiedad"><s:property /></s:set>
```

Además, si se trata de la primera fila, se utiliza el tag **th** en vez del tag **td** y se presenta el dato

```
<s:if test="#status.first == true">
   <th><s:property /></th>
</s:if>
<s:else>
   ...
</s:else>
```

Si no es la primera fila, se pasará a presentar el dato bajo el tag **td**, pero si se trata de la primera columna, el valor es internacionalizado usando el tag de

struts **s:text** y la variable propiedad establecida anteriormente con el valor del campo:

```
<td>
   <s:if test="#cuenta == 1">
     <s:text name="%{propiedad}"/>
```

se establece la variable cuenta a 2 (distinto de 1) para que solo la primera columna sea internacionalizada evitando de esta manera sobrecarga innecesaria.

```
<s:set var="cuenta" value="2"/>
```

En el resto de las columnas, se presenta el dato sin internacionalizar:

```
   </s:if>
   <s:else>
      <s:property />
   </s:else>
</td>
```

DatabaseMetaDataAction.java

Paquete es.ubu.alu.mydatabasejc.actions

El método `resultset` es el encargado de tratar la petición del cliente, recibir el método de la clase `DatabaseMetaDataImpl` que se debe ejecutar en la propiedad `metodo` y que devuelve el objeto `resultset` que, una vez tratado, será enviado a la página jsp correspondiente que lo presentará en forma de tabla html.

Métodos:

- **resultset()**: Es el método que, en función del método recibido en la propiedad `metodo`, llama al correspondiente método de la clase `DatabaseMetaDataImpl` obteniendo el resultset correspondiente y lo transforma y asigna a la lista `listResultSet` que es la que finalmente se visualizará. La transformación del resultset a la lista se realiza con la siguiente llamada

  ```
  listInfo = getListInfo(rs);
  ```

Para realizar la llamada al método se utiliza java `Reflection` y el método invoke acompañándole los parámetros necesarios para su ejecución. Puesto que es proceso complejo, se aísla en un método privado al que se llama con la siguiente línea:

```
ResultSet rs = getInvoke(
  connectionImpl.getConnection().getMetaData(),
  metodo,
  parametro);
```

- **info()**: Este método invoca al método correspondiente que devolverá ya el objeto List<List> capaz de ser representado por la página web
- **private getListInfo(ResultSet resultSet)**: Este método privado recibe un `ResultSet` y lo transforma en una lista de listas de objetos. Cada lista, a primer nivel, representa cada registro del resultset. Cada elemento de la lista interna contiene cada campo del registro. El primer elemento de la lista contiene los nombres de los campos del resultset.

```
int i = 0;
List cabecera = new ArrayList();
for (i = 1; i <= rsMetadata.getColumnCount(); i++) {
 cabecera.add(rsMetadata.getColumnName(i));
}
lista.add(cabecera);
```

En el bucle que recorre el recordset, si se llega al máximo de registros a devolver indicado en la clase ValoresPorDefecto, se manda un error:

```
int n = 1;
while (resultSet.next()) {
 if (n++>ValoresPorDefecto.numMaxRecords)
    throw new ResultSetException(
     "Demasiados.registros.Filtrar", resultSet);
 List record = new ArrayList();
 for (int j = 1; j < i; j++)
    record.add(resultSet.getObject(j));
 lista.add(record);
}
```

- **private getInvoke(DatabaseMetaData, metodo, parametros):** invoca al método de la clase `DatabaseMetaData` indicada. Retorna el `ResultSet` obtenido por dicha invocación.

En primer lugar, decodifica los tipos de parametros que debe recibir el método y que están codificados en Base64 y recibidos en el parámetro `parametro`:

```
Class[] parameterTypes =
getParameterTypes(parametros);
```

Una vez que se tienen los tipos de parámetros en forma de array y el nombre del método a invocar, se obtiene de `DatabaseMetaDataImpl`:

```
Method method = DatabaseMetaDataImpl.class.
 getMethod(metodo, parameterTypes);
```

A continuación, se obtiene el array de parámetros del método, no sus tipos, sino sus nombres:

```
Parameter[] arrayParametros =
  method.getParameters();
```

y se crea una lista de objetos que serán finalmente los parámetros a pasar al método invocado

```
List<Object> listaParametros = new ArrayList<>();
```

se busca cada parámetro en la sesión del usuario y lo asigna a la lista de objetos a pasar

```
for (Parameter parametro : arrayParametros) {
 listaParametros.add(
  sesion.get(parametro.getName()));
}
```

Se pasa la lista a la forma array de objetos

```
Object[] args = listaParametros.toArray();
```

y finalmente se invoca al método pasándole los argumentos necesarios

```
Object o = method.invoke(dbMetadata, args);
```

Ya solo queda retornar el objeto previa comprobación de que es del tipo ResultSet:

```
if (o instanceof ResultSet)
  return (ResultSet)o;
```

o del tipo List:

```
if (o instanceof List)
  return (List<List>)o;
```

- **Class[] getParameterTypes(String parametros):** Este método privado devuelve el array de clases que un método requiere como argumentos a partir del argumento que contiene la información en formato String Base64:

```
ByteArrayInputStream bais =
  new ByteArrayInputStream(
    Base64.decodeBase64(parametros));
ObjectInputStream ois = new ObjectInputStream(bais);
```

se obtiene el array de parámetros a partir de los datos obtenidos

```
return (Class[])ois.readObject();
```

DatabaseMetaDataImpl.java

Como se comentó en el capítulo XXXX, existen una serie de métodos en esta tabla que presentan datos informativos del tipo `Propiedad - valor`, en formato `List<List>`, para que su tratamiento en la presentación de datos sea el mismo que el que se hace en los métodos de `DatabaseMetaData` que devuelven un `ResultSet` convertido finalmente a un `List<List>` como se ha tratado en este capítulo.

A continuación, se explica cómo se convierte la colección de datos `Propiedad Valor` en un List<List> en estos métodos.

- **getInfo(Methods[] metodos)**: Va creando una lista de objetos:

```
List<List> lista = new ArrayList();
```

y le añade una cabecera genérica:

```
List cabecera = new ArrayList();
cabecera.add("Propiedad");
cabecera.add("Valor");
lista.add(cabecera);
```

Por cada método de la clase referenciada, se invoca y se guarda el nombre del método y el valor obtenido en la lista. Los métodos invocados no deben admitir parámetros:

```
Object o = null;
try {
  o = method.invoke(this, new Object[]{});
} catch (Exception e) {
  o = e.getLocalizedMessage();
}
List registro = new ArrayList();
registro.add(method.getName());
registro.add(o);
lista.add(registro);
```

Finalmente retorna la lista de listas:

```
return lista;
```

Cada método personalizado de DatabaseMetaDataImpl envía a esta función un conjunto de métodos de ejecución simple (sin parámetros y que retornan objetos de tipo simple). Para conseguir ese conjunto de métodos se utilizan las anotaciones que marcan los métodos correspondientes y la categoría a la que pertenecen. Para ello, se ha creado una anotación personalizada que permite categorizar los métodos. Es la clase MetaDataIntoCategorias que se explica más adelante.

- **getMetodos(String categoria)**: Crea una lista de almacenamiento temporal

```
List lista = new ArrayList();
```

Obtiene todos los métodos

```
Method[] methods = this.getClass().getMethods();
```

Para cada método obtiene la anotación del tipo adecuado de cada método

```
MetaDataInfoCategorias anotacion = methods[i]
    .getAnnotation(MetaDataInfoCategorias.class);
```

si no existe la anotación, pasa al siguiente método

```
if (anotacion==null) continue;
```

si existe y es de la categoría buscada, se añade a la lista temporal de métodos

```
if (anotacion.categoria().equals(categoria))
     lista.add(methods[i]);
```

convierte la lista temporal a array y lo retorna

```
return (Method[])lista.toArray();
```

- Métodos informativos: Obtienen la lista de métodos marcados con la anotación en función de su categoría y retorna la información de estos métodos en formato resultset:

```
Method[] methods = getMetodos("Básica");
return getInfo(methods, metadataBasicInfo);
```

La declaración de uno de estos métodos sería, a modo de ejemplo:

```
public ResultSet getBasicInfo() {
```

- Métodos sobreescritos de tipo informativos: Como se comentó, algunos métodos deben ser marcados con una anotación para permitirnos agruparlos bajo un método propio que devuelva sus métodos y ejecuciones en forma de resultset. Estas anotaciones, a modo de ejemplo, serían de la siguiente forma:

```
@MetaDataInfoCategorias(categoria = "Usuario")
```

MetaDataInfoCategorias.java

Paquete es.ubu.alu.mydatabasejc.annotations

Esta clase define el comportamiento de la anotación que usamos para categorizar los métodos de DatabaseMetaData de tipo informativo simple (sin parámetros y devolviendo objetos de tipo simple).

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MetaDataInfoCategorias {
   String categoria() default "Básica";
```

```
}
```

5.4. Filtro y ordenación de resultset de DatabaseMetaData

El número máximo de registros de un resultset representados es el que corresponde con el atributo `ValoresPorDefecto.numMaxRecords`. Si se procesan más de esta cantidad de registros se genera la excepción `ResultSetException` y no se presenta ninguno, ni siquiera los primeros 1000 registros y en su lugar se presenta el texto correspondiente del error.

Como pueden existir fácilmente una gran cantidad de funciones o consultas que devuelvan una cantidad superior de registros, se hace necesario un filtrado de datos que, mediante la aplicación de uno o varios criterios, reduzca el número de registros del resultset.

En los resultset proporcionados por las funciones de `DatabaseMetaData`, el filtrado se realiza mediante los argumentos de llamada de la función. Existen funciones que no precisan argumentos y otras que sí los precisan. Entre estas, cuando a un argumento se le asigna el valor null, dicho concepto no entra en el filtrado del resultset resultante. Por ejemplo, en la función `getTables()` uno de los argumentos es `schemaPattern`. Si en la llamada se le asigna el valor null a este parámetro, la función no utilizará el filtro del nombre del esquema en el resultado de la misma.

Como se ha comentado antes, el número y nombre de los argumentos varía de una función a otra, por lo que se tiene que diseñar un sistema flexible que pueda adaptarse a estos requisitos. Además hay que tener en cuenta que el tipo de datos de los argumentos también cambia de unos argumentos a otros y existen argumentos de tipo `String`, de tipo `int`, de tipo `boolean` y de tipo `String[]`.

En cuanto a la ordenación, se pretende que el usuario pueda ordenar la tabla html obtenida por cualquiera de las columnas que la conforman tanto en orden ascendente como descendente. Es una operación que se desarrolla integramente en el cliente mediante tecnología javascript para lo que se utiliza jquery y propiedades css. Al ser un código de ejecución en el cliente, el punto de entrada será siempre el jsp.

DatabaseMetaDataAction.java

Paquete es.ubu.alu.mydatabasejc.actions

Los tipos de datos de una función de DatabaseMetaData son pasado en la request bajo el parámetro `parametros` codificado en `Base64` como vimos en capítulos anteriores. Estos datos son recogidos en la clase action y convertidos al tipo `Class[]` mediante la ejecución de la siguiente instrucción como ya se explicó en el capítulo XXX:

```
Class[] parameterTypes = getParameterTypes(parametros);
```

A continuación, se debe llamar al método que toma los valores de los argumentos recibidos en la llamada y los guarda en la sesión:

```
setParametrosSesion(parameterTypes);
```

porque, finalmente, la función que ejecuta el método correspondiente tomará de la sesión del usuario el valor del argumento previamente guardado. De esta manera, conseguimos que los parámetros iguales sean mantenidos en llamadas a funciones diferentes siempre que utilicen el mismo nombre.

Los siguientes métodos trabajan en este sentido:

- **void `setParametrosSesion(Class[] parameterTypes):`** Es un método privado que recibe los tipos de los argumentos que debe analizar y, si existen, los guarda en la sesión del usuario. Junto con el tipo de los parámetros, la función debe obtener tanto los nombres como los valores de los argumentos de dos atributos de tipo `String[]` que son pasado, o no, en la request, junto con el nombre y los tipos de los argumentos a invocar. Si no se mandan estos datos, la función no realiza ningún trabajo:

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

  Para cada tipo de dato requerido de la función se localiza el nombre del argumento correspondiente:

  ```
  for (int i = 0; i < parameterTypes.length; i++) {
       String atributo = filtroArgumentos[i];
  ```

Si el valor del argumento es distinto de la cadena vacía:

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

Se asigna a una variable de tipo Object el valor correspondiente casteado convenientemente. Para poder castear adecuadamente el valor que llega en formato String al tipo de dato requerido se utiliza una estructura switch:

```
switch (parameterTypes[i].getName()) {
```

Cuando el tipo de datos es un array de Strings se crea un array de Strings utilizando la coma como separador de elementos y deshechando los elementos vacíos:

```
case "[Ljava.lang.String;":
  String[] valores = filtroValores[i].split(",",0);
  List<String> lvalores = new ArrayList();
  for (int j = 0; j< valores.length; j++)
    if (!"".equals(valores[j]))
     lvalores.add(valores[j].trim());
  valor = lvalores.toArray(valores);
  break;
```

Para otros tipos de datos simples, se utiliza la clase correspondiente para hacer el casteado desde String. Por ejemplo, para tipo de dato boolean:

```
case "boolean":
  valor = Boolean.valueOf(filtroValores[i]); break;
```

En caso contrario, cuando el tipo sea String o desconocido, se asigna directamente el valor correspondiente:

```
default: valor = filtroValores[i];
```

Finalmente, se añade el valor a la sesión del usuario:

```
sesion.put(atributo, valor);
```

- **List getArrayParametros()**: Esta clase es utilizada en la página jsp para generar el array de parámetros que se deben incluir en la invocación del método. Una vez que se ha determinado el método, se puede conocer el

array de parámetros que se deben incluir en su invocación mediante el siguiente código:

```
method.getParameters();
```

que se almacena en la variable global:

```
Parameter[] arrayParametros
```

El método, transforma el array en formato List para su mejor tratamiento en un `iterador struts:`

```
List camposFiltro = new ArrayList();
for (Parameter p : arrayParametros) {
  camposFiltro.add(p.getName());
}
return camposFiltro;
```

- **Object getParameter(int i)**: Este método devuelve desde la sesión del usuario el valor almacenado para el parámetro de posición `i`:

```
return sesion.get(arrayParametros[i].getName());
```

Filtro.jsp

Carpeta /WEB-INF/jspf

Esta página visualiza los campos que conforman el filtro para la ejecución del correspondiente método. Dentro de un bloque form, puesto que estos datos serán enviados en una request:

```
<s:form name="filtro" method="POST" theme="simple">
```

Lo primero es incluir tanto el método, como los tipos de los argumentos que requiere. Se oculta puesto que información irrelevante para el usuario:

```
<s:hidden name="metodo"/>
<s:hidden name="parametros"/>
```

y mediante la utilización de una lista, se recorre el iterador `arrayParametros`. Se define una variable counter que nos permitirá llevar un contador del argumento en el que nos encontramos:

```
<ul>
```

```
    <s:set var="counter" value="0"/>
    <s:iterator value="arrayParametros">
        ...
    </s:iterator>
</ul>
```

Dentro del iterador, cada elemento de la lista contiene la etiqueta del nombre del argumento correspondientemente idiomatizada, un campo oculto (`filtroArgumentos`) con el nombre del argumento (sin idiomatizar) y un campo visible (`filtroValores`) con el valor del argumento obtenido mediante la función `getParameter`:

```
<li>
    <s:set var="filtro"><s:property /></s:set>
    <s:text name="%{filtro}" />:
    <s:hidden name="filtroArgumentos" value="%{filtro}"/>
    <s:textfield name="filtroValores"
                 value="%{getParameter(#counter)}"/>
</li>
```

Se completa el iterador con el incremento del contador

```
<s:set var="counter" value="%{#counter+1}"/>
```

y se completa el formulario con un botón de submit

```
<s:submit name="filtrar"/>
```

DatabaseMetaData.jsp

carpeta /

El código que a continuación se detalla ha sido adaptado del indicado por Wanderson López en la web http://wanderlp.com/tabla-con-ordenamiento-utilizando-css-y-jquery/

El fichero DatabaseMetaData.jsp es modificado para incluir referencias que permitan realizar la ordenación de la tabla html que en él se representa.

Para que la ordenación pueda llevarse a cabo, la tabla html debe estar correctamente construida lo que lleva a diferenciar la zona de cabecera de la zona de cuerpo mediante los tags `<thead>` y `<tbody>` y los tags `<th>` y `<td>`

Para ello debemos modificar la página añadiendo el siguiente código:

```
<s:iterator value="listInfo" var="record"
            status="status">
   <s:if test="#status.first == true"><thead></s:if>
        ...
   <s:if test="#status.first == true"></thead></s:if>
</s:iterator>
```

que marca la zona de cabecera solo en el primer registro del iterador

Se debe incluir la referencia a la biblioteca jquery utilizada para realizar la ordenación mediante el siguiente código en el bloque <head>:

```
<script
src="${pageContext.request.contextPath}/resources/jquery-
3.3.1.min.js">
</script>
```

Con la utilización de `pageContext.request.contextPath` conseguimos que se localice el javascript en la carpeta correcta independientemente de la ubicación del fichero jsp.

El método de ordenación de la tabla consiste en tres fases:

1. Enviar el contenido de la tabla a un arreglo
2. Ordenar el arreglo
3. Mostrar en el encabezado de la tabla cual es el orden aplicado

Ordenar tabla

Este paso debe realizarse en respuesta a una acción clic en algún elemento de la cabecera de la tabla. Para ello, programamos una función que responda a un evento clic sobre el objeto th:

```
$('th').click(function() {
   ...
});
```

Carga la variable table con el objeto html padre table y la tabla rows con las filas mayores que 0 y ordenadas

```
var table = $(this).parents('table').eq(0)
   var rows = table.find('tr:gt(0)').toArray()
```

```
        .sort(comparer($(this).index()))
```

La función comparer permite comparar los valores de la tabla entre sí:

```
function comparer(index) {
   return function(a, b) {
        var valA = getCellValue(a, index),
        valB = getCellValue(b, index)
        return $.isNumeric(valA) && $.isNumeric(valB)
          ? valA - valB : valA.localeCompare(valB)
   }
}
```

para lo que utiliza la función getCellValue que obtiene los valores de la celda:

```
function getCellValue(row, index) {
   return $(row).children('td').eq(index).html()
}
```

En la función original del evento click, una variable mantiene el tipo de ordenación establecido:

```
this.asc = !this.asc
if (!this.asc) {
   rows = rows.reverse()
}
```

Finalmente añade a la tabla las filas en el orden correcto

```
for (var i = 0; i < rows.length; i++) {
table.append(rows[i])
}
```

y con la siguiente llamada visualiza el icono que le indicará al usuario el campo y el tipo de ordenación establecido:

```
setIcon($(this), this.asc);
```

Esta función muestra gráficamente qué ordenamiento se está aplicando asignando o desasignando clases css a los distintos componentes th de la cabecera de la tabla.

```
function setIcon(element, asc) {
   $("th").each(function(index) {
        $(this).removeClass("sorting");
```

```
        $(this).removeClass("asc");
        $(this).removeClass("desc");
    });
    element.addClass("sorting");
    if (asc) element.addClass("asc");
    else element.addClass("desc");
}
```

Mostrar cuál es el orden aplicado

Para conseguir este efecto se recurre a la utilización de CSS.

En primer lugar, el cursor debe ser un puntero cuando apunte a la cabecera de la tabla. Si no, aparecerá un cursor:

```
table tr th {
    cursor: pointer;
    -webkit-user-select: none;
    -moz-user-select: none;
    -ms-user-select: none;
    user-select: none;
}
```

La cabecera de la columna utilizada para la ordenación se diferencia del resto mediante un color de fondo diferente:

```
.sorting {
    background-color: #D4D4D4;
}
```

y se incluirá una flecha hacia arriba o hacia abajo para indicar el tipo de ordenación ascendente o descendente:

```
.asc:after {
    content: ' ↑';
}
.desc:after {
    content: " ↓";
}
```

Todo esto se añade en la sección <head> mediante tags <style>

5.5. 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.

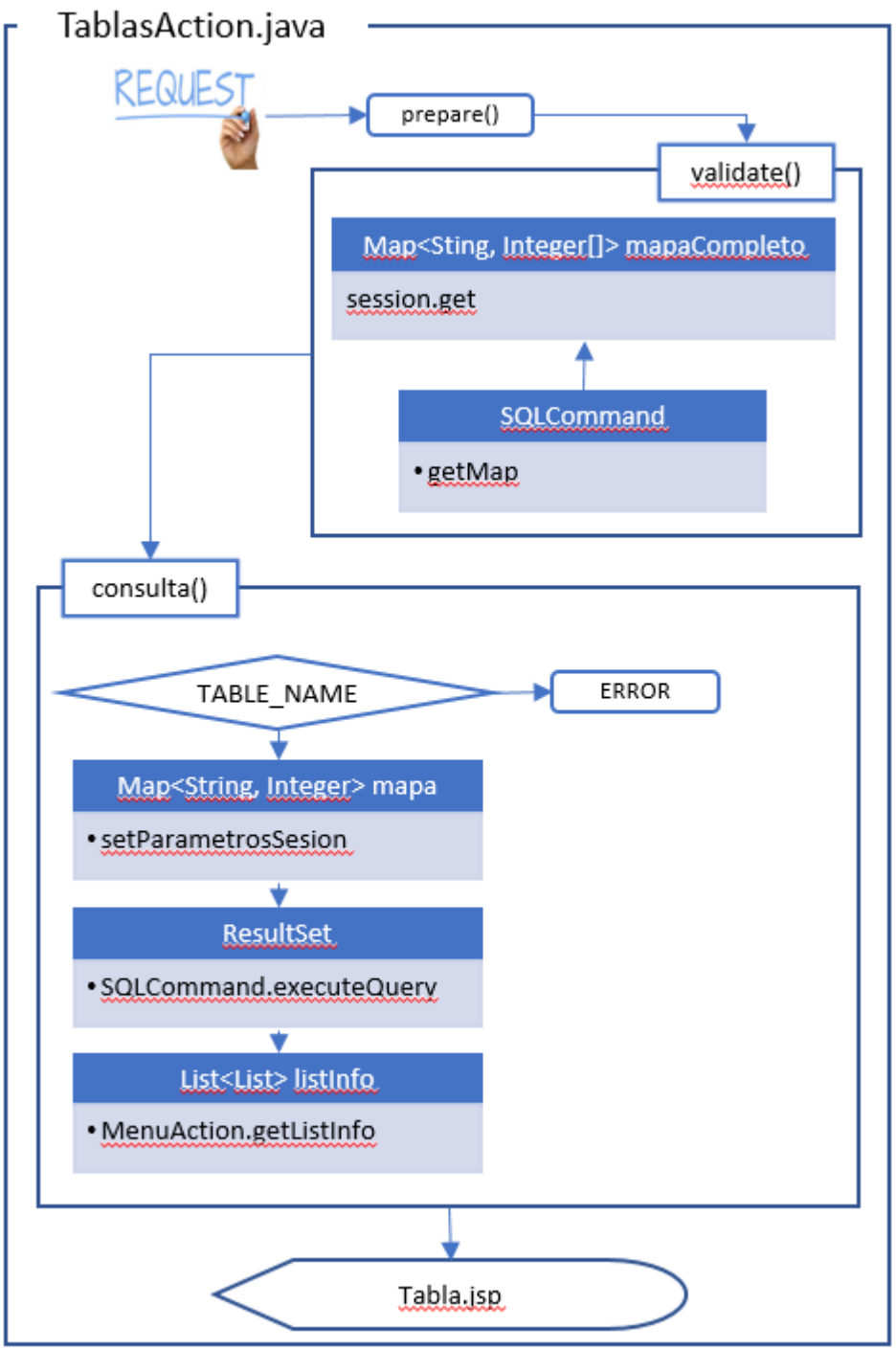


Figura 5.9: Lógica de consulta de una tabla

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.

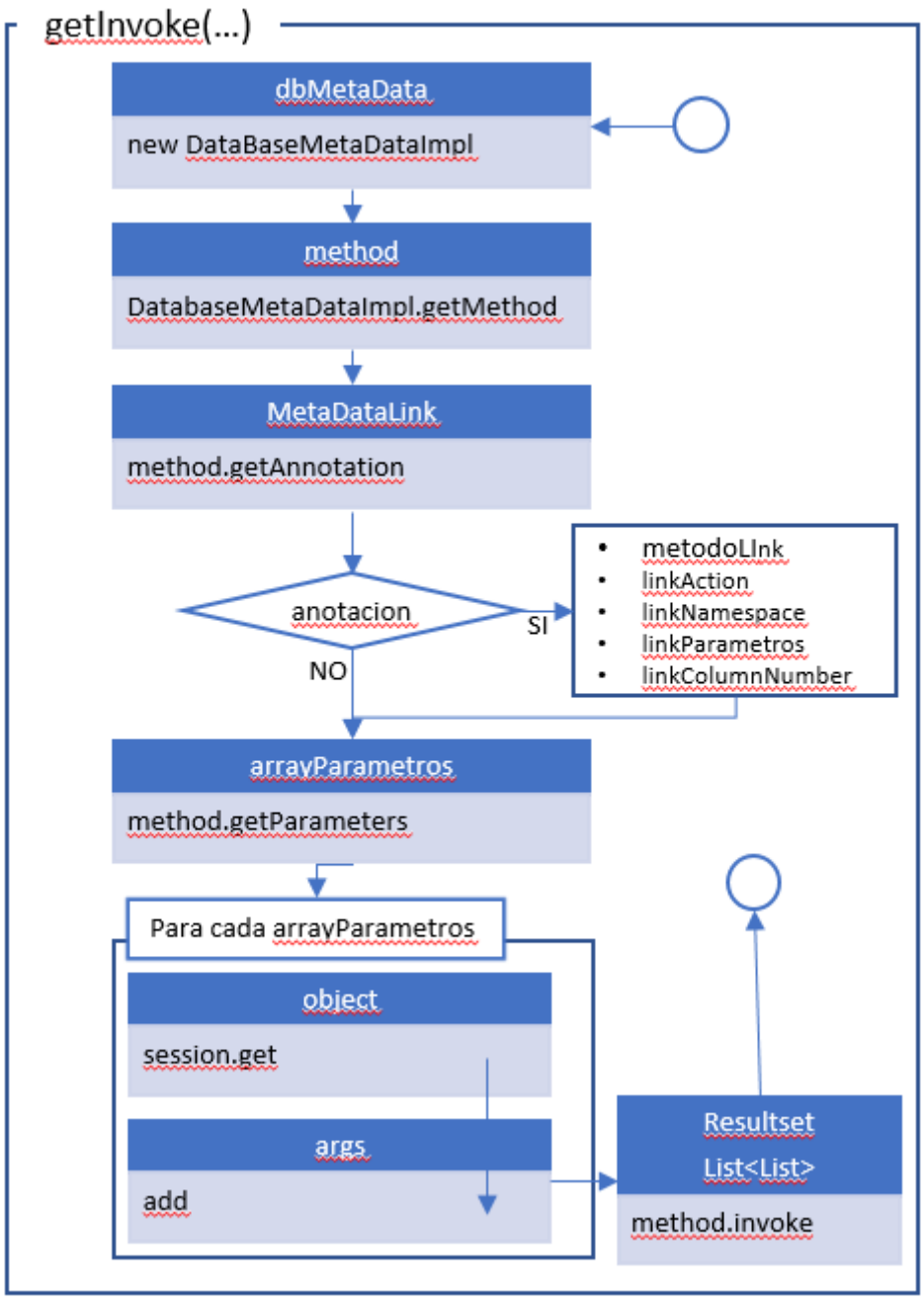


Figura 5.10: getInvoke para obtener resultsets o listas

- **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:

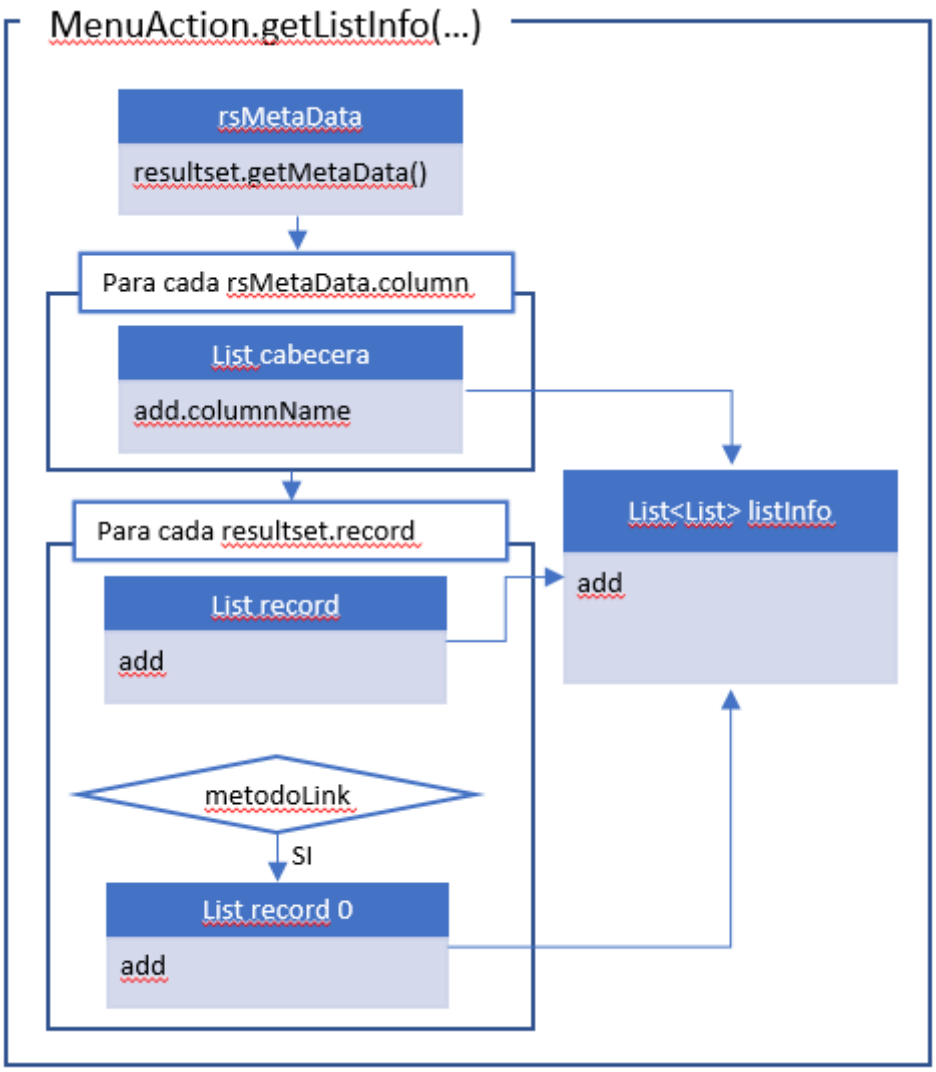


Figura 5.11: getListInfo. Obtención de lista a partir de resultset

```
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 el 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 && #stat.count != 1">
       <th><s:property /></th>
```

```
    </s:if>
    <s:elseif test="#stat.count != 1">
        <th><s:property /></th>
    </s:elseif>
<s:elseif test='#propiedad != null'>
    <th class="ayuda"><s:property/></th>
</s:elseif>
```

Para el resto de los 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">
     rO0ABXVyABJbTGphdmEubGFuZy5DbGFzczurFteuy81amQIAAHhw
     AAAABHZyABBqYXZhLmxhbmcuU3RyaW5noPCkOHo7s0ICAAB4cHEA
     fgADcQB-
     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[]>)sesión
  .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"/>
```

5.6. 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 las 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">
         
   </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;
```

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

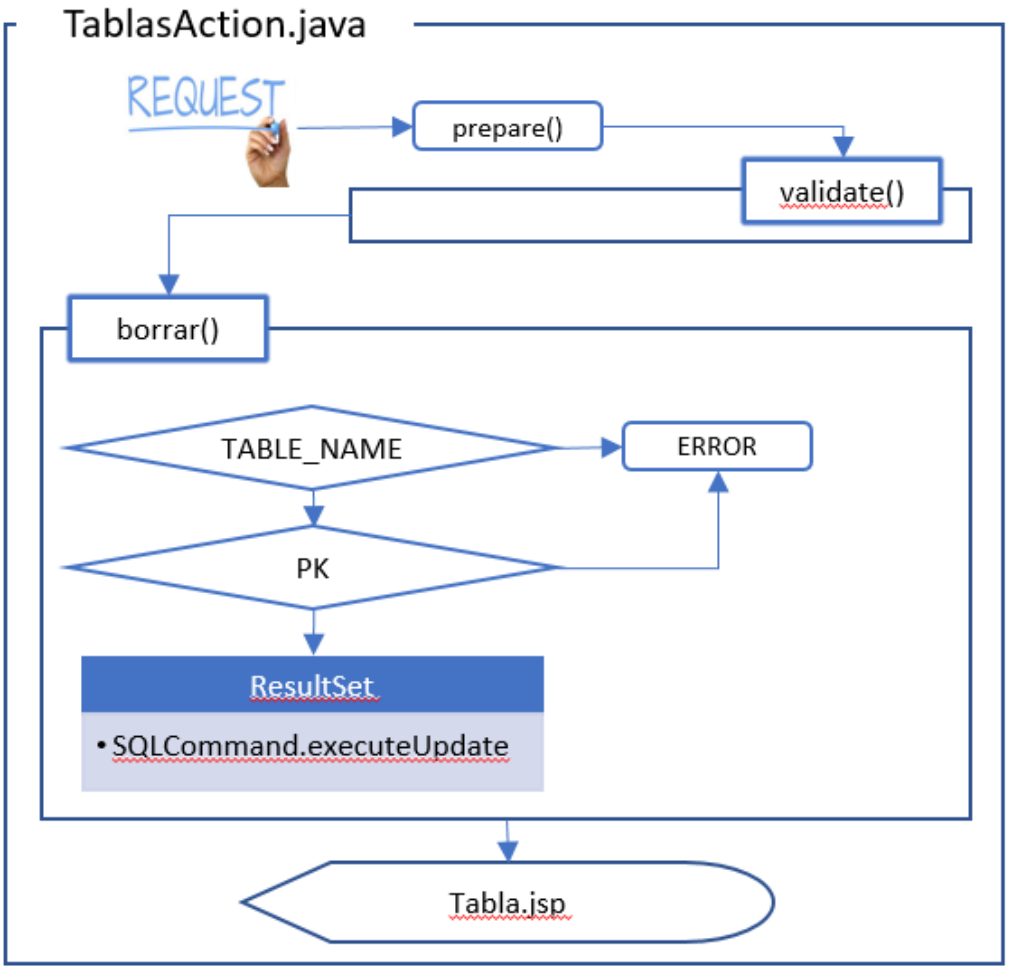


Figura 5.12: Lógica de la eliminación de un registro

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">
     rO0ABXVyABJbTGphdmEubGFuZy5DbGFzczurFteuy81amQIAAHhw
     AAAABHZyABBqYXZhLmxhbmcuU3RyaW5noPCkOHo7s0ICAAB4cHEA
     fgADcQB-
     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:

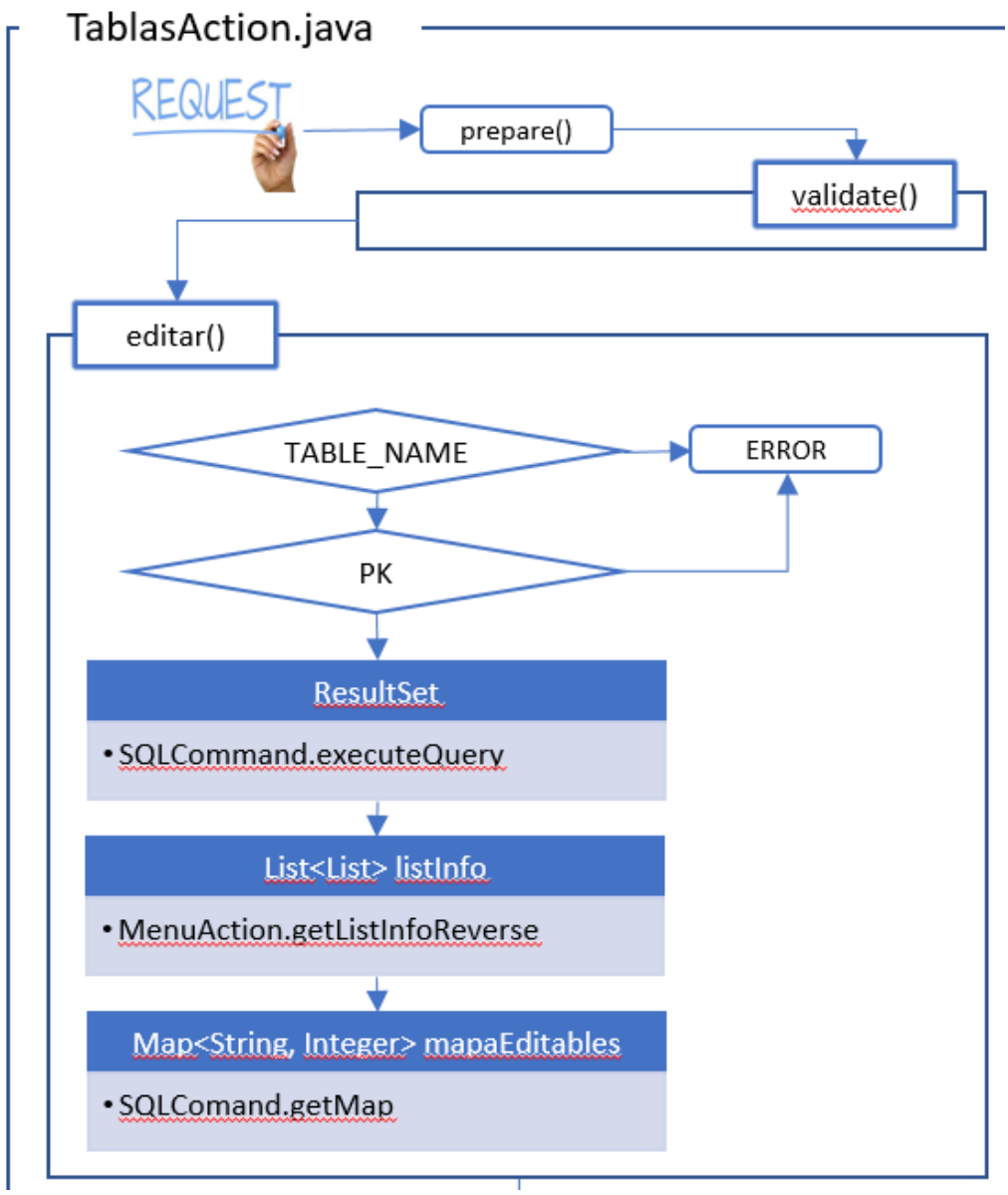


Figura 5.13: Lógica de edición de registro

```
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)) {
  sesión.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"/>
```

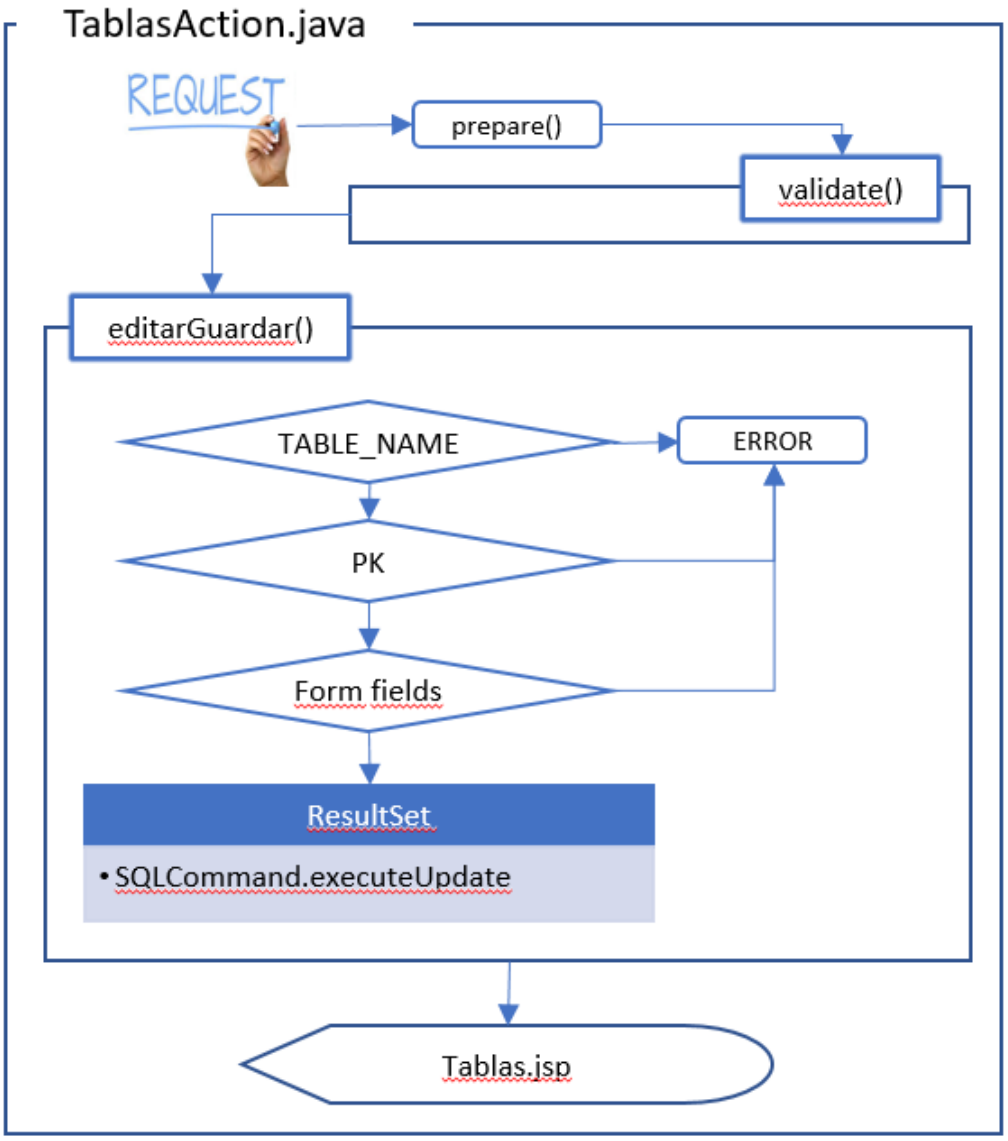


Figura 5.14: Lógica para guardar un registro editado

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=CONTRAT
O&TABLE_TYPE=TABLE
```

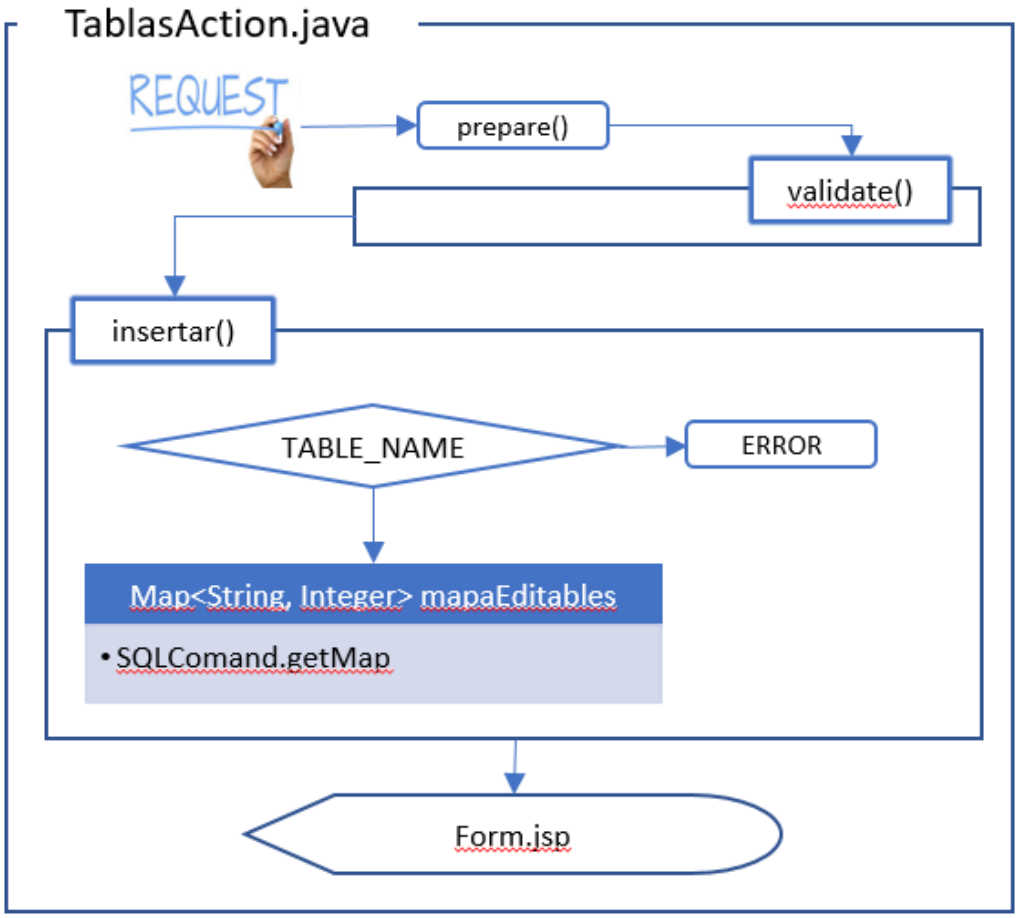


Figura 5.15: Lógica de inserción de un registro

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>
```

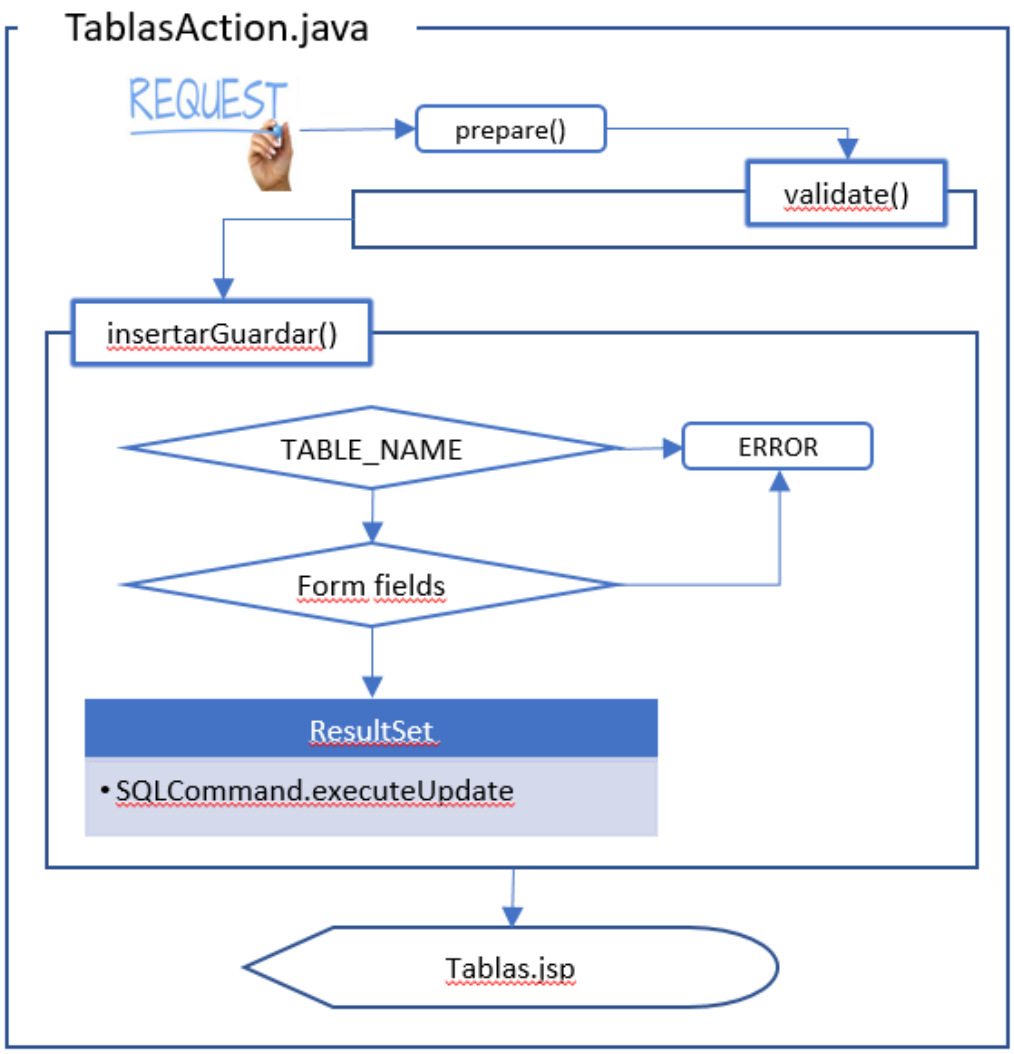


Figura 5.16: Lógica de guardado en la inserción de un registro

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"/>
```

Trabajos relacionados

Haciendo un repaso rápido por internet localizamos las siguientes aplicaciones, todas ellas de corte comercial, que permiten una interacción con los datos de bases de datos relacionales.

Se ha obviado de la búsqueda las herramientas propietarias de los fabricantes de bases de datos puesto que entendemos que no permiten la conexión con sistemas diferentes al del propio fabricante.

6.1. Herramientas de administración de bases de datos

Existe, como ya indicamos, una gran variedad de herramientas de distintos fabricantes, que permiten realizar infinidad de operaciones en distintas bases de datos. A modo de resumen presentamos una serie de cuadros donde se pueden ver distintos aspectos de cada uno de ellos y compararlos unos con otros.

Los datos han sido obtenidos de https://en.wikipedia.org/wiki/Comparison_of_database_tools

Descripción General

A modo de resumen, se indican los aspectos más importantes de cada herramienta:

Producto	Windows	Mac OS	Linux	Oracle	MySQL	Postgre SQL	MS SQL Server	ODBC	JDBC	SQLite
Adminer	Sí	Sí	Sí	Sí	Sí	Sí	Sí			Sí
DaDaBIK	Sí	Sí	Sí	Sí	Sí	Sí	Sí	No	No	Sí
Database Deployment Manager	Sí	No	Sí		Sí					
Database Spy	Sí	No	No	Sí	Sí	Sí	Sí	Sí	Sí	
Database Workbench	Sí	requiere wine	requiere wine	Sí	Sí		Sí	Sí		
DataGrip	Sí	Sí	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí
DBeaver	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
DBEdit	Sí	Sí	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí

Producto	Windows	Mac OS	Linux	Oracle	MySQL	Postgre SQL	MS SQL Server	ODBC	JDBC	SQLite
Epictetus	Sí	Sí	Sí	Sí		Sí	Sí			
HeidiSQL	Sí	requiere wine	requiere wine		Sí	Sí	Sí			
Maatkit	Sí	Sí	Sí		Sí					
Microsoft SQL Server Management Studio	Sí	No	No				Sí			
ModelRight	Sí	No	No	Sí	Sí		Sí	Sí		
MySQL Workbench	Sí	Sí	Sí		Sí					
Navicat	Sí	Sí	requiere wine	Sí	Sí	Sí	Sí	Sí		Sí
Navicat Data Modeler	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí		Sí

Producto	Windows	Mac OS	Linux	Oracle	MySQL	Postgre SQL	MS SQL Server	ODBC	JDBC	SQLite
Oracle Enterprise Manager	Sí	No	Sí	Sí	Sí		Sí			
Oracle SQL Developer	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí	
Orbada	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
pgAdmin III	Sí	Sí	Sí			Sí				
phpLiteAdmin	Sí	Sí	Sí	No	No	No	No	No	No	Sí
phpMyAdmin	Sí	Sí	Sí		Sí					
SQL Database Studio	Sí	No	No	No	No	No	Sí			
SQLyog	Sí	requiere wine	requiere wine		Sí					
SQuirreL SQL	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí

Producto	Windows	Mac OS	Linux	Oracle	MySQL	Postgre SQL	MS SQL Server	ODBC	JDBC	SQLite
Toad	Sí	No	No	Sí	Sí		Sí	Sí		
Toad Data Modeler	Sí	No	No	Sí	Sí	Sí	Sí			
TORA	Sí	Sí	Sí	Sí	Sí	Sí				

Tabla 6.1: Estado del arte

Características básicas

A continuación se indican qué herramientas son de tipo escritorio, en entorno gráfico, o basados en navegador.

Herramientas	Interfaz de usuario
Adminer	Basado en navegador
Altova DatabaseSpy	escritorio
DaDaBIK	Basado en navegador
Database Deployment Manager	escritorio
Database Spy	
Database Workbench	escritorio
DataGrip	escritorio
DBeaver	escritorio
DBEdit	escritorio
Epictetus	escritorio
HeidiSQL	
Maatkit	
Microsoft SQL Server Management Studio	escritorio
ModelRight	escritorio
MySQL Workbench	escritorio
Navicat	escritorio

Herramientas	Interfaz de usuario
Navicat Data Modeler	escritorio
Oracle Enterprise Manager	Basado en navegador
Oracle SQL Developer	escritorio
Orbada	escritorio
pgAdmin III	TDI
phpLiteAdmin	Basado en navegador
phpMyAdmin	Basado en navegador
SQL Database Studio	escritorio
SQLyog	escritorio
SQuirreL SQL	escritorio
Toad	escritorio
Toad Data Modeler	escritorio
TORA	escritorio

Tabla 6.2: Interfaz de usuario de herramientas

Si nos centramos exclusivamente en las herramientas basadas en navegador, podemos ver las siguientes características comparadas entre ellas:

Herramientas	Adminer	DaDaBIK	Oracle Enterprise Manager	phpLiteAdmin	phpMyAdmin
Asistente para crear y modificar					
Base de datos	Sí	No	Sí	Sí	Sí
Tabla	Sí	Algunos	Sí	Sí	Sí
Procedimiento	Sí	No	Sí	No	Sí
Visor					
Trigger	Sí	No	Sí	Sí	Sí
Base de datos	Sí	No	Sí	Sí	Sí
Tabla	Sí	Algunos	Sí	Sí	Sí
Procedimiento	Sí	No	Sí	No	Sí
Trigger	Sí	No	Sí	Sí	Sí

Herramientas	Adminer	DaDaBIK	Oracle Enterprise Manager	phpLiteAdmin	phpMyAdmin
Autocompletar	No	No	Sí	No	Sí
Sintaxis coloreada	Sí	No	Sí	No	Sí
Soporte multi servidor	?	No	?	?	Sí
Servidor de monitoreo	?	?	Sí	?	Sí
Administrador de usuarios	Sí	Algunos			Sí
Enchufar	Sí	No			Algunos
Comparar	Sí	No			Sí
Importar	[1]	No			[2]

[1] Script SQL, CSV, TSV o lo anterior en zip (como complemento); importaciones de archivos de sitio de servidor en SQL o SQL en zip, gzip o bzip2
[2] Sí - CSV , SQL , XML , Excel , ODS

Herramientas	Adminer	DaDaBIK	Oracle Enterprise Manager	phpLiteAdmin	phpMyAdmin
Exportar	[3]	CSV			[4]
Depurador	No	Sí			Sí
Fuente de control	Git	No			Git
Visión espacial					
Generador de consultas visuales	Sí	Algunos			Sí
Esquema visual / modelo / diseño de diagrama ER	Sí	No			Sí
Ingeniería inversa	Sí	No			Sí
Ingeniería delantera	No	No			No

[3] Script SQL, CSV, TSV o lo anterior en zip , gzip , bzip2 ; XML (como complemento)
[4] Sí, CSV , LaTeX , Excel , Word , ODS , ODT , XML , SQL , YAML , Texy! , JSON , NHibernate , PHP , PDF ,MediaWiki

Herramientas	Adminer	DaDaBIK	Oracle Enterprise Manager	phpLiteAdmin	phpMyAdmin
Diagrama de ER cuadros de grupo	No	No			No

Tabla 6.3: Características de herramientas basadas en navegador

De la tabla podemos deducir que la herramienta más completa es phpMyAdmin. En los cuadros anteriores veíamos que esta herramienta estaba solo diseñada para la gestión de la base de datos MySQL, y que no disponía de enlace con las demás bases de datos, ni a través de ODBC ni a través de JDBC.

A continuación destacaremos la herramienta Adminer, disponible para las bases de datos más importantes, pero sin conexiones JDBC.

Podríamos, por tanto, partir de la herramienta phpMyAdmin para dotarla de conexión a bases de datos JDBC y, en la medida que nos de tiempo, intentar realizar las funcionalidades que podamos indicadas en esta tabla.

6.2. phpMyAdmin

https://www.phpmyadmin.net/

Se trata de una herramienta gratuita, desarrollada en php que permite la administración de la base de datos MySQL vía web.

Permite las operaciones de uso frecuente como gestión de bases de datos, tablas, columnas, relaciones, índices, usuario y permisos junto con la capacidad de ejecutar directamente cualquier instrucción SQL.

Con una amplia documentación, los usuarios pueden colaborar para compartir ideas o procedimientos.

Entre las características más importantes disponemos de una interfaz web bastante intuitiva que soporta la mayoría de las características de la base de datos a la que da soporte. Además incluye la posibilidad de importar y exportar datos en distintos formatos, administrar múltiples servidores, crear gráficos del diseño de la base de datos, crear consultas complejas, buscar globalmente en una base de datos o en un conjunto de ellas, mostrar datos BLOB como imagen y más.

6.3. Adminer

https://www.adminer.org/

Se trata de una herramienta de administración desarrollada también en PHP. A diferencia de phpMyAdmin se trata de un solo archivo preparado para

implementarse directamente en el servidor de destino y disponible para la mayor parte de las bases de datos más representativas del mercado.

Según el propio desarrollador, mejora a phpMyAdmin en las siguientes cuestiones fundamentales:

- Experiencia de usuario
- Funciones de MySQL soportadas
- Actuación
- Seguridad

Junto con la herramienta de administración, también está disponible el **Editor de Adminer**, también escrita en PHP y adecuada para usuarios comunes que proporciona manipulación de datos de alto nivel.

Permite utilizar un "conjunto separado de credenciales de inicio de sesión diferentes del usuario y la contraseña de la base de datos real", aunque solo se maneja una base de datos por cada instancia de Editor.

Permite mostrar imágenes almacenadas en BLOB, mostrar casillas de verificación e iconos para campos booleano, utilizar formatos nacionales para fechas, enviar mensajes a correos electrónicos encontrados en tablas pero sin edición o visualización de comandos SQL.

6.4. Oracle Enterprise Manager

Esta "herramienta" no será incluida en este capítulo puesto que en realidad se trata de un conjunto completo de herramientas que pasan por la gestión de aplicaciones empaquetadas, de middleware o de base de datos y llegan a gestión de la nube, monitoreo de empresas o rendimiento de de aplicaciones.

6.5. Herramientas de escritorio

Las herramientas descritas a continuación son todas ellas de fabricante independiente, con posibilidad de conexión a la mayor parte de las bases de datos actuales y con interfaz gráfico o de escritorio, lo que les otorga una posibilidades

visuales y de interacción con el usuario que no tienen comparación posible con las indicadas en los apartados anteriores

ValentinaDB

https://www.valentina-db.com/en/valentina-studio-overview

Dentro del grupo de fabricantes de bases de datos, ValentinaDB desarrolla una herramienta de administración universal que permite trabajar con sistemas de distintos fabricantes, algunos tan conocidos como MySQL, SQL Server o PostgreSQL.

Permite la creación de formularios de manera visual, la transferencia de datos entre bases de datos diferentes, la comparación de esquemas de bases de datos diferentes generando automáticamente los scripts necesarios para pasar de uno a otro.

Dispone herramientas de modelado que interactúan visualmente con el usuario así como editores de informes para transformar consulta en informes empresariales con una gran calidad visual y un conjunto completo de elementos.

También dispone de un generador visual de consultas que permite trabajar directamente con tablas y enlaces así como de un editor SQL con funciones completas de edición.

DbVisualizer

https://www.dbvis.com/features/tour/command-line-interface/

Se trata de una herramienta independiente de fabricantes, que permite conectarse con una gran cantidad diferente de bases de datos entre las que destacan las más importantes del mercado. Además es multientorno, corriendo en windows,mac y unix

Igual que todos los demás, dispone de una interfaz gráfica, con profusión de pantallas, divisiones y pestañas lo que les permite mostrar una gran cantidad de información relacionada en una sola pantalla.

Las conexiones a las bases de datos son flexibles y se basan en JDBC, por lo que es conectable prácticamente a cualquier base de datos que disponga de

controladores JDBC, que son casi todas en la práctica. Se permiten conexiones físicas separadas para proporcionar aislamiento de transacciones.

Con un gran editor SQL, se permite el uso de SQL parametrizado, generador visual de consultas, plan de explicación/ejecución e incluso una interfaz basada en línea de comandos si fuera necesaria. Se le añade un historial de SQL y la visualización de resultados mediante gráficos.

Continuando con las herramientas visuales, dispone de un gestor de objetos de base de datos que permite editar, compilar y ejecutar procedimientos, funciones, triggers, etc. También permite la administración de objetos de la base de datos, desde tablas a paquetes o módulos mediante una navegación basada en árbol a través de los objetos mostrando objetos y jerarquías específicos de la base de datos con la posibilidad de incluir filtros y mostrar los detalles de cualquier objeto.

En cuanto a la gestión pura de datos, dispone de un editor de datos de tabla tipo hoja de cálculo, incluyendo datos binarios (BLOB y CLOB). Además permite la exportación de objetos de base de datos y tablas mediante sentencias tipo CREATE, INSERT, CSV o XML

Si fuera poco, disponemos de herramientas de administración de la base de datos en aspectos tales como almacenamiento de sesión, administración de seguridad, etc. También existe la posibilidad de ejecución en segundo plano de tareas con un alto coste computacional o de entrada/salida

SQuirreL SQL

http://squirrel-sql.sourceforge.net/

Herramienta independiente de fabricante. También se permite el trabajo mediante herramientas gráficas.

El editor SQL permite la finalización de código de forma contextual con una simple combinación de teclas.

Se permite la configuración de atajos a las distintas herramientas para una localización rápida y sencilla.

Dispone de un amplio conjunto de plantillas de código predefinidas para las sentencias SQL y DDL más comunes.

Permite la creación automática de tablas a partir de los resultados de consultas SQL y, por supuesto, herramientas para crear nuevos gráficos o añadirles tablas a los ya existentes

Aquafold

https://www.aquafold.com/aquadatastudio

Según se indica por el fabricante, permite desarrollar, acceder, gestionar y analizar visualmente los datos, ya se encuentre en bases de datos relacionales, no SQL o en la nube.

Igual que los demás, dispone de un entorno de espacio de trabajo con marcos de acoplamiento y ventana, con pestañas que pueden agruparse o flotar para mayor flexibilidad.

Dispone de formateador de SQL, coloreado en función de la sintaxis y funciones de autocompletar para ahorro de tiempo al escribir. Visualización de sentencias en cuadrícula, texto, pivote o formulario con rápidos filtrados y posibilidad de exportación. También dispone de historial de sql.

El ya clásico generador visual de consultas dotado además de la adición de parámetros de consulta permite generar una declaración SQL completa.

Se puede mostrar un plan de ejecución para una consulta para mejorar su rendimiento.

Para algunas bases de datos comerciales importantes como Oracle, Sybase o SQL Server, “se permite rastrear y depurar procedimientos almacenados para identificar cualquier problema de desarrollo o producción.”

Se le dota a la herramienta de una shell interactiva combinando la línea de comandos SQL con la versatilidad de una shell Unix.

Dispone un cliente de control de versión completamente integrado para Subversion, CVS o Git.

También, por supuesto, dispone de una herramienta analítica visual que permite identificar patrones y tendencias en los datos, uso de filtros, etiquetas, gráficos, etc. y un editor de datos de tabla mediante una cuadrícula editable similar a Excel, herramientas de importación y exportación en diferentes formatos de distintos tipos de objetos, examinador de objetos con edición visual, entorno de desarrollo de secuencias de comandos y herramientas de administración para las bases de datos más importantes del momento, incluyendo sistemas de Log, rollback, etc.

Herramientas de comparación y sincronización de objetos de esquema de diferentes bases de datos en interfaz gráfica, y modelador Entidad/Relacion con ingeniería inversa

6.6. Conclusiones

A la vista de los productos disponibles podemos extraer las siguientes conclusiones:

1. La gran mayoría combinan la gestión de los datos de la base de datos con funciones de administración.
2. Existe una amplia mayoría de productos en formato escritorio, aunque tampoco son menos importantes aquellos cuyo entorno de trabajo es entorno web.
3. No existe una amplia variedad de productos que realicen sus conexiones a la base de datos mediante JDBC y, cuando esto ocurre, las conexiones realizadas son propietarias de la aplicación o no se ha indicado el sistema de conexión.

De estas conclusiones podríamos extraer que sería interesante realizar un proyecto en aquello en lo que se encuentran menos soluciones por lo que me decantaría por lo inicialmente propuesto:

- Entorno web
- Servidor web tomcat
- Conexiones JDBC a bases de datos, con lo que se cubre un amplio espectro de bases de datos disponibles.
- Desarrollo de todas las funcionalidades JDBC disponibles en el estándar, en la medida que nos de tiempo a realizarlas.

- Aprovechamiento de herramientas de desarrollo tipo Modelo-Vista-Controlador, planteándose inicialmente el uso de struts y java
- Aprovechamiento del estándar JSPs junto con HTML5, css, javascript y Ajax para conseguir una interface moderna y actual con el usuario

Conclusiones y Líneas de trabajo futuras

Falta por desarrollar

Bibliografía

[1] Stack Exchange, Inc., «What is the difference between javaee-api and javaee-web-api?,» 2013. [En línea]. Available: https://stackoverflow.com/questions/16789020/what-is-the-difference-between-javaee-api-and-javaee-web-api.

[2] Fundación Wikimedia, Inc., «Java EE,» 2018. [En línea]. Available: https://es.wikipedia.org/wiki/Java_EE.

[3] Fundación Wikimedia, Inc., «Java Servlet,» 2018. [En línea]. Available: https://es.wikipedia.org/wiki/Java_Servlet.

[4] Programación en castellano, S.L., «Servlets básico,» [En línea]. Available: https://programacion.net/articulo/servlets_basico_108/2.

[5] Fundación Wikimedia, Inc., «Java Server Page,» 12 06 2018. [En línea]. Available: https://es.wikipedia.org/wiki/JavaServer_Pages.

[6] Oracle Corporation, «Java JDBC API,» 2018. [En línea]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/.

[7] Oracle Corporation, «Industry Support,» [En línea]. Available: https://www.oracle.com/technetwork/java/index-136695.html.

[8] Oracle Corporation, «JDBC Overview,» [En línea]. Available: https://www.oracle.com/technetwork/java/overview-141217.html.

[9] Oracle Corporation, «Features,» [En línea]. Available: https://www.oracle.com/technetwork/java/features-135464.html.

[10] L. Hernández, «API JDBC como interfaz de acceso a bases de datos SQL,» 28 9 2004. [En línea]. Available: http://www.iuma.ulpgc.es/users/lhdez/inves/pfcs/memoria-ivan/node8.html.

[11] chuidiang, «DataBaseMetaData y ResultSetMetaData con java y MySQL,» 4 2 2007. [En línea]. Available: http://www.chuidiang.org/java/mysql/ResultSet-DataBase-MetaData.php.

[12] Fundación Wikipedia, inc., «Tomcat,» 15 12 2018. [En línea]. Available: https://es.wikipedia.org/wiki/Tomcat.

[13] Apache Software Foundation, «Tomcat Setup,» 9 11 2018. [En línea]. Available: https://tomcat.apache.org/tomcat-7.0-doc/setup.html.

[14] Apache Software Fundatio, «Tomcat Web Application Deployment,» 9 11 2018. [En línea]. Available: https://tomcat.apache.org/tomcat-7.0-doc/deployer-howto.html.

[15] Apache Software Foundation, «Manager App HOW-TO,» 9 11 2018. [En línea]. Available: https://tomcat.apache.org/tomcat-7.0-doc/manager-howto.html.

[16] Apache Software Foundation, «JNDI Datasource HOW-TO,» 9 11 2018. [En línea]. Available: https://tomcat.apache.org/tomcat-7.0-doc/jndi-datasource-examples-howto.html.

[17] Apache Software Foundation, «Class Loader HOW-TO,» 9 11 2018. [En línea]. Available: https://tomcat.apache.org/tomcat-7.0-doc/class-loader-howto.html.

[18] Apache Software Foundation, «Connectors How To,» 9 11 2018. [En línea]. Available: https://tomcat.apache.org/tomcat-7.0-doc/connectors.html.

[19] Apache Software Foundation, «Security Considerations,» 9 11 2018. [En línea]. Available: https://tomcat.apache.org/tomcat-7.0-doc/security-howto.html.

[20] Wikimedia Foundation, Inc., «Apache Struts 2,» 18 11 2018. [En línea]. Available: https://en.wikipedia.org/wiki/Apache_Struts_2.

[21] Apache Software Foundation, «Key Technologies Primer,» 2018. [En línea]. Available: https://struts.apache.org/primer.html.

[22] Apache Software Foundation, «How To Create A Struts 2 Web Application,» 2018. [En línea]. Available: https://struts.apache.org/getting-started/how-to-create-a-struts2-web-application.html.

[23] Apache Software Foundation, «Apache Maven Compiler Plugin,» 26 7 2018. [En línea]. Available: https://maven.apache.org/plugins/maven-compiler-plugin/.

[24] Apache Software Foundation, «Apache Maven WAR Plugin,» 3 6 2018. [En línea]. Available: https://maven.apache.org/plugins/maven-war-plugin/.

[25] Apache Software Foundation, «Apache Maven Dependency Plugin,» 19 5 2018. [En línea]. Available: https://maven.apache.org/plugins/maven-dependency-plugin/.

[26] Wkimedia Foundation, Inc., «Maven,» 8 1 2019. [En línea]. Available: https://es.wikipedia.org/wiki/Maven.

[27] Wikimedia Foundation, Inc., «Netbeans,» 28 12 2018. [En línea]. Available: https://es.wikipedia.org/wiki/NetBeans.

[28] Apache Software Foundation, «Creating an Enterprise Application Using Maven,» 2018. [En línea]. Available: https://netbeans.org/kb/docs/javaee/maven-entapp.html.