Introducción a la Programación Web con Java: JSP y Servlets, JavaServer Faces

Dra. María del Carmen Gómez Fuentes
Dr. Jorge Cervantes Ojeda

Obra ganadora del Tercer Concurso para la publicación de libros de texto y materiales de apoyo a la impartición de los programas de estudio de las licenciaturas que ofrece la Unidad Cuajimalpa

Introducción a la Programación Web con Java: JSP y Servlets, JavaServer Faces

1) Introducción a la Programación Web con Java: JSP y Servlets, JavaServer Faces
Clasificación Dewey: 005.2762 G66
Clasificación LC: TK5105.8885.J38 G66
Gómez Fuentes, María del Carmen
Introducción a la programación web con Java : JSP y Servlets, JavaServer Faces / María del Carmen Gómez Fuentes, Jorge Cervantes Ojeda. – Ciudad de México : UAM, Unidad Cuajimalpa, 2017.
245 p. : il. col., diagrs. tablas ; 17 x 24 cm.
ISBN: 978-607-28-1069-3
1. Aplicaciones web – Desarrollo – Libros de texto. 2. Java (Lenguaje de programación para computadora) – Manuales. 3. Programación de computadoras – Libros de texto. 4. Informática – Manuales de laboratorio. 5. Universidad Autónoma Metropolitana – Unidad Cuajimalpa – Planes de estudio.
Cervantes Ojeda, Jorge, coaut.

Esta obra fue dictaminada positivamente por pares académicos mediante el sistema doble ciego y evaluada para su publicación por el Consejo Editorial de la UAM Unidad Cuajimalpa.

© 2017 Por esta edición, Universidad Autónoma Metropolitana, Unidad Cuajimalpa
Avenida Vasco de Quiroga 4871
Col. Santa Fe Cuajimalpa, delegación Cuajimalpa de Morelos
C.P. 05348, Ciudad de México (Tel: 5814 6500)
www.cua.uam.mx

ISBN: 978-607-28-1069-3

Primera edición: 2017

Corrección de estilo: Omar Campa
Diseño editorial y portada: Literatura y Alternativas en Servicios Editoriales S.C.
Avenida Universidad 1815-c, Depto. 205, Colonia Oxtopulco,
C. P. 04318, Delegación Coyoacán, Ciudad de México.
RFC: LAS1008162Z1

Ninguna parte de esta obra puede ser reproducida o transmitida mediante ningún sistema o método electrónico o mecánico sin el consentimiento por escrito de los titulares de los derechos.

Impreso y hecho en México
Printed and made in Mexico

Dra. María del Carmen Gómez Fuentes y Dr. Jorge Cervantes Ojeda

Introducción a la Programación Web con Java: JSP y Servlets, JavaServer Faces

UNIVERSIDAD AUTÓNOMA METROPOLITANA

Dr. Eduardo Peñalosa Castro
Rector General

Dr. José Antonio De los Reyes Heredia
Secretario General

Dr. Rodolfo Suárez Molnar
Rector de la Unidad Cuajimalpa

Dr. Álvaro Peláez Cedrés
Secretario de la Unidad Cuajimalpa

Mtro. Octavio Mercado González
Director de la División de Ciencias de la Comunicación y Diseño

Dr. Raúl Roydeen García Aguilar
Secretario Académico de la División de Ciencias de la Comunicación y Diseño

Dr. A. Mauricio Sales Cruz
Director de la División de Ciencias Naturales e Ingeniería

Dr. José Javier Valencia López
Secretario Académico de la División de Ciencias Naturales e Ingeniería

Dr. Roger Mario Barbosa Cruz
Director de la División de Ciencias Sociales y Humanidades

Dr. Jorge Lionel Galindo Monteagudo
Secretario Académico de la División de Ciencias Sociales y Humanidades

Índice

INTRODUCCIÓN 11

IMPORTANCIA DE LOS CONOCIMIENTOS A ADQUIRIR Y LAS HABILIDADES A DESARROLLAR 13

IMPORTANCIA DEL APOYO QUE BRINDA ESTE LIBRO A LAS UEA DE PROYECTO TERMINAL 15

OBJETIVOS 17

1. ¿QUÉ SE NECESITA HACER PARA QUE UNA APLICACIÓN FUNCIONE EN LA RED? 19
1.1 Objetivo 19
1.2 ¿Qué es una aplicación web? 19
1.3 Peticiones y respuestas en las páginas Web 21
1.4 ¿Qué es una JSP? 22
1.5 ¿Qué es un Servlet? 22
1.6 La arquitectura Modelo-Vista-Controlador 23
1.7 Frameworks 24
1.8 Preparación del ambiente para desarrollar una aplicación web 25
1.8.1 Para instalar el software servidor Apache Tomcat 25
1.8.2 Para instalar NetBeans 25
1.9 Iniciando un proyecto Web con NetBeans 27

2. INTRODUCCIÓN AL HTML 29
2.1 Objetivos 29
2.2 Estructura de una página HTML 29
2.3 Desplegando una página HTML 30
2.4 Principales Tags de HTML 31
2.4.1 Tags para campos de texto 32
2.4.2 Tags para hacer una tabla 33
2.4.3 Tags para incluir una imagen 35
2.5 Formularios 36
2.5.1 Tags del formulario 37
2.5.2 El tag <input> dentro de un formulario 37
2.5.3 Los controles del formulario 38

2.6 Prácticas 46
2.6.1 Práctica 46
2.6.2 Práctica 46
2.6.3 Solución 47

3. COMUNICACIÓN ENTRE PÁGINAS WEB: PASO DE PARÁMETROS 51
3.1 Objetivo 51
3.2 Transición de una página a otra (enlace) 51
3.3 La petición de una JSP con el método GET 53
3.4 La petición de una JSP con el método POST 54
3.5 Scriptlets y expresiones en una JSP 54
3.6 Los tags en una JSP 58
3.7 Recepción de los valores de radio-button y checkbox 58
3.8 Recepción de valores en lista de opciones (combo box, list box) 61
3.9 Recepción de un área de texto 63
3.10 Práctica 65
3.10.1 Solución 66

4. COMUNICACIÓN ENTRE JSPS Y CLASES JAVA 69
4.1 Objetivos 69
4.2 Conceptos básicos 69
4.2.1 Los hilos (threads) en una JSP 69
4.2.2 Encapsulamiento 72
4.2.3 La ruta real para acceder a los archivos 73
4.3 Procesamiento de los datos de una JSP con una clase normal Java 74
4.3.1 Haciendo cálculos con una clase Java 74
4.3.2 Escribir los datos en un archivo con una clase Java 77
4.4 Prácticas 81
4.4.1 Práctica 1 81
4.4.2 Práctica 2 81
4.5 Solución a las prácticas 83
4.5.1 Solución de la práctica 1 83
4.5.2 Solución de la práctica 2 85

5. EL MODELO DE TRES CAPAS CON JSP, SERVLETS, Y CLASES JAVA 89
5.1 Objetivos 89
5.2 Los servlets y sus principales métodos 89
5.3 Paso de información de una JSP a un servlet 91
5.4 Paso de información de un servlet a una JSP 94

5.4.1 Transferencia de control con el objeto request 94
5.4.2 Transferencia de control con el objeto response 97
5.5 Comunicación entre servlets y clases Java 97
5.6 Práctica 105
5.6.1 Planteamiento 105
5.6.2 Solución a la práctica 106

6. ACCESO A BASE DE DATOS 115
6.1 Objetivo 115
6.2 Breve repaso de bases de datos 115
6.2.1 Creación de una base de datos: CREATE DATABASE 116
6.2.2 Comandos para crear y modificar tablas 117
6.2.3 Comando par consultar datos en una tabla: SELECT 119
6.2.4 Comandos para modificar datos en una tabla 120
6.3 Conexión con la base de datos 121
6.3.1 Ambiente 121
6.3.2 Código del conector 123
6.3.3 Escribiendo datos en una tabla 124
6.3.4 Lectura de un renglón de la tabla (consulta) 129
6.3.5 Lectura de un conjunto de renglones de la tabla 133
6.4 Errores más comunes 136
6.5 Práctica 137
6.5.1 Planteamiento 137
6.5.2 Solución 140

7. ALGUNOS ASPECTOS ADICIONALES 155
7.1 Objetivos 155
7.2 Introducción 155
7.3 Seguimiento de una sesión 155
7.4 Validación de datos de entrada 158
7.5 Introducción a los JavaBeans 160
7.6 Secuencias de escape 164
7.7 Cómo pasar una base de datos de una computadora a otra 164
7.7.1 Exportar la base de datos 164
7.7.2 Para importar la base de datos 165

8. LOS PRINCIPIOS DE JAVASERVER FACES 167
8.1 Objetivos 167
8.2 Introducción a JavaServer Faces 167
8.2.1 Tag Libraries 168
8.2.2 JavaBeans Administrados (managed Beans) 168

8.2.3 El Model-View-Controller (MVC) con JavaServer Faces 169
8.2.4 Las anotaciones Java y la tecnología XML 170
8.2.5 XHTML 171
8.3 Características importantes de los JavaBeans Administrados 172
8.3.1 Anotaciones para establecer el ámbito de los Beans 172
8.3.2 Declarando un Managed Bean con anotaciones 172
8.3.3 Los tres objetos JavaBean en la aplicación web 173
8.4 Comunicación de las vistas con la lógica de la aplicación 174
8.4.1 El lenguaje EL (Expression Languaje) 174
8.4.2 Las etiquetas de JSF 177
8.4.3 Ligado de datos de la página web con el managed Bean 178
8.5 La navegación en JSF 181
8.5.1 La navegación estática y dinámica 181
8.5.2 Flujo desde una página web hacia un Bean 181
8.5.3 Flujo desde una clase Java hacia una página web 181
8.6 Elementos básicos de la Interfaz de Usuario en JSF 187
8.6.1 Forma, botón de comando e imagen 187
8.6.2 Enlaces, campos de captura y salida de texto 187
8.6.3 Objetos para hacer una selección 188

9. PRIMER PROYECTO JSF EN NETBEANS 191
9.1 Objetivos 191
9.2 Creación de un proyecto JSF en NetBeans 191
9.2.1 Organización del proyecto implementando la arquitectura MVC 192
9.2.2 Desplegando una página de inicio 194
9.3 Una aplicación que contiene un Managed Bean 195
9.3.1 La página principal (index) 196
9.3.2 La página que despliega la sonrisa 197
9.3.3 Desplegando elementos de interfaz de usuario 197
9.3.4 Lo que sucede cuando un managed Bean no contiene todos sus setters y/o getters quevse usan a lavista 198
9.3.5 La página muestraElementosInterfaz.xhtml 199
9.3.6 El managed Bean Formulario.java 201
9.3.7 Ejecución en vivo del proyecto de ejemplo 202

10. CAPTURA, PROCESAMIENTO Y MUESTRA DE INFORMACIÓN EN JSF 203
10.1 Objetivos 203
10.2 Introducción 203
10.3 Captura de los datos del usuario 203
10.3.1 Estructura del proyecto 204

10.3.2 Managed Bean ligado al facelet de captura 205
10.3.3 Facelet que captura los datos 206
10.4 Procesamiento de los datos 207
10.4.1 Inyección de un Managed Bean dentro de otro Managed Bean 207
10.4.2 El bean que se inyecta debe tener su setter y getter 209
10.5 Desplegar resultados: usando los datos de los *Managed Beans* en los *facelets* 210
10.5.1 La elección del alcance de los Managed Beans 211
10.5.2 Compatibilidad en el alcance de los Managed Beans: Disponibilidad del contenido 211
10.5.3 Usando los datos de un solo managedBean en resultado.xhtml 212
10.5.4. Ejecución en vivo del proyecto de ejemplo 213
10.6 Práctica 214

11. PROYECTO JSF CON ACCESO A BASE DE DATOS 215
11.1 Objetivos 215
11.2 Proyecto "CRUD" con JSF 215
11.3 Creación de la base de datos 215
11.4 Creación del Proyecto 216
11.5 Conexión con la base de datos 217
11.6 Leer de la base de datos 219
11.6.1 Desplegando los registros leidos en la base de datos 220
11.6.2 El controllerManagedBean pide al model que lea la lista de UEA en la base de datos 222
11.7 Crear un elemento en la base de datos 223
11.7.1 La página que captura los datos de la nueva UEA 223
11.7.2 El método pedirDatosUEA_aAgregar() del controller 225
11.7.3 El método para guardar la UEA en el Controller 225
11.7.4 Desplegando la lista actualizada de UEA 226
11.7.5 El método para guardar la UEA en el model 226
11.8 Borrar un elemento en la base de datos 227
11.8.1 La página que captura los datos de la UEA a eliminar 227
11.8.2 El método pedirDatosUEA_aBorrar()del controller 229
11.8.3 El método para borrar la UEA en el Controller 229
11.8.4 Desplegando la lista actualizada de UEA 230
11.8.5 El método para borrar la UEA en el model 230
11.9 Modificar un elemento en la base de datos 231
11.9.1 Validar la existencia de la UEA a modificar 231
11.9.2 El método en el model para localizar la UEA 234
11.9.3 La página que captura los datos de la UEA a modificar 234
11.9.4 El método para modificar la UEA en el Controller 235

11.9.5 Desplegando la lista actualizada de UEA 235
11.9.6 El método para actualizar la UEA en el model 236
11.10 Práctica 237

12. VALIDADORES Y EXISTENCIA DE LIBRERÍAS ADICIONALES 241
12.1 Objetivos 241
12.2 Los validadores 241
12.2.1 Campo obligatorio 241
12.2.2 Validador de longitud 242
12.2.3 Validador de rango 243
12.2.4 Métodos validadores 244
12.3 Librerías adicionales a JSF para mejorar la presentación de las páginas web 245

BIBLIOGRAFÍA 246

GLOSARIO DE TÉRMINOS 247

Introducción

Alcances y pertinencia de la obra

Construir una primera aplicación web con acceso a base de datos, requiere de un gran esfuerzo por parte del estudiante, ya que debe poner en práctica varios conocimientos, y también adquirir otros más. Tomando en cuenta que las aplicaciones web se pueden construir con diversos lenguajes y tecnologías, es difícil para un principiante elegir el libro o la secuencia de libros que debe usar para iniciar su aprendizaje. En este material hemos seleccionado el lenguaje Java, y la justificación de esta elección se expone en una de las siguientes secciones. Elegimos también las tecnologías JSP-Servlets y JavaServer Faces, que, en nuestra opinión, facilitan el proceso de enseñanza aprendizaje. Además, JavaServer Faces es uno de los frameworks más utilizados actualmente en la industria del software. La primera parte de este curso abarca la implementación del patrón de diseño Modelo-Vista-Controlador con JSPs, Servlets y clases Java; y en la segunda parte se estudia la elaboración de aplicaciones Web usando el Framework JavaServer Faces.

Inspirado en el modelo educativo de la UAM Cuajimalpa, este libro de texto se apoya en un innovador sistema de autoaprendizaje llamado: "Sistema de Enseñanza de Aplicaciones web (SEAWeb)[1]", esta herramienta digital de apoyo a la docencia integra, en un sistema computacional, la teoría y la práctica. Su aportación principal es permitir al usuario trabajar con el ejecutable de las prácticas del curso. Los libros, en general, se valen de las capturas de pantalla (screenshots) para mostrar el funcionamiento de una aplicación, y algunos incluyen también un CD o un sitio web con los proyectos del curso. Los tutoriales en línea utilizan videos en los que el alumno observa cómo el profesor opera el sistema. SEAWeb es diferente ya que, además de utilizar el recurso de los *screenshots* en las explicaciones teóricas, propone prácticas que el estudiante debe resolver, pero que adicionalmente, se pueden ver funcionando en la red. A diferencia de los videos, es el estudiante y no el profesor el que interactúa con la aplicación. Así, antes de resolver la práctica, el alumno puede utilizar SEAWeb para experimentar con la práctica resuelta, es decir, introducir diferentes datos y usar los botones para observar cuál es la respuesta de la aplicación a cada uno de éstos.

Experimentar con la aplicación le permite darse cuenta de cómo es que algo funciona o puede fallar, lo cual ayuda a una mejor comprensión de los nuevos conceptos adquiridos.

---

[1]Gómez M. C., Custodio I., Cervantes J. Sistema de Enseñanza de Aplicaciones Web (SEAWeb). XXVIII Congreso Nacional y XIV Congreso Internacional de Informática y Computación del ANIEI (CNCIIC-ANIEI 2015), 28 al 30 de Octubre 2015 Puerto Vallarta Jalisco, pp. 332-239.

Incluimos dentro del código de este libro *cajas con aclaraciones*. Éstas contienen explicaciones adicionales en los lugares clave, las cuales ayudan a enfatizar los nuevos conocimientos que ya han sido expuestos en forma de texto.
Hemos observado que los tutoriales disponibles en Internet abarcan desde la introducción a la programación web, hasta los temas más avanzados. La buena voluntad de los autores de proporcionar la mayor cantidad de información posible en un solo documento, hace que se vea mermada la profundidad con la que se abordan los temas, de tal forma que quedan varios *cabos sueltos*. Hay temas sencillos que mientras para un experto son triviales, a un principiante le puede llevar varias horas encontrar en internet. Existen foros para programadores, "*Stack overflow*", por ejemplo, es uno de los más importantes. Sin embargo, no siempre es fácil encontrar las respuestas. Es necesario preguntar, en inglés, una y otra vez, de diferentes maneras, hasta encontrar por fín la información requerida. En este curso tratamos de minimizar la cantidad de cabos sueltos. En los ejemplos se explica paso a paso, como se construye una aplicación, sin dar por hecho que ya se sabe algo que no se ha dicho, como sucede en la gran mayoría de tutoriales disponibles en internet. Y como una ayuda adicional para el principiante, mencionamos e ilustramos algunos de los errores más comunes.

Las aplicaciones de apoyo a este libro se encuentran en los siguientes enlaces:

SEAWeb 1: http://148.206.168.124:8080/seaweb/
SEAWeb 2: http://148.206.168.124:8080/Sea_Web_2/

Nuestro objetivo es que este libro de texto y su complemento, el sistema de enseñanza SEAWeb, sean útiles no sólo para alumnos y profesores de la UAM-C, sino para todos aquellos que deseen aprender los principios de la programación web con Java.

Importancia de los conocimientos a adquirir y las habilidades a desarrollar

Las aplicaciones web son muy útiles en la vida moderna. Se utilizan constantemente en las computadoras y en los dispositivos móviles para llevar a cabo diversos tipos de negocios y para acceder a una gran variedad de servicios. Es por esto que la industria del software necesita desarrolladores web y ofrece empleos relativamente bien remunerados. Por lo tanto, hay un gran interés entre los que eligieron profesiones relacionadas con la informática, por aprender a elaborar y mantener aplicaciones web. Sin embargo, el aprendizaje de este tema es un proceso complejo, porque el estudiante debe integrar y poner en práctica los conocimientos adquiridos recientemente en su carrera y además adquirir otros más.

Los conocimientos y habilidades necesarios para construir una aplicación web son muy variados, por ejemplo, se requiere: programación orientada a objetos, lenguaje HTML, páginas JSP, Servlets, JavaBeans, JavaScript, EL, JSTL, bases de datos, nociones de sistemas concurrentes, etc. Algunos de estos temas son muy extensos, y hay cursos para estudiar cada uno de ellos por separado. Este libro integra los conceptos básicos necesarios para construir una aplicación web con lenguaje Java. Los temas se exponen a un nivel introductorio, pero de manera detallada, y se promueve la adquisición de las habilidades requeridas mediante las prácticas propuestas. El alumno puede operar la práctica antes de resolverla y entender mejor su funcionamiento mediante el sistema de enseñanza SEAWeb.

Importancia del apoyo que brinda este libro a las UEA de Proyecto Terminal

Hay un interés general de los alumnos de la Licenciatura de Ingeniería en Computación por aprender a desarrollar aplicaciones Web con Java, no solamente porque Java es uno de los principales lenguajes que aprenden en su carrera, sino también porque es el más popular y con una alta oferta de empleos. La Tabla 1 muestra los resultados de la búsqueda de empleos en **www.indeed.com.mx**, en 2015, 2016 y 2017. En ésta se observa claramente la popularidad de Java.

Clave de búsqueda	Número de empleos (abril 2015)	Número de empleos (diciembre 2016)	Número de empleos (febrero 2017)
Desarrollador Web Java	313	369	321
Desarrollador Web .NET	266	340	292
Desarrollador Web C#	170	254	221
Desarrollador Web PHP	165	261	243
Desarrollador Web Phyton	3	9	3
Desarrollador Ruby	0	23	27

Tabla 1. Empleos en México para desarrolladores web en diferentes lenguajes (www.indeed.com.mx).

No obstante, el plan de estudios de Ingeniería en Computación no contempla alguna UEA para aplicaciónes Web con Java. El contenido del programa de estudio de la UEA optativa de orientación: "Desarrollo de aplicaciones Web" está pensado para el lenguaje PHP.

Los proyectos terminales que tienen como objetivo el desarrollo de aplicaciones Web con Java, brindan a los alumnos la oportunidad de aprender, por su cuenta, esta importante tecnología. Sin embargo, la cantidad de información en librerías, bibliotecas e internet es abrumadoramente grande. Para tener una idea, si usamos como clave de búsqueda en *Google*: "learn Java web application development", se despliegan cerca de tres millones de resultados de material en inglés, y con "how to develop Java web application" 109 millones. Del material en español,

tenemos que con la clave "como desarrollar aplicaciones web con java" aparecen 780 mil resultados. Tal cantidad de información suele desorientar aún más al principiante, ya que no hay un criterio que permita elegir cuál es el material más apropiado para comenzar, ni la calidad del mismo. Además, muchos de estos materiales no son gratuitos. Por todo lo anterior, es difícil para un principiante elegir el libro, la secuencia de libros o sitios de internet que debe consultar para iniciar su aprendizaje. Este libro de texto junto con el Sistema SEAWeb, integran la teoría con la práctica para la creación de aplicaciones web con Servlets, JSP y JavaServer Faces, proporcionando una guía efectiva para el autoaprendizaje.

Agradecemos a los alumnos: Josué Alan Aguilar Ramírez, Fiordalizo Michel Lira Gómez (2016-I, 2016-O), Irvin Osvaldo Custodio Sánchez (2014-I, 2014-P, 2014-O), Iván Rosales Soriano y Javier Martínez Hernández (2014-P, 2014-O, 2015-I), quienes demostraron con su dedicación la efectividad del autoaprendizaje con SEAWeb.

Objetivos

Objetivo General:

Desarrollar en el alumno la habilidad de construir aplicaciones en Java, que funcionen en la red y tengan acceso a base de datos.

Objetivos Específicos:

1. Conocer los principios de la programación Web.
2. Implementar el Modelo-Vista-Controlador (MVC) con *JSPs*, *Servlets* y clases *Java*.
3. Conocer algunos de los aspectos adicionales básicos, que ayudan a mejorar y a optimizar una aplicación Web.
4. Construir aplicaciones Web con la tecnología *JavaServer Faces*.

Conocimientos previos:

Para poder comprender este curso, es necesario que el lector tenga conocimientos básicos de programación orientada a objetos, de Java y de bases de datos.

1. ¿Qué se necesita para hacer una aplicación que funciona en la red?

1.1 Objetivo

Ubicar los componentes esenciales de una aplicación Web.

1.2 ¿Qué es una aplicación Web?

El término Web proviene del inglés, y significa red o malla, este término ha sido adoptado para referirse al internet. Una aplicación Web es un conjunto de páginas que funcionan en internet, estas páginas son las que el usuario ve a través de un navegador de internet (Internet Explorer de Microsoft, Chrome, Mozilla Firefox, etc.) y están codificadas en un lenguaje especial. Existen varios tipos de páginas Web: HTML, JSPs, XML… En la primera parte de este curso trabajaremos con las JavaServer Pages (JSPs). Las páginas JSP se ejecutan en una máquina virtual de Java, el resultado de la ejecución es código HTML listo para correr en el navegador. Las JSP constituyen la interfaz de la aplicación con el usuario.

Las aplicaciones Web se almacenan en un servidor, el cual es una computadora que se encarga de que éstas sean accesibles a través de internet. Como se ilustra en la Figura I-1, una aplicación Web corre en un *servidor* bajo el control de un software especial, al cual se le llama también *servidor*. Para evitar confusiones es importante aclarar que el *software servidor* corre en una *computadora servidor*.

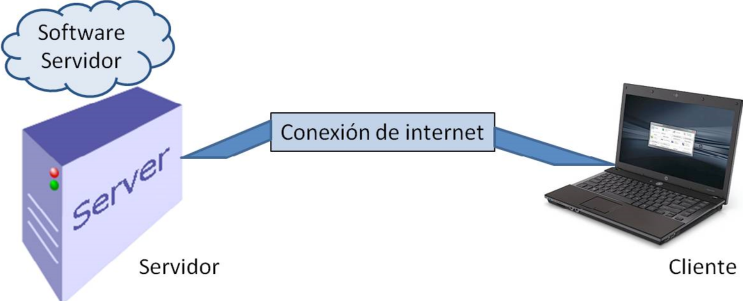


Figura I-1. Una aplicación web funciona con una computadora-servidor y una o varias computadoras-cliente conectadas a través de internet

Uno de los *software servidores* más ampliamente utilizados es el *Apache Tomcat*, y es el que usaremos en este curso, es de distribución libre. *GlassFish* es otro software servidor que también es muy utilizado.

Es muy común que las aplicaciones Web hagan uso de una base de datos ubicada en la computadora-servidor, los manejadores de bases de datos más populares son Oracle y MySQL. En este curso utilizaremos MySQL, porque es gratuito. El manejador de base de datos permite que varios clientes compartan la información, éste es uno de los aspectos más útiles de las aplicaciones web, ya que permite el comercio en línea (tiendas virtuales, reservaciones de hoteles, vuelos, etc.) y facilita la organización en las instituciones (solicitudes en línea, sistemas de inscripción, control de préstamos bibliotecarios, etc.), como se ilustra en la Figura I-2.

Figura I-2. Gracias al administrador de base de datos que corre en el servidor, las computadoras-cliente comparten una misma información.

Finalmente, al "cerebro" de la aplicación web, se le conoce como la *lógica del negocio* y ésta se le puede codificar de diversas formas (C#, PHP, .NET, JavaScript,...) en el presente curso utilizaremos *Java* para hacer *Servlets* y *clases Java*. Dos de los ambientes de desarrollo integrado más utilizados para el desarrollo de aplicaciones web son NetBeans y Eclipse, ya que son gratuitos. En el curso utilizaremos NetBeans.

1.3 Peticiones y respuestas en las páginas Web

El software servidor y el navegador del cliente se comunican por medio de un protocolo llamado HiperText Transfer Protocol (HTTP). El navegador hace la petición de una página Web al servidor enviándole un mensaje conocido como *petición HTTP (request)*, la cual incluye el nombre de un archivo *.html, y el servidor contesta a esta petición con un mensaje conocido como *respuesta HTTP (response)*.

En el caso de las ***páginas Web estáticas***, el servidor proporciona en la respuesta HTTP el documento *.html que el navegador solicitó. El usuario que visualiza una página Web estática, no puede hacer modificaciones en ésta. Cuando el usuario da clic en la liga a otra página, se envía otra petición HTTP pero ahora con el nombre del archivo de la otra página que se desea visualizar. Otra manera de pedir una página diferente es escribiendo directamente en el navegador su dirección Web.

En el caso de las ***páginas web dinámicas***, el servidor pasa la petición HTTP generada por el navegador a una *aplicación Web,* la cual procesa la información que contiene la petición. La respuesta que genera la aplicación se envía al servidor, quien contesta al navegador con una respuesta HTTP, como se ilustra en la Figura I-3. La respuesta que recibe el usuario no es siempre la misma, sino que depende de la información que éste proporciona, por eso se dice que la página es dinámica.

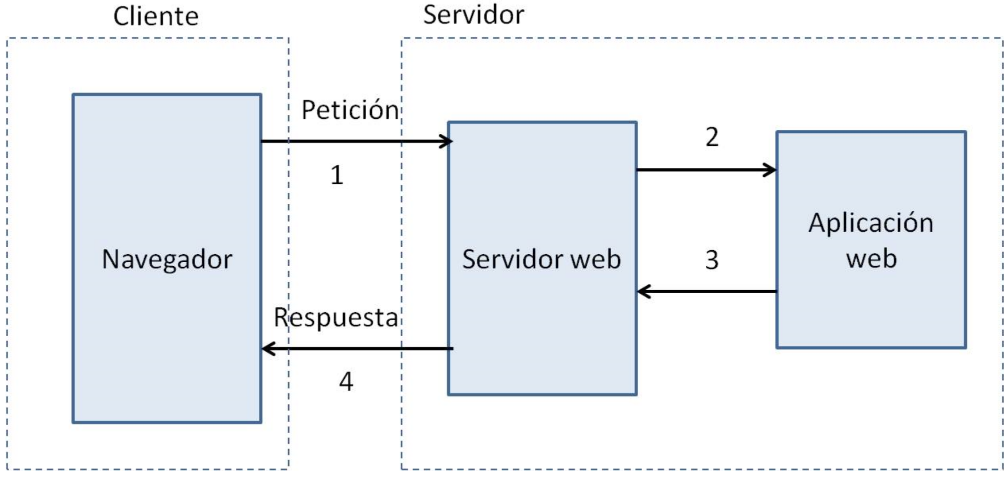


Figura I3. El servidor web turna el procesamiento de las páginas web dinámicas a una aplicación web

1.4 ¿Qué es una JSP?

Una página JSP (*JavaServer Page*) es una página HTML a la que se le incrusta código Java. En el capítulo II se da una introducción al código HTML para los lectores que no están familiarizados con éste. El código Java se incrusta entre los siguientes indicadores <% y %>. En el capítulo III trabajaremos con las JSP.

1.5 ¿Qué es un Servlet?

Un *servlet* es una clase Java (hija de la clase `HttpServlet`) y corre en el servidor. Su nombre se deriva de la palabra *applet*. Anteriormente se utilizaban los *applets*, que eran pequeños programas, escritos en Java, que corrían en el contexto del navegador del cliente, sin embargo, desde que Microsoft Explorer suspendió su mantenimiento, los *servlets* substituyeron a los *applets*, sólo que los *servlets* no tienen una interfaz gráfica. Un *servlet* da servicio a las peticiones de un navegador Web, es decir, recibe la petición, la procesa y devuelve la respuesta al navegador. Un *servlet* es una clase Java en la que se puede incrustar código HTML. Como los *servlets* están escritos en Java, son tan portables como cualquier aplicación Java, es decir, pueden funcionar sin necesidad de cambios en diferentes servidores.

A manera de ejemplo, presentamos el código de un *servlet* muy sencillo, el cual genera como respuesta una página HTML que despliega un breve mensaje.

El objeto `request` se usa para leer los datos que están en los formularios que envía el navegador. El objeto `response` se usa para especificar códigos y contenidos de la respuesta.

```
public class EjemploServlet extends HttpServlet {

    // Escribe una página Web sencilla

    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {

    PrintWriter out;
    String title = "Ejemplo de un servlet";


    response.setContentType("text/html"); // Regresa texto y html
```

Recibe los parámetros enviados por el navegador

Tipo de información que regresa el *servlet*

```
                                        Regresa el objeto en el que se escribe
                                        la respuesta para el cliente

    out = response.getWriter();

    // Con este código se escriben los datos de la respuesta
    out.println("<HTML><HEAD><TITLE>");
    out.println(title);
    out.println("</TITLE></HEAD><BODY>");
    out.println("<H1>" + title + "</H1>");
    out.println("<P>Este es un servlet muy sencillo.");
    out.println("</BODY></HTML>");
    out.close();
    }
}
```

Un servlet puede generar su resultado consultando a una base de datos, invocando a otra aplicación o computando directamente la respuesta. También puede dar formato al resultado generando una página HTML, y enviar al cliente un código ejecutable.

1.6 La arquitectura Modelo-Vista-Controlador

La arquitectura Modelo-Vista-Controlador (MVC), es un patrón que organiza la aplicación en tres partes independientes:

- ***La vista***.- Son los módulos SW involucrados en la interfaz con el usuario, por ejemplo, las páginas de internet que se despliegan en la computadora del usuario.
- ***El controlador***.- Es el software que procesa las peticiones del usuario. Decide qué modulo tendrá el control para que ejecute la siguiente tarea.
- ***El modelo***.- Contiene el núcleo de la funcionalidad, es decir, ejecuta la "lógica del negocio". Se le llama *lógica del negocio* a la forma en la que se procesa la información para generar los resultados esperados. El modelo se conecta a la base de datos para guardar y recuperar información.
-

El patrón MVC convierte la aplicación en un sistema modular, lo que facilita su desarrollo y mantenimiento. En la Figura I-4 se ilustran las tres partes del modelo MCV.

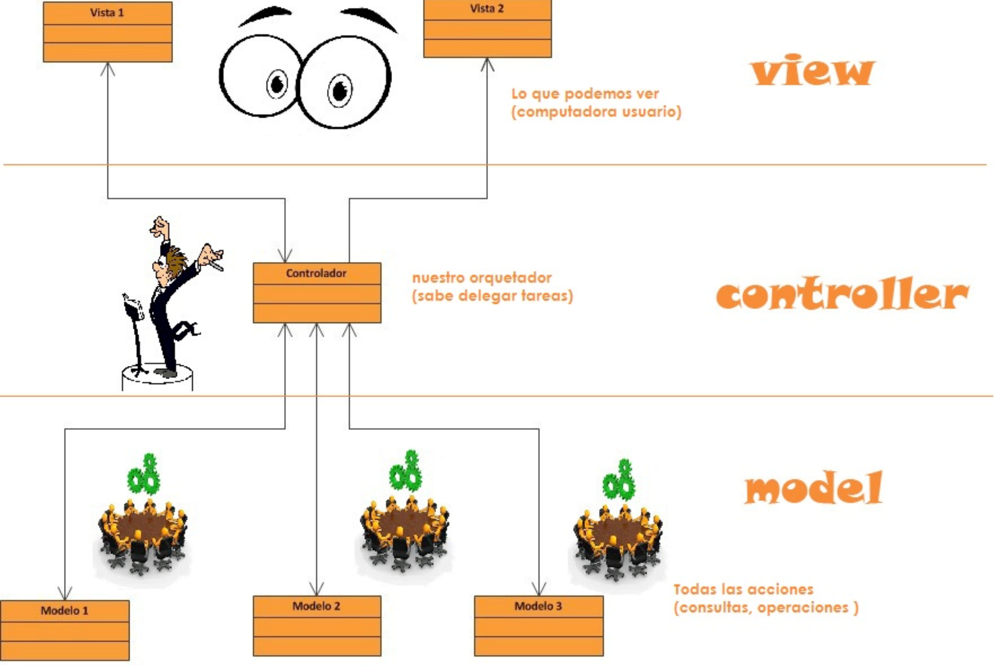


Figura I-4. Arquitectectura Modelo-Vista-Controlador (MVC)

En las aplicaciones Web más sencillas de este curso, sólo separamos la vista (con páginas JSP) y el procesamiento de los datos (con *servlets*). Sin embargo, para aplicaciones Web un poco más avanzadas utilizamos el patrón MVC. La vista la implementaremos con páginas JSP, el controlador con *servlets* y el modelo con clases Java(ver capítulo V).

Al diseñar y codificar los módulos MVC, es importante que se haga una buena división de las tareas. Por ejemplo, las páginas JSP no deben incluir tareas de procesamiento de datos y los *servlets* no deben contener código para la presentación de las páginas. La división de tareas entre el controlador y el modelo es la más difícil. Mantener una total independencia entre los módulos es a veces imposible, sin embargo es el objetivo ideal.

1.7 Frameworks

Un *framework* se traduce al español como *marco de trabajo*, y es un esqueleto para el desarrollo de una aplicación. Los *frameworks* definen la estructura de la aplicación, es decir, la manera en la que se organizan los archivos e, inclusive, los nombres de algunos de los archivos y las convenciones de programación. Como el framework proporciona el esqueleto que hay que

rellenar, el programador ya no se preocupa por diseñar la estructura global de la aplicación. Como la información está estandarizada es más sencillo el trabajo colaborativo, y el mantenimiento de las aplicaciones, porque la estructura de la aplicación es bien conocida.

Podemos ver a un *framework* como una estructura de software que tiene componentes personalizables e intercambiables y constituye una aplicación genérica incompleta y configurable, a la que se le añaden las últimas piezas para construir una aplicación concreta.

La mayoría de los *frameworks* para desarrollo Web implementan el patrón MVC con algunas variantes. Entre los *frameworks* más conocidos están:

- JavaServer Faces
- Struts
- Spring Web MVC

En la segunda parte de este libro aprenderemos a hacer aplicaciones Web con *JavaServer Faces*.

1.8 Preparación del ambiente para desarrollar una aplicación wWeb

1.8.1 Para instalar el software servidor *Apache Tomcat*

Apache Tomcat es un software que actúa como *contenedor de servlets*, es decir, recibe las peticiones de las páginas Web y redirecciona estas peticiones a un objeto de clase *servlet*.

Apache Tomcat puede descargarse de http://tomcat.apache.org/

Descargar el zip (32 o 64bit) y descomprimirlo para iniciar la instalación.

1.8.2 Para instalar *NetBeans*

NetBeans es un entorno de desarrollo en el que los programadores puedan escribir, compilar, depurar y ejecutar programas. Éste se puede descargar de:

https://netBeans.org/downloads/index.html

Para poder instalar NetBeans es necesario tener previamente instalado el JDK (*JavaDevelopment Kit*), ya que éste es la plataforma en la que se ejecuta *NetBeans*.

Cuando se ejecuta el instalador de *NetBeans*, es recomendable seleccionar los dos servidores: **Apache Tomcat** y **Glassfish**, que son con los que trabaja *NetBeans*. Esto se hace con el boton "*Customize*", como se indica en la Figura I-5.

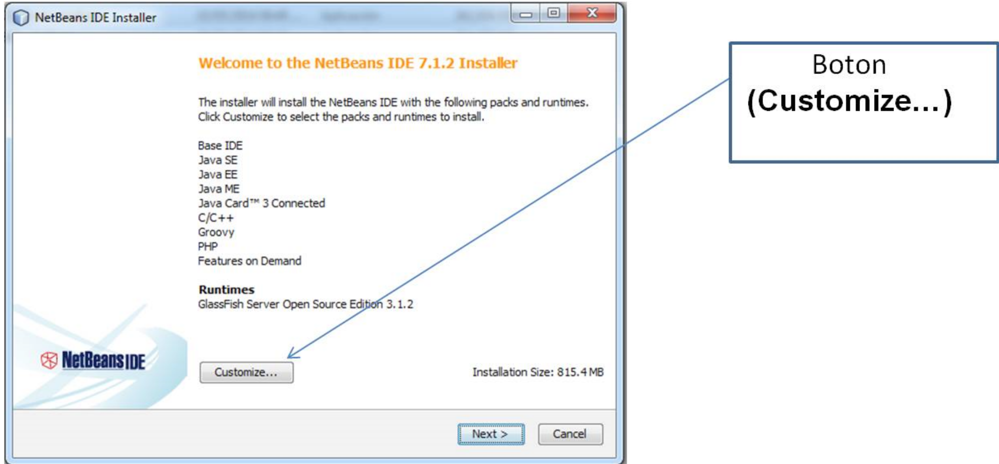


Figura I-5 El botón para personalizar la instalación

La selección de los dos servidores se ilustra en la Figura I-6.

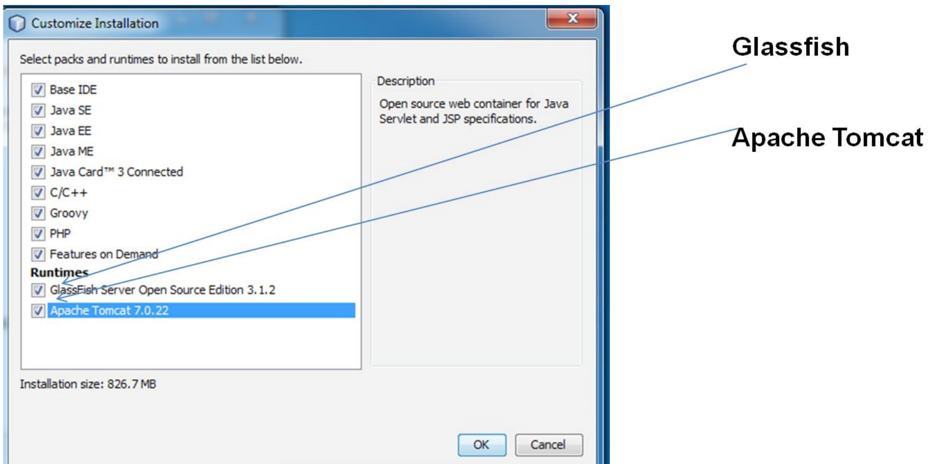


Figura I-6. Selección de los dos servidores

En el capítulo III se explica cómo crear una aplicación Web con *NetBeans*.

1.9 Iniciando un proyecto Web con NetBeans

Para iniciar una aplicación Web se selecciona File →New Project→JavaWeb, como se ilustra en la Figura I-7.

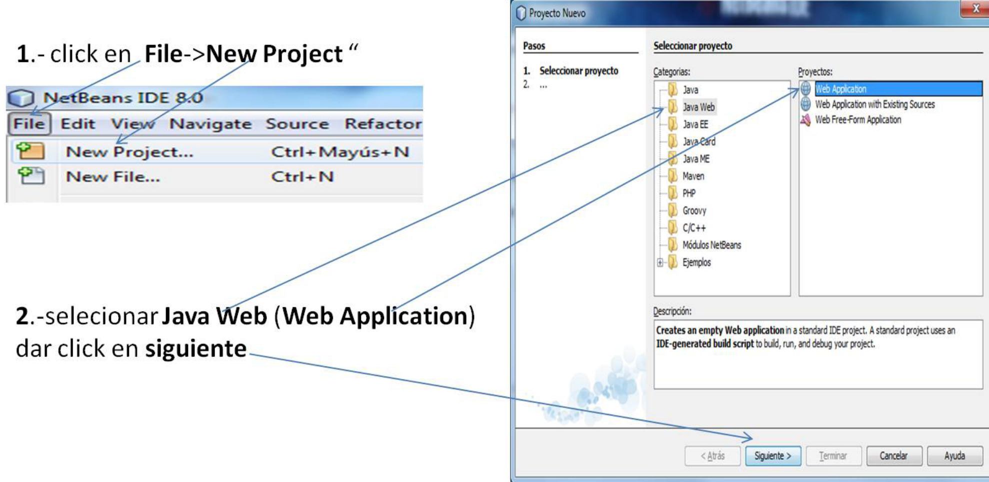


Figura I-7. Inicio de una aplicación web en NetBeans

Es muy importante seleccionar correctamente el servidor que tenemos instalado, en este caso Tomcat, como se ilustra en la Figura I-8.

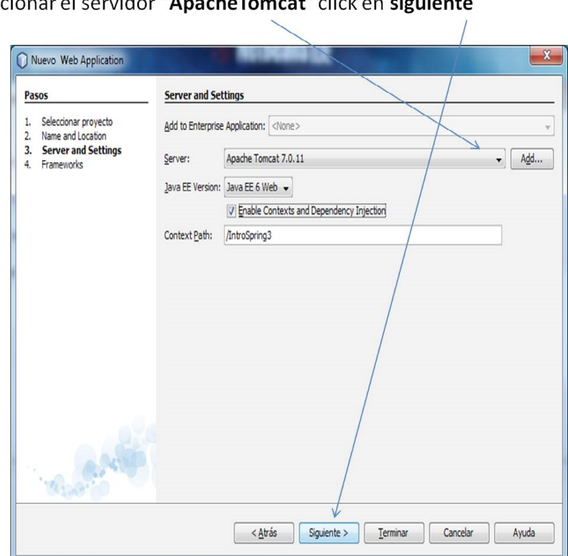


Figura I-8. Selección del servidor Tomcat

Para una aplicación Web sencilla no es necesario seleccionar un *framework*. Cuando se selecciona el botón “Terminar”, *NetBeans* crea automáticamente la estructura de un proyecto Web. Esta estructura se muestra en la Figura I-9.

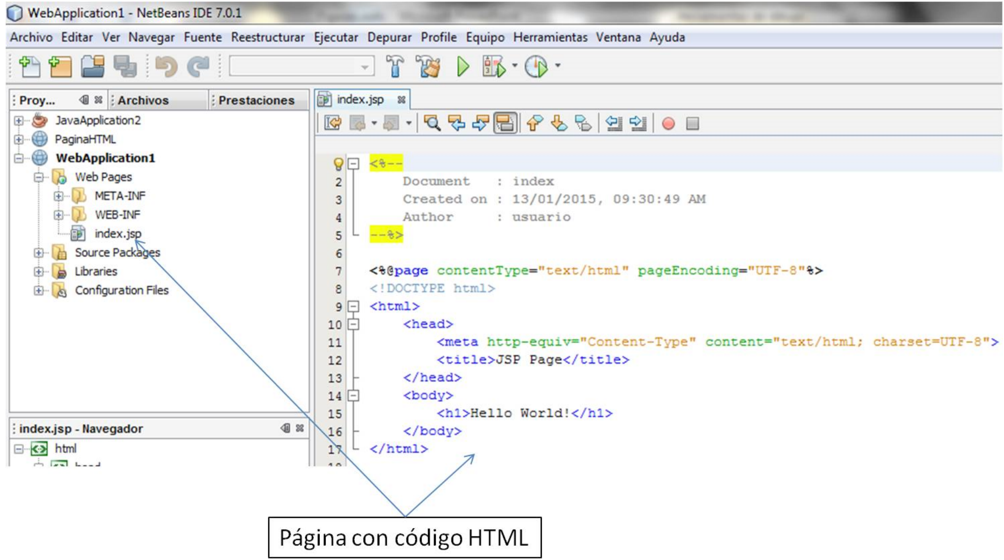


Figura I-9. Estructura de un proyecto Web en NetBeans

El archivo *index.jsp* contiene código HTML. Por *default* la página *index.jsp* sólo contiene el saludo “Hello World!”. En el siguiente capítulo aprenderemos a trabajar con código HTML y a visualizar las páginas en el navegador.

2. Introducción al HTML

2.1 Objetivos

- Aprender a estructurar una página HTML.
- Saber utilizar los comandos principales para trabajar con ésta.

2.2 Estructura de una página HTML

HTML es el acrónimo de las siglas del inglés: **H**yper**T**ext **M**arkup **L**anguage (lenguaje de marcas de hipertexto). Las páginas de hipertexto se codifican con HTML. Los navegadores de internet están preparados para interpretar los comandos de HTML y desplegar la información en forma de una página de internet con texto e imágenes.

El hipertexto de las páginas HTML está definido por *tags*, que son los comandos. Los tags comienzan con el carácter < y terminan con el carácter >. La información del tag va dentro de estos dos caracteres. Los navegadores arman y dan forma a las páginas de internet interpretando estos comandos. A continuación, presentamos la estructura general de una página HTML. Observa que hay tags para marcar el comienzo y otros para marcar el final. Por ejemplo, el comienzo de la página comienza con el tag <html> y para señalar el final de la página se usa el tag </html>. La diagonal es importante para señalar los finales.

HTML no es sensible a las mayúsculas y minúsculas, sin embargo, por claridad se acostumbran las minúsculas. Para hacer más legible la página HTML, se emplean espacios y líneas en blanco.

```
<html>
   <head>
        <title> Título de la página </title>
   </head>

   <body>
        <h1> Página de ejemplo de HTML </h>
        <h2> Este es un subtítulo </h2>
        <p> El texto en este tag es un párrafo,
              por ejemplo pueden ser instrucciones </p>
  </body>
</html>
```

El título no se despliega en la página de internet

Este texto aparece como encabezado de la página

2.3 Desplegando una página HTML

Las páginas HTML se pueden escribir en cualquier editor de textos (NotePad, Word, etcétera). También hay poderosas herramientas, que sirven para trabajar con páginas HTML y darles un acabado profesional. Para diseñar una aplicación Web, se acostumbra utilizar un ambiente de desarrollo integrado (IDE: Integrated Development Environment).

En los ejemplos de esta sección utilizaremos *NetBeans*. Recomendamos *Sublime Text* (http://www.sublimetext.com/) para trabajar fácilmente con la edición de páginas Web. Otra opción es *Atom*, un editor gratuito y de código abierto.

Recordar que para crear una aplicación Web con *NetBeans*, hay que crear un nuevo proyecto seleccionando la categoría *JavaWeb* y el proyecto *Web Aplication*. Una vez que se da el nombre y localización del proyecto, hay que seleccionar el servidor, en nuestro caso *Apache Tomcat*. Finalmente *NetBeans* nos da la opción de seleccionar algún *framework* (marco de trabajo). Como estamos iniciando con el desarrollo Web, no seleccionaremos un *framework* para nuestra aplicación.

NetBeans genera automáticamente una página JSP con código HTML que contiene únicamente el título “Hello World!”. Si sustituimos este código con el de la sección anterior tenemos el código de la Figura II-1.

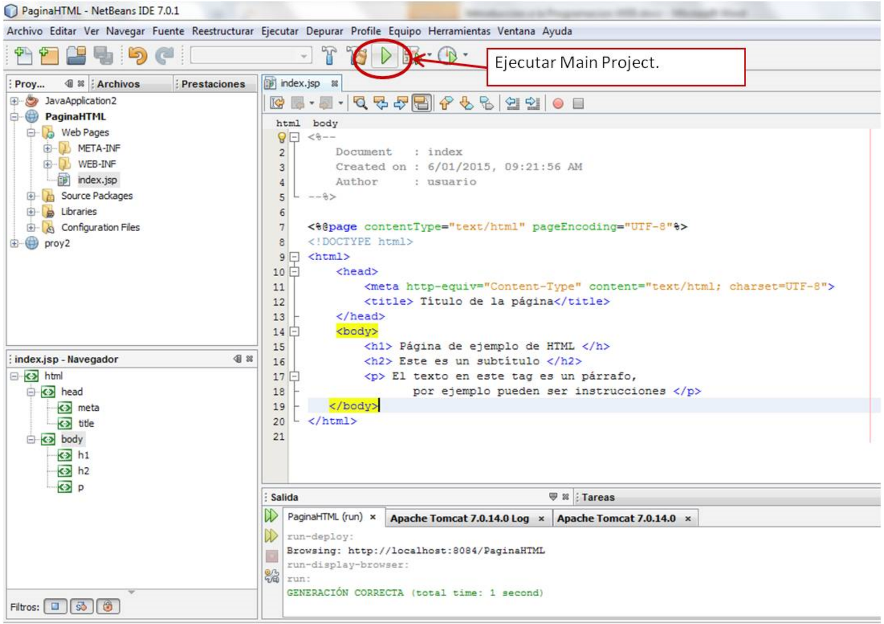


Figura II-1. Página HTML en NetBeans

Para hacer que la página HTML se despliegue en el navegador es necesario elegir la opción "Ejecutar Main Project", en la barra del menú o con el ícono señalado en la Figura II-1. En la Figura II-2 se muestra la página desplegada por el navegador.

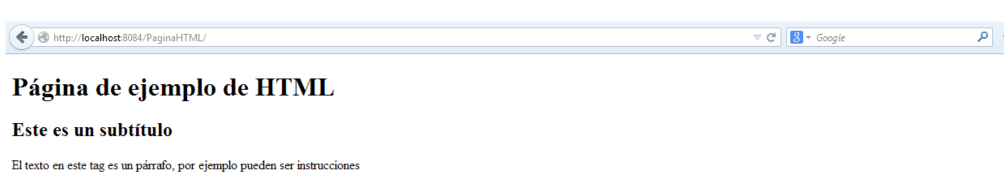


Figura II-2. Página HTML interpretada y desplegada por el navegador

Otra forma de correr un proyecto es seleccionarlo y, con el botón derecho del mouse y elegir el comando *run* (Ejecutar).

2.4 Principales *Tags* de HTML

Recordar que los *tags* son los comandos de HTML. En la Tabla II1se muestran los *tags* básicos, es decir, los que sirven para construir una página HTML y dar los formatos más comunes.

Tabla II-1. Tags básicos de HTML	
Tag	Descripción
<!doctype>	Identifica el tipo de documento HTML
<html> </html>	Marca el inicio y el final del documento HTML.
<head> </head>	Marca el inicio y final del encabezado del documento HTML.
<title> </title>	El texto que se incluya es el que aparecerá en la barra de título del navegador.
<head> </head>	Marca el inicio y final del encabezado del documento HTML.
<body> </body>	Marca el inicio y el final del cuerpo del documento HTML.
	El texto tendrá el formato de "encabezado 1"
<h2 > </h2>	El texto tendrá el formato de "encabezado 2"
	El texto tendrá el formato de "párrafo normal"
 	Se inserta un cambio de línea
 	Marca el texto en **negrita**
<i> </i>	Marca el texto en *itálica*
<u> </u>	Subraya el texto
<!-- comentario --	El navegador ignora lo que esté dentro de este tag

Tabla II-1. Tags básicos de HTML

2.4.1 *Tag* para campos de texto

Los campos de texto se utilizan para capturar la información del usuario. Estos campos se definen con el *tag* <input> el cual tiene varios atributos, por lo pronto estudiaremos el atributo *type*. Cuando indicamos type=”text” los caracteres que introduce el usuario se muestran dentro del campo. Si indicamos type=”password”, en lugar de los caracteres se muestran puntos. Además, existe type=”hidden”, que permite guardar un campo que no se despliega en la página.

El siguiente código HTML despliega un campo de texto normal, uno de tipo password y un campo de texto oculto.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title> Campos de texto</title>
    </head>
    <body>
        <p>
            Cuenta: <input type="text" name="cuenta"> <br>
            Contraseña: <input type="password" name="password">
            <br>
            Campo Oculto: <input type="hidden" name="codigo"> <br>
        </p>

    </body>
</html>
```

La página de este código se muestra en la Figura II-3.

Cuenta:
Contraseña:
Campo Oculto:

Figura II-3. Campos de texto vacío

El atributo *value* sirve para indicar el valor por *default* del campo, en el siguiente código agregamos este atributo y los campos ya no se despliegan vacíos, sino con su valor de *dafault*, como se aprecia en la Figura II-4.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
<meta http-equiv="Content-Type" content="text/html;charset=UTF-8">
        <title> Campos de texto</title>
   </head>
   <body>
      <p>
         Cuenta: <input type="text" name="cuenta"
                                    value="PedroSanchez"> <br>
         Contraseña: <input type="password" name="password"
                                            value="Pedro123"> <br>
         Campo oculto: <input type="hidden" name="codigo"
                                            value="usuario52">
      </p>
    </body>
</html>
```

Figura II-4. Campos de texto con valores por default

El *tag* <input> no solamente se usa para campos de texto, en la sección II.5 se estudian los otros usos de <input>.

2.4.2 *Tags* para hacer una tabla

La Tabla II-2 contiene los principales *tags* de HTML para construir una tabla. Cada *tag* tiene uno o varios atributos. En la última columna se indica cuáles son los valores que puede tomar cada uno de los atributos.

Tabla II-2. Tags de una tabla			
Tag	Descripción	Atributo	Valores posibles
<table> </table>	Marca el inicio y el final de una tabla.	• border • cellspacing • width • height	• Sin borde: 0 con borde 1 o mayor. • # de pixeles entre celdas • Ancho de la tabla • Alto de la tabla (ancho y alto se expresan en pixeles: 200, 300...; o en porcentaje del espacio en el display: 40%, 60%...)
<tr> </tr>	Marca el inicio y el final de renglón en la tabla.	• valign	Top, Bottom, Middle
<td> </td>	Marca el inicio y el final de cada celda de la tabla en un mismo renglón.	• align • colspan • rowspan • height • width • valign	• Left, Right, Center • Número de columnas que la celda se extenderá • Número de renglones que la celda se extenderá • peso de la celda en pixeles • ancho de la celda en pixeles • Top, Bottom, Middle

Tabla II-2. Tags de una tabla

El siguiente código HTML despliega una tabla con tres columnas y cuatro renglones, el primer renglón es el encabezado de la tabla.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title> Tablas </title>
  </head>
  <body>
     <h1> Ejemplo de Tabla </h1>
     <table cellspacing="4" cellpadding="3" border="1">
         <tr>
             <td align="center"> Nombre</td>
             <td align="center"> Apellidos</td>
             <td align="center"> Calificacion</td>
         </tr>

         <tr>
             <td > Nombre 1</td>
             <td > Apellidos 1</td>
             <td > Calific. 1</td>
         </tr>
```

```
            <tr>
                <td > Nombre 2</td>
                <td > Apellidos 2</td>
                <td > Calific. 2</td>
            </tr>

            <tr>
                <td > Nombre 3</td>
                <td > Apellidos 3</td>
                <td > Calific. 3</td>
            </tr>
        </table>

    </body>
</html>
```

En la Figura II-5 se muestra la tabla correspondiente al código anterior.

Ejemplo de Tabla

Nombre	Apellidos	Calificacion
Nombre 1	Apellidos 1	Calific. 1
Nombre 2	Apellidos 2	Calific. 2
Nombre 3	Apellidos 3	Calific. 3

Figura II-5. Tabla con 4 renglones y tres columnas

2.4.3 Tag para incluir una imagen

La imagen es uno de los elementos que se pueden incluir en una página HTML, en la Tabla II-3 se describen los atributos del *tag* <img>.

Tabla II-3. El tag <*img*>	
Atributo	Descripción
src	Especifica la URL donde se encuentra el archivo GIF o JPG
height	Altura de la imagen en pixeles
width	Ancho de la imagen en pixeles
alt	Texto que se muestra cuando no se puede desplegar la imagen
border	Ancho del borde (en pixeles), 0 significa "sin borde"
hspace	Espacio horizontal (en pixeles), que se añade a la izquierda y derecha de la imagen
vspace	Espacio vertical (en pixeles), que se añade a la arriba y abajo de la imagen.
align	Alineación de la imagen en la página: Left, Right, Top Bottom, Middle

Tabla II-3. El Tag <img>

En la Figura II-6 se muestra una página con dos imágenes, una alineada a la izquierda y la otra a la derecha.

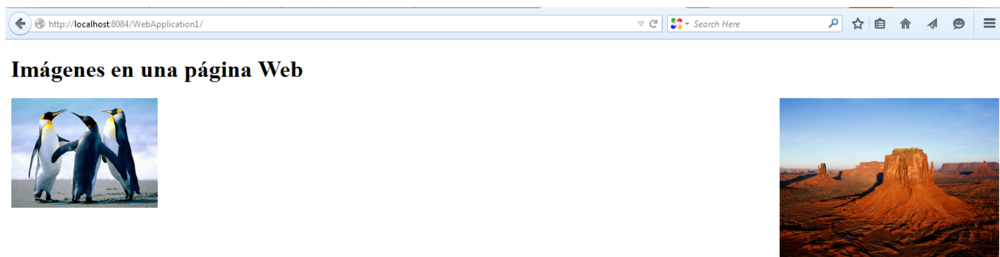


Figura II-6. Página Web con imágenes

Para que el navegador tenga acceso a las imágenes es necesario que el archivo con la imagen se encuentre dentro de la carpeta del proyecto. El código de la página de la Figura II-6 es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title> Imágenes </title>
    </head>
    <body>
        <h1> Imágenes en una página Web </h1>

        <img src="Pingu.jpg" width="200" heigth="200"
             align="left" alt="No se pudo mostrar la imagen">

        <img src="Desert.jpg" width="300" heigth="300"
             align="right" alt="No se pudo mostrar la imagen">

    </body>
</html>
```

2.5 Formularios

Los *formularios Web* se utilizan para cifrar los datos del usuario, así viajan del cliente al servidor de manera más segura, (el cifrado de datos es una forma de protegerlos de un acceso no deseado). Un formulario debe contener por lo menos un botón, el cual sirve de *control*, ya que cuando el usuario pulsa el botón se envían los datos del formulario a la página indicada.

2.5.1 Tags del formulario

El comando para construir un formulario se muestra en la Tabla II-4.

Tag	Descripción	Atributo	Descripción
<form> </form>	Define el inicio y el final de una formulario.	• action • method	• Especifica la URL del Servlet o de la JSP a la que se invoca cuando el usuario pulsa el botón "submit" • GET (default) o POST (ver sección III.1)

Tabla II-4. Tag del formulario

2.5.2 El tag <input> dentro de un formulario

Dentro del *tag* del formulario el *tag* <input> se usa para definir los campos en los que se captura la información y también para definir los *controles* del formulario. En la Tabla II-5 se incluyen todos los atributos de <input>.

Tag	Atributo	Descripción
<input >	• type	- Define el tipo del campo de entrada. Text.-Texto normal Password.- se despliegan puntos en lugar de los caracteres. Hidden.- para guardar un campo que no se despliega en la página. - Define el tipo de control (un botón, texto, etc.)
	• name	Nombre del campo de captura
	• value	Cuando se trata de un control, define su valor por default.

Tabla II-5. Atributos del tag input

2.5.3 Los controles del formulario

Los controles del formulario sirven para reunir la información proporcionada por el usuario y para enviarla al destino indicado en el atributo *action*. Existen varios tipos de controles, en esta sección estudiaremos: botones, check-box, combo-box, list-box y áreas de texto.

2.5.3.1 Botones

Para que funcione un formulario, es obligatorio que exista un botón de envío, ya que sin él no se enviarían los datos capturados.

Existen tres tipos de botones:

1. Botón de envío (submit)
2. Botón de reinicialización (reset).- reinicializa todos los controles a sus valores iniciales (definidos en el atributo *value*).
3. Botón pulsador (button).- Llama a un método escrito en JavaScript para que se ejecute.

El siguiente código HTML despliega un formulario con cuatro campos de texto y un botón para enviar los datos. Nótese que ninguno de los campos tiene definido su valor de *default*, es decir, no se usa el atributo *value*.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title> Texto y Botones</title>
    </head>

    <body>
      <h1> Ejemplo de Campos de Texto y Botones</h1>

      <p> Despues de introducir sus datos oprima el boton "ok"
      </p>
      <form action="paginaDestino.jsp" >
            <p>
                Nombre: <input type="text" name="Nombre"> <br>
                Apellido1: <input type="text" name="apell1"> <br>
                Apellido2: <input type="text" name="apell2"> <br>
                Telefono: <input type="text" name="tel">
            </p>
            <input type="submit" value="Ok">

        </form>


    </body>
</html>
```

La página de este código se ve en la Figura II-7.

http://localhost:8084/WebApplication1/

Ejemplo de Campos de Texto y Botones

Despues de introducir sus datos oprima el boton "ok"

Nombre:
Apellido1:
Apellido2:
Telefono:

ok

Figura II-7. Campos de Texto dentro de un formulario

En el siguiente ejemplo un campo de texto está afuera del formulario, y hay un botón para borrar que está dentro del formulario. Nótese que el primer campo de texto del ejemplo tiene definido un valor por *default* con el atributo *value*.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title> Texto y Botones</title>
    </head>

    <body>
      <h1> Campos de Texto y Botones</h1>
      <p> Campo fuera del formulario: <input type="text" name="C"
                                 value= "Datos del Campo"></p>

      <p> Despues de introducir sus datos oprima el boton "ok"
      </p>
      <form action="paginaDestino.jsp" >
            <p>
                Nombre: <input type="text" name="Nombre"> <br>
                Apellido1: <input type="text" name="apell1"> <br>
                Apellido2: <input type="text" name="apell2"> <br>
                Telefono: <input type="text" name="tel">
                <input type="reset" value="Borrar">
            </p>
            <input type="submit" value="Ok">

        </form>


    </body>
</html>
```

Este campo no se borra con el botón *borrar*

En la Figura II-8 se observa como el botón que está dentro del formulario se despliega a la derecha de éste, mientras que el que está fuera se despliega abajo. En la Figura II8 a) se muestra una página en la que el usuario ya introdujo datos. Al dar clic en el botón "borrar" (Figura II-8 b), se borran los datos de los campos que están dentro del formulario, pero no el del campo que está fuera.

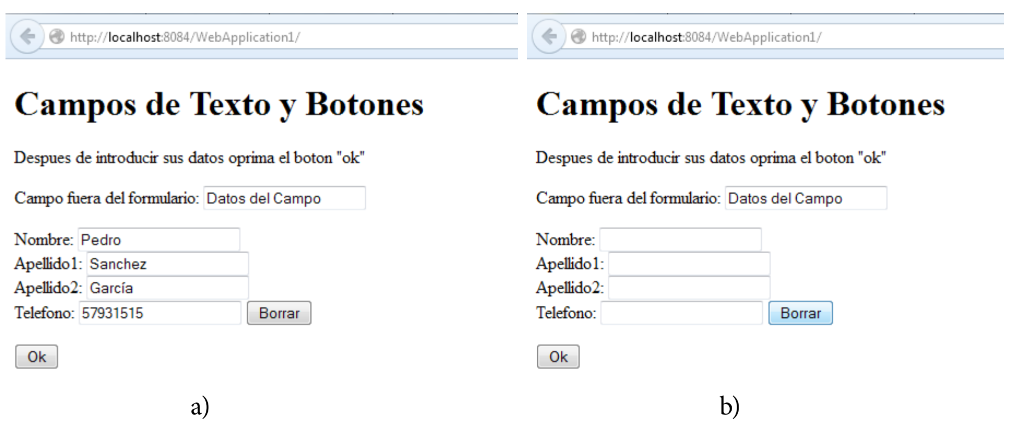


Figura II-8. Campos y botones dentro y fuera del formulario

2.5.3.2 Radio-botón (radio button)

Los radio-botones (*radio button*) sirven para seleccionar una de varias opciones. Aunque, también se pueden configurar para que el usuario pueda seleccionar dos o más opciones, éstos no se usan de esta manera, porque cuando el usuario selecciona un radio-botón éste ya no se puede des-seleccionar. En el siguiente código de ejemplo se muestra el funcionamiento de los radio-botones.

Nótese que el primer conjunto de radio-botones se despliega en una misma línea, mientras que en el segundo se despliega cada opción en una línea por separado. Además, en el primer grupo los tres tienen el mismo *name="transporte"*, por lo que el usuario sólo puede elegir una opción (transporte sólo puede tener un valor). El calificativo *checked* indica que ése es el radio-botón seleccionado por default. Cuando el usuario selecciona una opción, las demás opciones se des-seleccionan automáticamente.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <head>
       <meta http-equiv="Content-Type"
             content="text/html;charset=UTF-8">
        <title> Lista de opciones (list box)</title>
   </head>
   <body>
     <h1> Ejemplo de Radio-Botones <br>
               (Radio Buttons) </h1>
     <p> Elige que transporte prefieres </p>

     <input type="radio" name="transporte"
                         value="Automovil"checked> Coche
     <input type="radio" name="transporte" value="Avion"> Avión
     <input type="radio" name="transporte"
                         value="Camion"> Autobus <br>


     <p> Cuales son tus destinos favoritos? </p>

     <input type="radio" name="ciudad" value="Ciudad"> Ciudad <br>
     <input type="radio" name="bosque" value="Bosque"> Bosque <br>
     <input type="radio" name="playa" value="playa"> Playa <br>

   </body>
</html>
```

Los tres radio button tienen el mismo *name*

Cada radio button tiene su propio *name*

En la Figura II-9 a) se muestra la página antes de la selección del usuario y en la Figura II- 9 b) la selección del usuario.

Figura II-9. Radio-Botones con una sola selección y con varias selecciones

En el segundo grupo, cada radio-botón tiene un *name* diferente, y cada uno tiene su valor. Por lo tanto, el usuario puede seleccionar varias opciones. Sin embargo, si el usuario selecciona una, ésta ya no se puede des-seleccionar. Cuando se requiere que el usuario pueda seleccionar varias opciones, en lugar de utilizar radio botones se utilizan las casillas de verificación (*check box*), como se explica en la siguiente sección.

2.5.3.3 Casilla de verificación (Check box)

Un *check box* funciona como un interruptor encendido/apagado. Si la casilla no está marcada no se envía información. Solamente se envía la información cuando la casilla está marcada (*checked*), en este caso se envía *nombreCasilla= on*. En el siguiente ejemplo se muestran dos grupos de opciones. En el grupo 1, el usuario debe elegir sólo una de las opciones, por lo tanto se utilizan radio-botones. En el grupo 2, el usuario puede elegir varias. En este caso usamos las casillas de verificación, y cada casilla tiene su nombre.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type"
           content="text/html; charset=UTF-8">
        <title> Lista de opciones (list box)</title>
   </head>
   <body>
      <h1> Combinación de Radio-Botones <br>
           y Casillas de Verificación </h1>
      <p> Elige que transporte prefieres </p>            Grupo 1

      <input type="radio" name="transporte"
                          value="Automovil"checked> Coche
      <input type="radio" name="transporte" value="Avion"> Avión
      <input type="radio" name="transporte"
                          value="Camion"> Autobus <br>


      <p> Cuales son tus destinos favoritos? </p>        Grupo 2

      <input type="checkbox" name="ciudad"
             value="Ciudad">Ciudad<br>
      <input type="checkbox" name="bosque"
             value="Bosque">Bosque<br>
      <input type="checkbox" name="playa" value="playa">Playa <br>

   </body>
</html>
```

http://localhost:8084/WebApplication1/

Combinación de Radio-Botones y Casillas de Verificación

Elige que transporte prefieres

Coche Avión Autobus

Cuales son tus destinos favoritos?

Ciudad
Bosque
Playa

a)

http://localhost:8084/WebApplication1/

Combinación de Radio-Botones y Casillas de Verificación

Elige que transporte prefieres

Coche Avión Autobus

Cuales son tus destinos favoritos?

Ciudad
Bosque
Playa

b)

Figura II-10. Radio Botones y Casillas de Verificación

En la Figura II-10 se observa cómo el usuario puede elegir varias casillas a la vez, y las puede seleccionar y des-seleccionar libremente con el *mouse*.

2.5.3.4 Lista de opciones (Combo box, List box)

Con la lista de opciones el usuario selecciona (con clic) la opción deseada de una lista, en este caso se le llama “*Combo box*”. También es posible seleccionar varias opciones y a este caso se le llama “*list box*”. En la primera lista del siguiente ejemplo, llamada *transporte*, el usuario puede seleccionar solamente una opción, mientras que en la lista de opciones llamada *destinos*, el usuario puede seleccionar varias opciones, ya que se incluye el calificador ***multiple***.

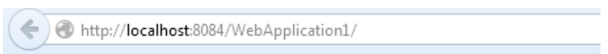
```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                 content="text/html; charset=UTF-8">
        <title> Lista de opciones (list box)</title>
    </head>
    <body>
        <h1> Ejemplo de Lista de Opciones</h1>
        <p> Elige que transporte prefieres </p>

        <select name="transporte">
            <option selected>Coche</option>
            <option>Avión</option>
            <option>Autobus</option>
        </select>
```

Por default la opción *coche* queda seleccionada

```
        <p> Cuales son tus destinos favoritos? <br>
            (ctrl-clic para elegir varias opciones)
        </p>

        <select name="destinos" multiple>
            <option>Ciudad</option>
            <option>Bosque</option>
            <option selected>Playa</option>
        </select>

    </body>
</html>
```

El calificativo *multiple* permite que el usuario seleccione varias opciones

En la Figura II-11 se muestra la página que despliega el navegador. Para seleccionar dos o más opciones, el usuario debe dar ctrl-clic.



Ejemplo de Lista de Opciones

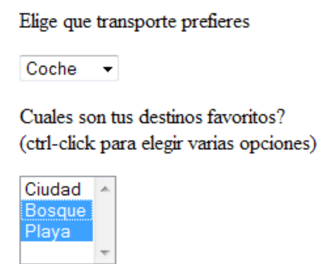


Figura II-11 Listas: en la primera se selecciona una opción, en la segunda, varias.

2.5.3.5 Area de texto

El área de texto sirve para capturar caracteres, como un campo de texto. La diferencia es que el área de texto está preparada para capturar muchos más caracteres que el campo de texto y desplegar varias líneas. Se despliega una barra de desplazamiento vertical para poder visualizar todo el texto.

Tag	Atributo	Descripción
<textarea>	• rows • cols	- Define el número de líneas visibles en el área de texto - Define el ancho del área de texto (en caracteres)

Tabla II-6. Atributos del tag textarea

El siguiente código despliega dos áreas de texto: la primera con menos renglones, menos columnas y con un texto que se despliega por *default*, la segunda con más renglones y columnas, se despliega en blanco.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
       <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
      <title> Area de texto</title>
  </head>
  <body>

    <h1> Area de texto</h1>
    <p> Se puede desplegar texto por default </p>
    <textarea name="area1" rows="3" cols="30"> Si la cantidad de palabras
del texto es mayor a los tres renglones que se especificaron para esta area
de texto, entonces se despliega automáticamente una barra de desplazamiento
vertical.</textarea>

    <p> O puede ser un area en blanco para capturar texto </p>
    <textarea name="area2" rows="4" cols="60"></textarea>

    </body>
</html>
```

En la Figura II-12 se muestra la página correspondiente al código anterior.

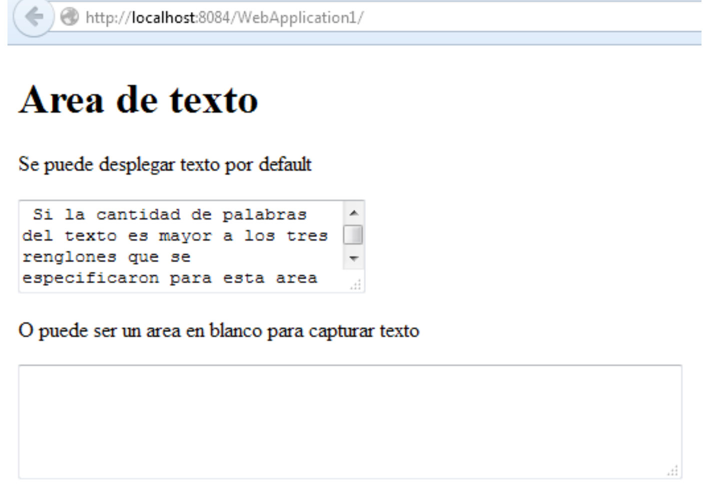


Figura II-12. Areas de texto

2.6 Prácticas

2.6.1 Práctica 1

Hacer una página HTML con una tabla que contenga tres columnas y dos renglones. En el primer renglón van los encabezados de la imágenes, y en el segundo renglón se despliegan las imágenes, como se muestra en la Figura II-13.

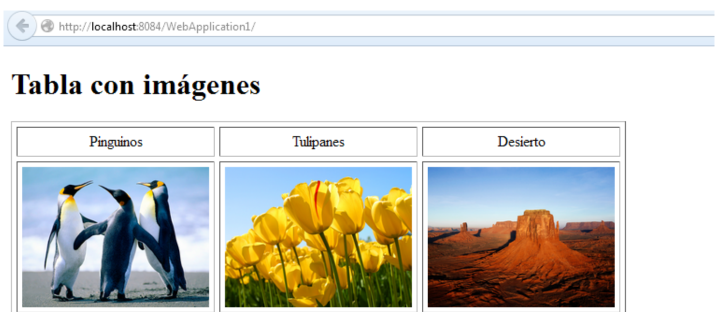


Figura II-13. Imágenes dentro de una tabla

2.6.2 Práctica 2

Hacer una página HTML que capture los datos en una solicitud como la que se muestra en la Figura II14. Observar que al oprimir el botón "borrar" se borran todos los datos de la solicitud. En la Figura II14 b) se pueden apreciar los campos borrados y los valores por *default*.

Figura II-14. Práctica 2: Solicitud

2.6.3 Solución

2.6.3.1 Solución de la práctica 1

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
      <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">
      <title> Tabla con imágenes </title>
  </head>
  <body>

    <h1> Tabla con imágenes</h1>
      <table cellspacing="5" cellpadding="5" border="1" >
        <tr>
          <td align="center"> Pinguinos </td>
          <td align="center"> Tulipanes </td>
          <td align="center"> Desierto </td>
        </tr>
        <tr>
            <td>
              <img src="Pingu.jpg" width="200" heigth="200"
             align="left" alt="No se pudo mostrar la imagen">
            </td>
            <td>
               <img src="Tulips.jpg" width="200" heigth="200"
             align="left" alt="No se pudo mostrar la imagen">
            </td>
            <td>
                <img src="Desert.jpg" width="200" heigth="200"
                     align="left"
                     alt="No se pudo mostrar la imagen">

            </td>
        </tr>
      </table>

    </body>
</html>
```

2.6.3.2 Solución de la práctica 2

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
       <meta http-equiv="Content-Type"
             content="text/html; charset=UTF-8">
      <title> Solicitud </title>
  </head>
  <body>

    <h1> Solicitud</h1>
    <p> Despues de introducir tus datos oprime el botón
        "enviar"</p>

    <form action="paginaDestino.jsp" >
      <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
              <td align="right"> Nombre: </td>
              <td><input type="text" name="Nombre"></td>
          </tr>
          <tr>
              <td align="right"> Apellidos: </td>
              <td> <input type="text" name="apellidos"> </td>
          </tr>
          <tr>
              <td align="right"> Correo: </td>
              <td> <input type="text" name="mail"> </td>
          </tr>
      </table>

      <p> Eres:
        <input type="radio" name="genero"
                            value="masculino"checked> Hombre
        <input type="radio" name="genero"
                            value="femenino">Mujer<br>

        Selecciona lo que sabes: <br>
        <input type="checkbox" name="java" value="java"> Java
        <input type="checkbox" name="c" value="c">C/C++
        <input type="checkbox" name="basic" value="basic">Basic
        <input type="checkbox" name="html" value="html">HTML <br>
      </p>
      <table cellspacing="5" cellpadding="5" border="0" >
        <tr>
         <td align="center">Selecciona los idiomas que comprendes:
          </td>
          <td align="center"> Comentarios adicionales </td>
        </tr>
        <tr>
            <td>
              <select name="idiomas" multiple>
                 <option selected>Español</option>
                 <option>Inglés</option>
                 <option>Frances</option>
                 <option>Alemán</option>
              </select>
            </td>
```

```
            <td>
                <textarea name="comentario" rows ="3" cols="20">
                </textarea>
            </td>
        </tr>
      </table>


      <input type="reset" value="Borrar">
      <input type="submit" value="Enviar">

    </form>

   </body>
</html>
```

3. Comunicación entre páginas Web: paso de parámetros

3.1 Objetivo

Usar una JSP para capturar datos del usuario, y enviar la información capturada a otra JSP que despliegue esta información.

3.2 Transición de una página a otra (enlace)

Una aplicación Web contiene un conjunto de páginas de internet que están conectadas entre sí por medio de *ligas* (*links*). Una liga contiene la dirección de URL de la nueva página que se desplegará. La *URL* es una cadena de caracteres con la dirección en donde se encuentra la página destino.

Cuando queremos pasar de una página a otra sin enviar algún tipo de información, utilizamos un *enlace*. El *tag* de HTML llamado "ancla", permite pasar de una pagina1.JSP a otra pagina2.JSP. Para hacer esto, es necesario poner el ancla en la pagina1.JSP, cuyo formato es el siguiente:

<**a href**=" URL de la pagina2.JSP"> *Texto del enlace* </**a**>

Cuando el navegador despliega la pagina1 la liga se muestra con el texto indicado, en este caso: *Texto del enlace*. Cuando el usuario da clic en esta liga, el navegador despliega ahora la *página2.JSP*.

En la Figura III-1 se muestra la transición de *index.jsp* a la *pagina2.jsp* y luego a la *pagina3.jsp* por medio de un enlace.

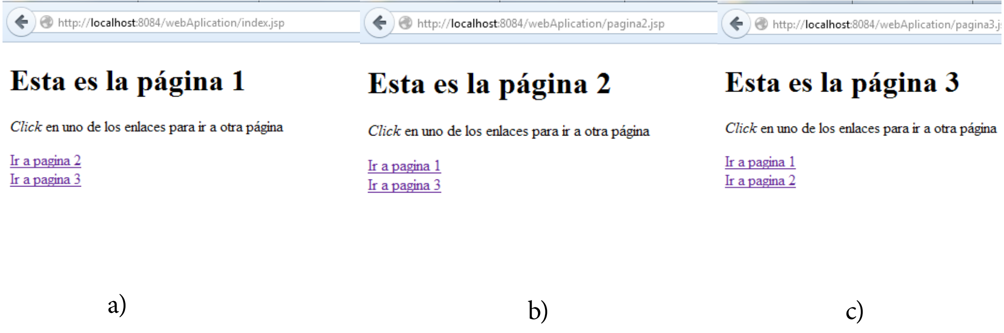


Figura III-1. Enlace entre páginas Web

El código de la *index.jsp* es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
         <meta http-equiv="Content-Type"
                  content="text/html; charset=UTF-8">
        <title>JSP 1</title>
    </head>
    <body>
        <h1> Esta es la página 1</h1>
        <p> <i>Clic</i> en uno de los enlaces para ir a otra
                         página</p>
        <a href="pagina2.jsp"> Ir a pagina 2</a> <br>
        <a href="pagina3.jsp"> Ir a pagina 3</a>
    </body>
</html>
```

El código de la *pagina2.jsp* es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
        <title>JSP 2</title>
    </head>
    <body>
        <h1> Esta es la página 2</h1>
        <p> <i>Clic</i> en uno de los enlaces para ir a otra
                         página</p>
        <a href="index.jsp"> Ir a pagina 1</a> <br>
        <a href="pagina3.jsp"> Ir a pagina 3</a>
    </body>
</html>
```

El código de la *pagina3.jsp* es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
       <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
        <title>JSP 3</title>
    </head>
    <body>
        <h1> Esta es la página 3</h1>
        <p> <i>Clic</i> en uno de los enlaces para ir a otra
                         página</p>
        <a href="index.jsp"> Ir a pagina 1</a> <br>
        <a href="pagina2.jsp"> Ir a pagina 2</a>
    </body>
</html>
```

Las páginas JSP deben estar en la misma carpeta. Si la página JSP se encuentra en otro lugar, será necesario proporcionar la ruta.

3.3 La petición de una JSP con el método GET

Como se estudió en el capítulo I, la comunicación entre el cliente y el servidor se lleva a cabo mediante el protocolo de comunicación HTTP el cual consta de peticiones (*request*) y respuestas (*response*). (Ver figura I.3). Entre los tipos de peticiones HTTP más conocidas están: GET y POST, las cuales sirven para pasar información (parámetros) del cliente al servidor.

La petición GET se utiliza para solicitar datos que están en el servidor, por ejemplo, para solicitar una página.

Las peticiones GET se pueden guardar en el historial de navegación, se pueden indexar en los buscadores, o agregar a enlaces favoritos. La información que se envía en el GET normalmente es una liga. Get contiene mucho menos información que POST.

El *tag* ancla: siempre utiliza el método GET para solicitar la página destino.

3.4 La petición de una JSP con el método POST

La petición POST sirve para enviar información al servidor para que ésta sea procesada. Una vez que se procesa, se envía en la respuesta al navegador alguna página con información. En una petición POST puede viajar mucho más información.El valor de los parámetros que proporcionó el usuario en una petición POST, se encuentran dentro de un objeto llamado *request*.

En la siguiente sección utilizaremos el método POST para enviar información de una página JSP a otra.

3.5 Scriptlets y expresiones en una JSP

Un *scriptlet* es un fragmento de código en Java que se incrusta en una página JSP. Para insertar un *scriptlet* en una JSP, se utiliza la siguiente sintaxis:

```
<% código en Java%>
```

A continuación, se muestra un scriptlet que sirve para recuperar los parámetros que recibe la JSP en el objeto `request` y guardarlas en variables `String`.

```
<%
    // Este es un scriptlet
    // Es código en Javaque captura los parámetros enviados
    // en el objeto "request"
    String nombre = request.getParameter("nombre");
    String color = request.getParameter("color");
    String mail = request.getParameter("mail");
 %>
```

Una *expresión* se utiliza para desplegar como texto, una variable Java. Para insertar una *expresión* en una JSP se usa la siguiente sintaxis:

```
<%= nombreVariable %>
```

En el siguiente ejemplo se muestran las expresiones que sirven para desplegar el contenido de las tres variables del scriptlet anterior:

```
<%= nombre %>
<%= color %>
<%= mail %>
```

La Figura III.2 a) muestra una página JSP llamada index.jsp que captura datos del usuario. Al oprimir el botón "enviar", *index.jsp* envía los datos (parámetros) dentro del objeto `request` a la paginaDestino.jsp. La *paginaDestino.jsp* de la Figura III.2 b) despliega los parámetros recibidos. El botón "*regresar*" en la *paginaDestino.jsp* sirve para regresar a *index.jsp*.

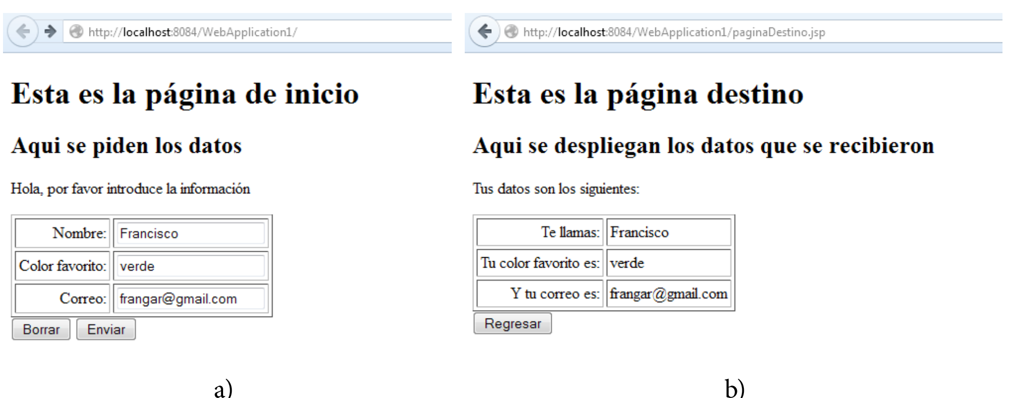


Figura III-2. Paso de parámetros de una página Web a otra

El código de *index.jsp* es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
      <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">
      <title> Area de texto</title>
  </head>
  <body>

    <h1> Esta es la página de inicio</h1>
    <h2> Aqui se piden los datos </h2>
    <p> Hola, por favor introduce la información </p>
    <form action="paginaDestino.jsp" method="post">
       <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
              <td align="right"> Nombre: </td>
              <td><input type="text" name="nombre"></td>
          </tr>
          <tr>
              <td align="right"> Color favorito: </td>
              <td> <input type="text" name="color"> </td>
          </tr>
          <tr>
              <td align="right"> Correo: </td>
              <td> <input type="text" name="mail"> </td>
          </tr>
       </table>

       <input type="reset" value="Borrar">
       <input type="submit" value="Enviar">
    </form>

  </body>
</html>
```

En *action* se indica el destino

El formulario envía la petición con el método POST

Estos son los nombres de los parámetros (atributos) que se envían en request

index.jsp pide los datos al usuario y las capturas en los atributos llamados `nombre`, `color` y `mail`. Cuando el usuario oprime el botón “*Enviar*”, se ejecuta la acción “ir a *paginaDestino.jsp*”. Si no se capturó un dato, se envía el valor *null*.

El código de *paginaDestino.jsp* es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>Pagina Destino JSP</title>
    </head>
    <body>

      <%
         // Este es un scriptlet
         // Es código en Javaque captura los parámetros enviados
         // en el objeto "request"
         String nombre = request.getParameter("nombre");
         String color = request.getParameter("color");
         String mail = request.getParameter("mail");
      %>

      <h1> Esta es la página destino</h1>
      <h2> Aqui se despliegan los datos que se recibieron</h2>
      <p> Tus datos son los siguientes: </p>

      <table cellspacing="3" cellpadding="3" border="1" >
        <tr>
           <td align="right"> Te llamas: </td>
           <td> <%= nombre %> </td>
        </tr>
        <tr>
           <td align="right"> Tu color favorito es: </td>
           <td> <%= color %> </td>
        </tr>
        <tr>
           <td align="right"> Y tu correo es: </td>
           <td> <%= mail %> </td>
        </tr>
      </table>

    <form action="index.jsp" method="post">
        <input type="submit" value="Regresar">
    </form>

    </body>
</html>
```

Son los mismos nombres de los atributos que envió la JSP previa, con el método POST

Éstas son las expresiones que se despliegan como texto

3.6 Los *tags* en una JSP

Existen varios *tags* que se pueden utilizar dentro de una JSP; ya hemos visto dos de ellos: el *tag* para insertar un *scriptlet* y el *tag* para insertar una expresión. En la Tabla III-1 se muestra un resumen con todos los *tags* y su uso.

Tabla III-1. Tags en una JSP		
Tag	Nombre	Descripción
<% %>	Scriptlet	Sirve para insertar un bloque de código Java.
<%= %>	Expresión	Despliega el valor de una expresión en forma de texto.
<%@ %>	Directiva	Establece una condición para toda la JSP
<%-- --%>	Comentario	Se ignora lo que está dentro del tag
<%! %>	Declaración	Para declarar variables y métodos en una JSP

Tabla III-1. Tags en una JSP

A lo largo del curso veremos ejemplos en los que utilizaremos estos *tags*.

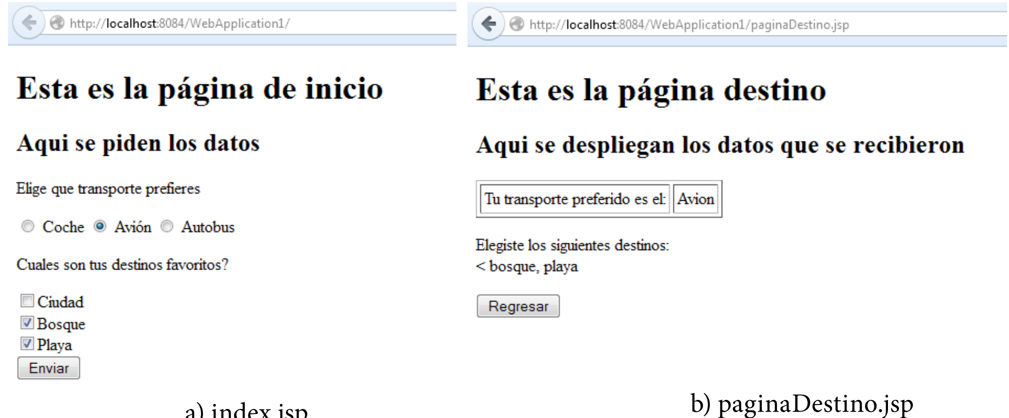
3.7 Recepción de los valores de radio-button y checkbox

La página de la Figura III.3 a), llamada *index.jsp,* captura los datos del usuario por medio de radio-botones y casillas de verificación. Su código es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type"
             content="text/html; charset=UTF-8">
        <title> Captura con Radio-button y Checkbox</title>
  </head>
  <body>
   <h1> Esta es la página de inicio</h1>
   <h2> Aqui se piden los datos </h2>

   <form action="paginaDestino.jsp" method="post">

     <p> Elige que transporte prefieres </p>
```

```
    <input type="radio" name="transporte"
                        value="Automovil"checked> Coche
    <input type="radio" name="transporte" value="Avion"> Avión
    <input type="radio" name="transporte"
                        value="Camion"> Autobus <br>


    <p> ¿Cuáles son tus destinos favoritos? </p>

    <input type="checkbox" name="ciudad" value="Ciudad">Ciudad<br>
    <input type="checkbox" name="bosque" value="Bosque">Bosque<br>
    <input type="checkbox" name="playa" value="playa">Playa <br>

    <input type="submit" value="Enviar">
   </form>

   </body>
</html>
```

Los datos capturados en *index.jsp* se envían a *paginaDestino.jsp*, la cual muestra la información recibida, como se observa en la Figura III3-b).



a) index.jsp b) paginaDestino.jsp

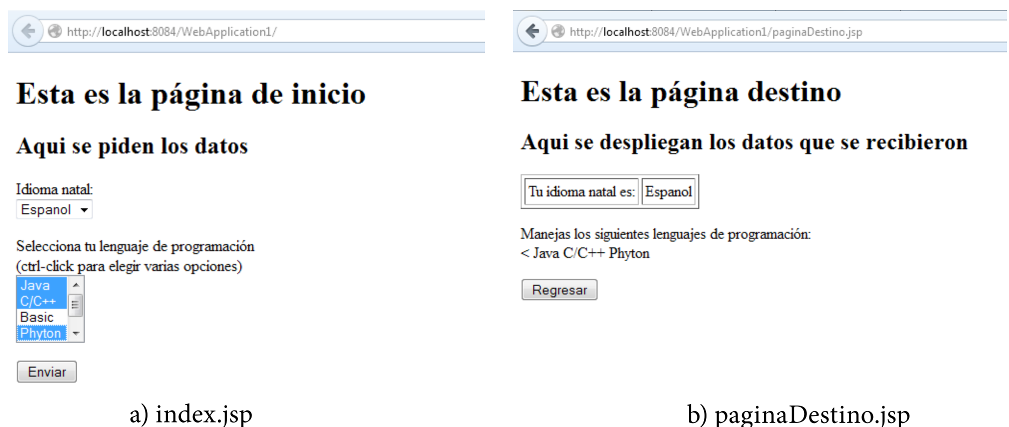
Figura III-3. Captura y despliegue con Radio-botones y Casillas de Verificación

La página *destino.jsp* de la Figura III-3 b) recupera la información con un scriptlet que captura los valores de cada uno de los atributos enviados por *index.jsp*. Posteriormente despliega la información recibida usando *scriptlets* que contienen el condicional `if`, como se aprecia en el siguiente código:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                 content="text/html; charset=UTF-8">
        <title>Pagina Destino JSP</title>
    </head>
    <body>

      <%
         // Este es un scriptlet
         // Es código en Javaque captura los parámetros enviados
         // en el objeto "request"
         String transp = request.getParameter("transporte");
         String ciudad = request.getParameter("ciudad");
         String bosque = request.getParameter("bosque");
         String playa = request.getParameter("playa");
      %>

      <h1> Esta es la página destino</h1>
      <h2> Aquí se despliegan los datos que se recibieron</h2>

      <table cellspacing="3" cellpadding="3" border="1" >
        <tr>
           <td align="right"> Tu transporte preferido es el: </td>
           <td><%= transp %></td>
        </tr>
      </table>

      <p> Elegiste los siguientes destinos: <br><
          <% if ( ciudad != null) { %>
                  ciudad,

          <% }%>

          <% if ( bosque != null) { %>
                  bosque,

           <% }%>
           <% if ( playa != null) { %>
                   playa

           <% }%>

      <form action="index.jsp" method="post">
        <input type="submit" value="Regresar">
      </form>

    </body>
</html>
```

3.8 Recepción de valores en lista de opciones (combo box, list box)

En un *combo box* solamente se puede seleccionar una opción. La página destino obtiene el valor seleccionado con el valor de su atributo, es decir, como se hace para los campos de texto o los radio-botones. Sin embargo, cuando se trata de una lista con múltiples opciones (*list box*), es necesario un poco más de procesamiento.



a) index.jsp b) paginaDestino.jsp

Figura III 4 Captura y despliegue con lista de opciones

En la Figura III4 a) se muestra una página *index.jsp* que captura la información del usuario por medio de una *combo box* y una *list box*. Su código es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
      <meta http-equiv="Content-Type"
               content="text/html; charset=UTF-8">
      <title> Lista de opciones</title>
  </head>
  <body>

   <h1> Esta es la página de inicio</h1>
   <h2> Aquí se piden los datos </h2>

   <form action="paginaDestino.jsp" method="post">
      <p> Idioma natal: <br>

      <select name="idioma">
          <option selected>Espanol</option>
          <option>Ingles</option>
          <option>Frances</option>
          <option>Aleman</option>
      </select>
      </p>
```

```
      <p> Selecciona tu lenguaje de programación <br>
            (ctrl-clic para elegir varias opciones)<br>

      <select name="lenguajes" multiple>
          <option Selected>Java</option>
          <option>C/C++</option>
          <option>Basic</option>
          <option>Phyton</option>
          <option>Pascal</option>
      </select>
      </p>

      <input type="submit" value="Enviar">
    </form>

  </body>
</html>
```

La *páginaDestino.jsp* recibe la información de *index.jsp* y la despliega, como se muestra en la Figura III4-b).

En el código de *páginaDestino.jsp* podemos observar que, para obtener todas las opciones que seleccionó el usuario para el atributo “lenguajes”, en el scriptlet del principio se captura la referencia al arreglo de *String* que contiene las opciones seleccionadas. Posteriormente, con la combinación de un scriptlet y una expresión se despliegan estas opciones en la página.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                 content="text/html; charset=UTF-8">
        <title>Pagina Destino JSP</title>
    </head>
    <body>
      <%
         // Extracción de los parámetros recibidos
         String idioma = request.getParameter("idioma");
         String[] lenguajesSelec =
                          request.getParameterValues("lenguajes");
      %>

      <h1> Esta es la página destino</h1>
      <h2> Aquí se despliegan los datos que se recibieron</h2>

      <table cellspacing="3" cellpadding="3" border="1" >
        <tr>
           <td align="right"> Tu idioma natal es: </td>
           <td><%= idioma %></td>
        </tr>
      </table>
```

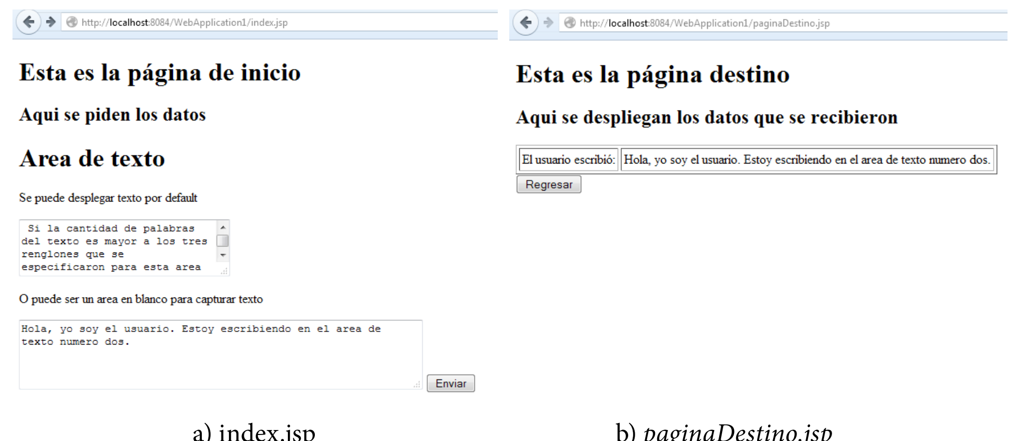
```
      <p> Manejas los siguientes lenguajes de programación: <br>
          <%
             for( int i=0; i< lenguajesSelec.length; i++)
             {
          %>
             <%= lenguajesSelec[i] %>
          <% }
          %>
      </p>

      <form action="index.jsp" method="post">
        <input type="submit" value="Regresar">
      </form>

    </body>
</html>
```

3.9 Recepción de un área de texto

La recepción de los caracteres que escribe el usuario en un área de texto, es muy sencilla, y se hace de la misma manera que para un campo de texto. El siguiente código es de la página origen index.jsp, cuya pantalla se muestra en la Figura III5-a).

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
       <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
      <title> Area de texto</title>
  </head>
  <body>

    <h1> Area de texto</h1>
    <p> Se puede desplegar texto por default </p>
    <textarea name="area1" rows="3" cols="30"> Si la cantidad de palabras
del texto es mayor a los tres renglones que se especificaron para esta area
de texto, entonces se despliega automáticamente una barra de desplazamiento
vertical.</textarea>

    <p> O puede ser un area en blanco para capturar texto </p>
    <textarea name="area2" rows="4" cols="60"></textarea>

  </body>
</html>
```

La página de la Figura III5a) contiene el texto que introdujo el usuario en el área de texto 2, en la Figura III5 b) la página destino muestra el texto recibido como parámetro en el atributo área 2.



a) index.jsp b) *paginaDestino.jsp*

Figura III-5. Paso de un área de texto a una página destino

El código de la página destino de la Figura III5 b) es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
       <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
       <title>Pagina Destino JSP</title>
    </head>
    <body>
        <%
           // Extracción de los parámetros recibidos
           String areaTexto = request.getParameter("area2");

        %>

        <h1> Esta es la página destino</h1>
        <h2> Aquí se despliegan los datos que se recibieron</h2>

         <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
              <td align="right"> El usuario escribió: </td>
              <td><%= areaTexto %></td>
          </tr>
          </table>
```

```
    <form action="index.jsp" method="post">
        <input type="submit" value="Regresar">
    </form>

    </body>
</html>
```

3.10 Práctica

Hacer una página HTML que capture los datos en una solicitud como la que se muestra en la Figura II14 a). Que es la misma que la de la práctica 2 del capítulo II. Al oprimir el botón "*Enviar*" se deben pasar todos los datos capturados en la solicitud a una *paginaDestino.jsp* como la de la Figura III6 b) en la que se despliegan todos los datos recibidos.

a) *index.jsp* b) *paginaDestino.jsp*

Figura III-6. Práctica: Solicitud

3.10.1 Solución

El código de la página *index.jsp* es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
       <meta http-equiv="Content-Type"
             content="text/html; charset=UTF-8">
      <title> Solicitud </title>
  </head>
  <body>

    <h1> Solicitud</h1>
    <p> Despues de introducir tus datos oprime el botón
        "enviar"</p>

    <form action="paginaDestino.jsp" >
      <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
              <td align="right"> Nombre: </td>
              <td><input type="text" name="Nombre"></td>
          </tr>
          <tr>
              <td align="right"> Apellidos: </td>
              <td> <input type="text" name="apellidos"> </td>
          </tr>
          <tr>
              <td align="right"> Correo: </td>
              <td> <input type="text" name="mail"> </td>
          </tr>
      </table>

      <p> Eres:
        <input type="radio" name="genero"
                            value="masculino"checked> Hombre
        <input type="radio" name="genero"
                            value="femenino">Mujer<br>

        Selecciona lo que sabes: <br>
        <input type="checkbox" name="java" value="java"> Java
        <input type="checkbox" name="c" value="c">C/C++
        <input type="checkbox" name="basic" value="basic">Basic
        <input type="checkbox" name="html" value="html">HTML <br>
      </p>
      <table cellspacing="5" cellpadding="5" border="0" >
        <tr>
         <td align="center">Selecciona los idiomas que comprendes:
          </td>
          <td align="center"> Comentarios adicionales </td>
        </tr>
        <tr>
            <td>
              <select name="idiomas" multiple>
                 <option selected>Español</option>
```

```
                <option>Inglés</option>
                <option>Frances</option>
                <option>Alemán</option>
             </select>
           </td>
       </table>


       <input type="reset" value="Borrar">
       <input type="submit" value="Enviar">

     </form>

   </body>
</html>
```

Y el código de la *paginaDestibo.jsp* es:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                 content="text/html; charset=UTF-8">
        <title>Pagina Destino JSP</title>
    </head>
    <body>
       <%
           // Extracción de los parámetros recibidos
           String nombre = request.getParameter("nombre");
           String apellidos = request.getParameter("apellidos");
           String mail = request.getParameter("mail");
           String genero = request.getParameter("genero");

           String Java= request.getParameter("java");
           String c = request.getParameter("c");
           String basic = request.getParameter("basic");
           String html = request.getParameter("html");

           String[] idiomasSelec =
                            request.getParameterValues("idiomas");
        %>

        <h1> Solicitud Recibida</h1>
        <h2> Tus datos son: </h2>

        <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
              <td align="right"> Te llamas: </td>
              <td> <%= nombre %> </td>
```

```
          </tr>
          <tr>
              <td align="right"> Apellidos: </td>
              <td> <%= apellidos %> </td>
          </tr>
          <tr>
              <td align="right"> Tu correo es: </td>
              <td> <%= mail %> </td>
          </tr>
          <tr>
              <td align="right"> Eres: </td>
              <td> <%=genero %> </td>
          </tr>

         </table>

        <p> Manejas los siguientes lenguajes de programación: <br>
              <% if ( Java!= null) { %>
                 Java,

              <% }%>

              <% if ( c != null) { %>
                 C/C++,

              <% }%>
              <% if ( basic != null) { %>
                 basic

              <% }%>
              <% if ( html != null) { %>
                 html

              <% }%>

          </p>
          <p> Los idiomas que comprendes son: <br>
              <%
                 for( int i=0; i< idiomasSelec.length; i++)
                 {
              %>
                 <%= idiomasSelec[i] %>
              <% }
              %>
          </p>

      <form action="index.jsp" method="post">
         <input type="submit" value="Regresar">
      </form>


    </body>
</html>
```

4. Comunicación entre JSPs y clases Java

4.1 Objetivos

- Capturar la información del usuario en una JSP inicial y enviarla a una clase *Java* que procesa la información.
- Desplegar los resultados que proporciona una clase *Java* en una JSP.
- Utilizar un archivo de texto para guardar la información capturada.
- Leer información de un archivo de texto y desplegarla mediante una JSP.

4.2 Conceptos básicos

Hasta el capítulo anterior hemos estudiado cómo capturar información del usuario en una página inicial y pasarla a una página destino para que se despliegue. Sin embargo, hasta ahora no hemos hecho ningún tipo de procesamiento a la información.

Es posible incluir código Java en las JSPs para procesar los datos, pero esto no es una buena práctica. Lo recomendable es utilizar algún patrón que organice la aplicación en partes independientes. Para aplicaciones muy sencillas, utilizamos páginas JSP para capturar y mostrar información, es decir, para implementar *la vista*. Y el procesamiento se hace con clases normales de *Java*. Sin embargo, para aplicaciones reales, que tienen mayor complejidad, es mejor utilizar un patrón de diseño, como el Modelo-Vista-Controlador MVC que estudiaremos en el capítulo V.

4.2.1 Los hilos (*threads*) en una JSP

Cuando se solicita una página JSP por primera vez, el motor JSP genera un *servlet* correspondiente a la página (se le llama *instancia de la JSP*). La instancia de la JSP se guarda en la memoria del servidor. Si la JSP contiene código Java, también se inicia lo que se llama un *hilo (thread)*. Posteriormente, cada que hay otra petición de la página, se utiliza la instancia de la JSP y además, se crea otro hilo por cada nueva petición de la página, como se ilustra en la Figura IV1.

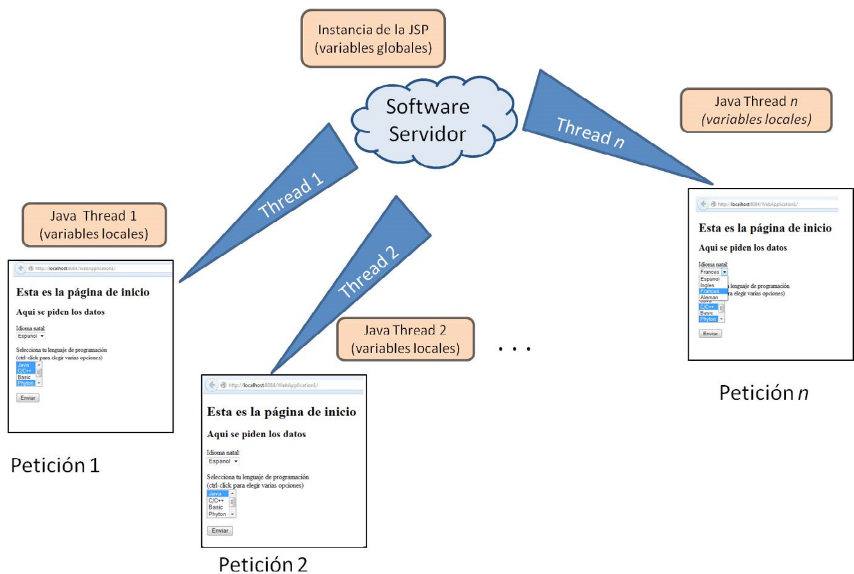


Figura IV-1. Creación de un hilo de Java por cada petición de una JSP

Las variables globales se guardan en la instancia de la JSP, y se declaran dentro del tag <%! %>. Las variables locales se crean para cada *thread*, y se declaran dentro de un *scriptlet*. Por ejemplo, en el siguiente código se declara una variable global que lleva la cuenta de las peticiones que se han hecho sobre la página.

Para comprender el código del ejemplo, recordemos que:

Tabla III-1		
Tag	Nombre	Descripción
<% %>	Scriptlet	Sirve para insertar un bloque de código Java.
<%= %>	Expresión	Despliega el valor de una expresión en forma de texto.
<%@ %>	Directiva	Establece una condición para toda la JSP
<%-- --%>	Comentario	Se ignora lo que está dentro del tag
<%! %>	Declaración	Para declarar variables y métodos en una JSP

(Tabla III1)

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
       <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
      <title> Cuenta total </title>
 </head>
 <body>

   <%@ page import="java.util.Date" %>
   <%! int cuentaGlobal = 0; %>


   <% cuentaGlobal++; %>

   <h1> En esta página se lleva la cuenta del número
        de accesos</h1>

   <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
              <td align="right"> Fecha y hora: </td>
              <td><%= new Date() %></td>
          </tr>
   </table>
   </br>
   <table cellspacing="5" cellpadding="5" border="1.5" >
          <tr>
              <td> Número de accesos: </td>
              <td><%= cuentaGlobal %></td>
          </tr>
   </table>
 </body>
</html>
```

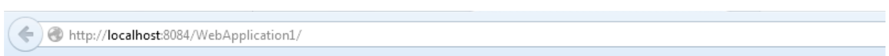
Para importar clases (directiva)

Variable global para esta JSP (declaración)

La variable se incrementa cada vez que se accesa la página (scriptlet)

(expresiones)

En la Figura IV2 se observa que esta página se solicitó tres veces. Esta técnica puede fallar cuando dos computadoras solicitan la página JSP exactamente al mismo tiempo. Se le dice en inglés *no thread-safe* cuando se manejan las variables globales de esta forma. En este caso, no sería grave que se perdiera la cuenta. Para los casos en que es indispensable que no se pierda la cuenta, existen métodos más sofisticados para manejar variables globales de forma más segura (*thread-safe*).



En esta página se lleva la cuenta del número de accesos

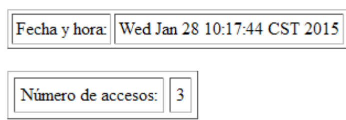


Figura IV-2. Número de accesos a una misma página JSP

4.2.2 Encapsulamiento

Uno de los principios a seguir en el desarrollo Web es el encapsulamiento. El encapsulamiento sirve para proteger los datos de los objetos. Éste se logra declarando los atributos de una clase como `private` y codificando métodos especiales para controlar el acceso. La manera de acceder a los atributos desde fuera de la clase es por medio de los métodos *getter*. Y la manera de modificar los atributos desde fuera de la clase es usando los métodos *setter*.

4.2.2.1 Métodos getter

Los *getters* no reciben parámetros y el tipo de dato que regresan es el mismo que el del atributo correspondiente. Su nombre comienza con "get" seguido del nombre del atributo, pero inician con mayúscula, y regresan el valor del atributo. Por ejemplo:

```
private double saldo;
private int cuenta;
. . .
public double getSaldo( ){
  return saldo;
}

public int getNumCuenta( ){
  return numCuenta;
}
```

4.2.2.2 Métodos setter

Para que otros objetos puedan modificar el valor de los atributos de una clase, usamos los métodos *setter*, por ejemplo:

```
public void setSaldo(double s){
  saldo = s;
}

public void setNumCuenta(int num){
  numCuenta = num;
}
```

Los *setters* reciben como parámetro el mismo tipo de dato que el del atributo. El nombre de los métodos *setter* se construye de forma análoga a la de los *getters*, pero iniciando con la palabra "set", y asignan al atributo el valor del parámetro recibido. El parámetro recibido tiene

el mismo nombre que el atributo. Para diferenciar entre el valor enviado como parámetro y el nombre del atributo se utiliza la palabra 'this'. De tal forma que `this.nombreAtributo` se refiere al atributo del objeto. Por ejemplo:

```
public void setSaldo(double saldo){
 // el parámetro saldo lo asigna al atributo this.saldo
 this.saldo = saldo;
}

public void setNumCuenta(int numCuenta){
 // el parámetro numCuenta lo asigna al atributo this.num-
Cuenta
 this.numCuenta = numCuenta;
}
```

4.2.3 La *ruta real* para acceder a los archivos

Cuando se trabaja con JSPs y *servlets*, se usan las *rutas relativas* para hacer referencia a los archivos, por ejemplo: "/WEB-INF/Promedios.txt". Sin embargo para leer o guardar datos en un archivo, es necesario tener la ruta completa, es decir la *ruta real* del archivo.El *servlet Context* (contexto del *servlet*) maneja la información a nivel de toda la aplicación Web. La clase `ServletContext` contiene métodos que sirven para que un *servlet* se comunique con su contenedor. Todos los *servlets* de una aplicación tienen el mismo `ServletContext`.

El `ServletContext` contiene un método que sirve para obtener la ruta real de un archivo que está dentro del proyecto de la aplicación. Entonces, para obtener la ruta real de "/WEB-INF/Promedios.txt" hacemos:

```
  ServletContext sc = this.getServletContext();
String path = sc.getRealPath("/WEB-INF/Promedios.txt");
```

El contenido de la variable path es el siguiente:

C:\Users\usuario\Documents\NetBeansProjects\WebApplication1\build\web\WEB-INF\Promedios.txt

La ruta expresada con diagonales invertidas: "\" es un problema en los sistemas operativos que requieren la diagonal normal "/". En nuestro caso, utilizaremos la siguiente instrucción para solucionar el problema:

```
path = path.replace('\\','/');
```

Los `String` tienen el método `replace`, que sirve para cambiar un caracter por otro. Primero se debe poner el caracter que se desea remplazar y después el caracter correcto. Estos caracteres deben ir entre comillas simples y separados por una coma. Además, en este caso, la diagonal invertida tiene un significado adicional, entonces usamos lo que se llama el "código de escape" que consiste en poner doble diagonal, de esta forma "\" se interpreta como el caracter diagonal invertida y no como el inicio de un código de control (por ejemplo, cambio de linea: \n).

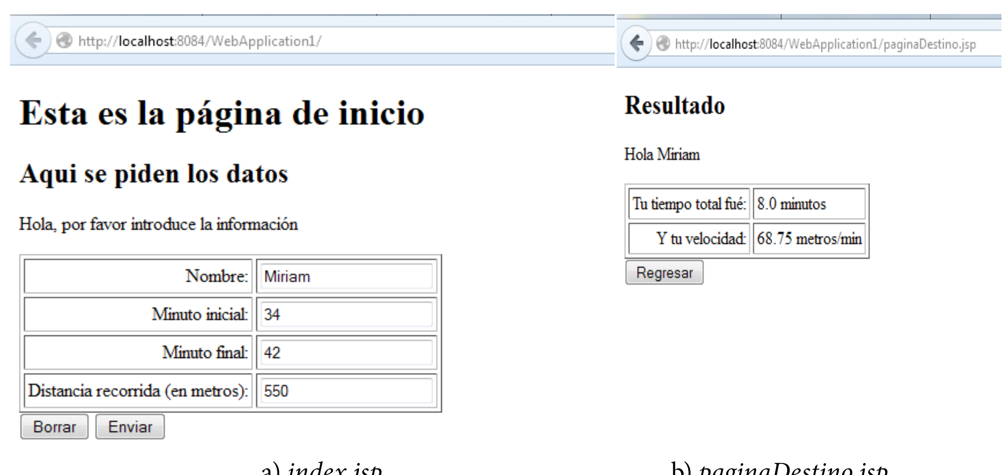
Después de haber usado el método replace, la ruta queda como se indica:

C:/Users/usuario/Documents/NetBeansProjects/WebApplication1/build/web/WEB-INF/Promedios.txt

4.3 Procesamiento de los datos de una JSP con una clase normal Java

4.3.1 Haciendo cálculos con una clase Java

Como ya habíamos mencionado, es una buena práctica separar las funcionalidades de tal forma que las JSP realicen las funciones de la *vista* y delegar el procesamiento de los datos a las clases Java (incluyendo los *servlets)*. Para una aplicación sencilla, como la del siguiente ejemplo, no es necesario utilizar *servlets*, basta con una clase normal de Java. En la Figura IV3 se ilustra un ejemplo de aplicación Web en la que se requiere hacer cálculos.



a) *index.jsp* b) *paginaDestino.jsp*

Figura IV-3. Aplicación web que hace cálculos con una clase Java

En la página de inicio (Figura IV3 a), se solicitan los datos. Cuando estos datos se envían al servidor, la aplicación Web debe realizar los cálculos y después desplegarlos, como se muestra en la Figura IV3 b). La página de inicio es muy sencilla, y se muestra a continuación:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
      <meta http-equiv="Content-Type"
               content="text/html; charset=UTF-8">
      <title> Captura Datos </title>
  </head>
  <body>
    <h1> Esta es la página de inicio</h1>
    <h2> Aquí se piden los datos </h2>
    <p> Hola, por favor introduce la información </p>

    <form action="paginaDestino.jsp" method="post">
       <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
              <td align="right"> Nombre: </td>
              <td><input type="text" name="nombre"></td>
          </tr>
          <tr>
              <td align="right"> Minuto inicial: </td>
              <td> <input type="text" name="tiempoIni"> </td>
          </tr>
          <tr>
              <td align="right"> Minuto final: </td>
              <td> <input type="text" name="tiempoFin"> </td>
          </tr>
          <tr>
              <td align="right">Distancia recorrida
                                  (en metros):</td>
              <td> <input type="text" name="distancia"> </td>
          </tr>
       </table>

       <input type="reset" value="Borrar">
       <input type="submit" value="Enviar">
    </form>
    </body>
</html>
```

La siguiente clase Java, llamada `Calcula` es la encargada de llevar a cabo los cálculos sobre los datos proporcionados por el usuario. Es importante que desde el principio nos acostumbremos a hacer paquetes en donde se agrupen las funcionalidades. Recordar que a la realización de los cálculos se le llama *lógica de negocio (business)*. Entonces, haremos un paquete llamado *negocios* dentro de los *source packages*, como se muestra en la Figura IV4.

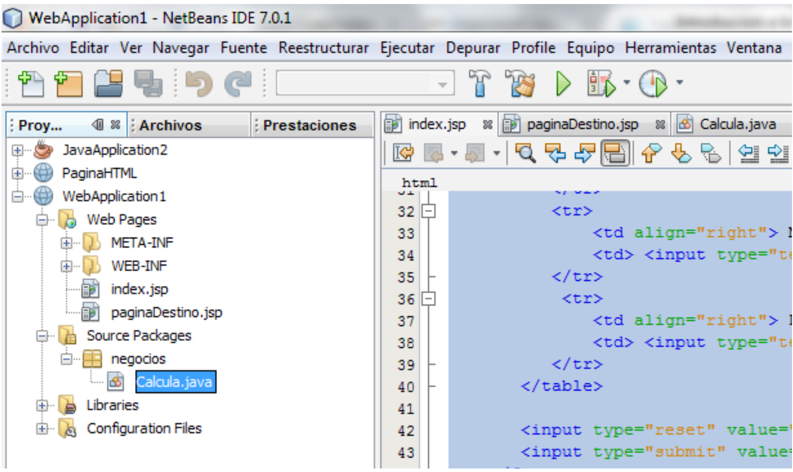


Figura IV-4. Las clases Javadeben ir en un paquete dentro de Source Packages

El código de `Calcula` es el siguiente:

```
package negocios;
public class Calcula {
    private Double tInicial;
    private Double tFinal;
    private Double distancia;
    private Double vel;
    private Double tTotal;

    public Calcula(String tIni, String tFin, String dist){
        tInicial = Double.parseDouble(tIni);
        tFinal = Double.parseDouble(tFin);
        distancia = Double.parseDouble(dist);
    }

    public void velocidad(){
        vel = distancia/(tFinal- tInicial);
    }

    public void tiempoTotal(){
        tTotal = tFinal-tInicial;
    }

    public Double getVel(){
        return vel;
    }

    public Double getTiempo(){
        return tTotal;
    }
}
```

Se convierten a `Double` las cadenas de caracteres recibidas

Los atributos son privados y hay métodos para acceder a ellos (*getters*)

La *paginaDestino.jsp* usa la clase `Calcula` para obtener los resultados de los cálculos y los despliega (Figura IV3 b), su código es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>Pagina Destino JSP</title>
    </head>
    <body>

       <%@ page import="negocios.Calcula" %>
       <%
           // Extracción de los parámetros recibidos
           String nombre = request.getParameter("nombre");
           String tInicial= request.getParameter("tiempoIni");
           String tFinal = request.getParameter("tiempoFin");
           String dist = request.getParameter("distancia");
           Double vel, tiempo;

           Calcula calcula = new Calcula(tInicial, tFinal, dist);
           calcula.velocidad();
           vel = calcula.getVel();
           calcula.tiempoTotal();
           tiempo=calcula.getTiempo();
        %>

        <h2> Resultado </h2>
        <p> Hola <%= nombre %> </p>
        <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
              <td align="right"> Tu tiempo total fué: </td>
              <td> <%= tiempo %> minutos </td>
          </tr>
          <tr>
              <td align="right"> Y tu velocidad: </td>
              <td> <%= vel %> metros/min</td>
          </tr>
         </table>

      <form action="index.jsp" method="post">
         <input type="submit" value="Regresar">
      </form>
    </body>
</html>
```

Para importar la clase `Calcula`

Obtención de resultados (los cálculos se hacen en la clase `Calcula`)

4.3.2 Escribir los datos en un archivo con una clase Java

Ahora trabajaremos con un archivo de texto desde una JSP utilizando una clase Java. En este ejercicio la página principal solicita al usuario su nombre, sus apellidos y su promedio, como en la Figura IV5 a).

a) *index.jsp* b) *paginaRestino.jsp*

Figura IV-5. Los datos que proporciona el usuario se registran en un archivo de texto

Cuando el usuario oprime el botón "*Enviar*" en la página de la Figura IV55 a), ésta envía los datos del usuario a la JSP de la Figura IV5 b), llamada *paginaRegistro.jsp*, quien se encarga de guardar los datos recibidos en un archivo llamado *Promedios.txt*. El código de la *paginaRegistro.jsp* manda guardar los datos a la clase `EscribeArchivo`, que se encuentra dentro del paquete `datos`. A continuación presentamos la clase `EscribeArchivo`

```
package datos;
import java.io.*;
import negocios.Alumno;
public class EscribeArchivo{
  public static void add(Alumno a, String path) throws IOException{
   File archivo;
   FileWriter fw=null;
   PrintWriter pw=null;
    try{
      archivo = new File(path);
      fw = new FileWriter(archivo, true);
      pw = new PrintWriter( fw );
      pw.println(a.getNombre()+","+a.getApellidos()+","
                                          +a.getPromedio() );

    }catch(Exception e){
      e.printStackTrace();
    }finally{
        try{
           if( pw != null)
               pw.close();
        } catch(Exception e2){
           e2.printStackTrace();}
    }
  }
}
```

La ruta del archivo donde se escriben los datos

Los campos se separan con coma para permitir espacios (varios nombres o apellidos)

Recordar que para escribir en un archivo de texto con Java, se utilizan los objetos `FileWriter` y `PrintWriter` que deben estar en un `try-catch-finally`

Si no existe el archivo que se proporciona en la *ruta* con la instrucción:

```
File archivo = new File(ruta);
```

entonces éste se crea automáticamente.
Cuando se crea el objeto de clase `FileWriter` con:

```
fw = new FileWriter(archivo, true);
```

Se da el valor *true* al parámetro *append* para indicar que cáda vez que se escriba en el archivo, se hará al final. Es decir, se agrega al final de lo que yá está escrito en el archivo. Cuando no se indica *true* en el constructor, entonces *append* está en *false* y cáda vez que se escribe algo nuevo se sobre-escribe lo que yá estaba, y se pierde la información anterior.

El método add de `EscribeArchivo` obtiene el nombre, apellidos y promedio del objeto de clase `Alumno`, por medio de los *getters*.

La clase `Alumno` (usada por `EscribeArchivo` y por *paginaRegistro.jsp*) se encuentra en el paquete `negocios`, y es la siguiente:

```
package negocios;

public class Alumno {
    private String nombre;
    private String apellidos;
    private Double promedio;

    public Alumno(String nombre, String apellidos,
                  Double promedio){
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.promedio = promedio;
    }

    public String getNombre(){
        return nombre;
    }

    public String getApellidos(){
        return apellidos;
    }

    public Double getPromedio(){
        return promedio;
    }

    public String toString(){
      return (nombre+" "+apellidos+" tiene promedio: "+promedio);
    }
}
```

Los atributos son privados y con los *getters* se tiene acceso a ellos

La *paginaRegistro.jsp* importa las clases `EnArchivo` y `Alumno`. En lugar de trabajar con cada uno de los atributos recibidos, éstos se agrupan en un objeto de clase `Alumno`. Nótese que los atributos que envía *index.jsp* son `String`, por lo tanto, hay que hacer la conversión de tipo cuando séa necesario. En este caso, `Alumno` requiere un `Double` en su constructor. A continuación presentamos el código de *paginaRegistro.jsp*.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>Página de Registro</title>
    </head>


    <body>
        <%@ page import="datos.EscribeArchivo, negocios.Alumno" %>
        <%
           // Obtención de los parámetros de la petición
           String nombre = request.getParameter("nombre");
           String apellidos = request.getParameter("apellidos");
           String promedio = request.getParameter("prom");

           Alumno alumno = new Alumno(nombre, apellidos,
                                Double.parseDouble(promedio));

           ServletContext sc = this.getServletContext();
           String path = sc.getRealPath("/WEB-INF/Promedios.txt");
           path = path.replace('\\','/');


           // Guardar en archivo
           EscribeArchivo.add(alumno, path);

        %>


        <h2>Tu registro se hizo con éxito</h2>

        <form action="index.jsp" method="post">
         <input type="submit" value="Nuevo Registro">
        </form>

    </body>
</html>
```

Los datos recibidos se guardan en un objeto de clase `Alumno`

Consultar la sección IV.1.3

`EnArchivo` es static, por lo que no se instancia

En una aplicación web, no se hace sólo una operación (como escribir en un archivo), sino que se requiere que el usuario pueda llevar a cabo diferentes operaciones, por ejemplo, registrarse, buscar un registro y ver todos los registros. Para hacer esto de una forma organizada, se usan *JSPs* para capturar y desplegar datos, clases para procesar los datos y *servlets* para controlar el flujo de datos entre las clases y las JSPs. En el siguiente capítulo estudiaremos qué son los *servlets* y como pasar información entre un *servlet* y una JSP y, entre un *servlet* y una clase Java.

4.4 Prácticas

4.4.1 Práctica 1

Hacer una aplicación Web que pida al usuario su nombre, su sueldo diario y la cantidad de días trabajados. Al seleccionar el botón "*Calcular sueldo*" debe calcular y desplegar el sueldo total. Usar JSPs y una clase Java para hacer el cálculo. La Figura IV-6 muestra un ejemplo de ejecución de esta aplicación:

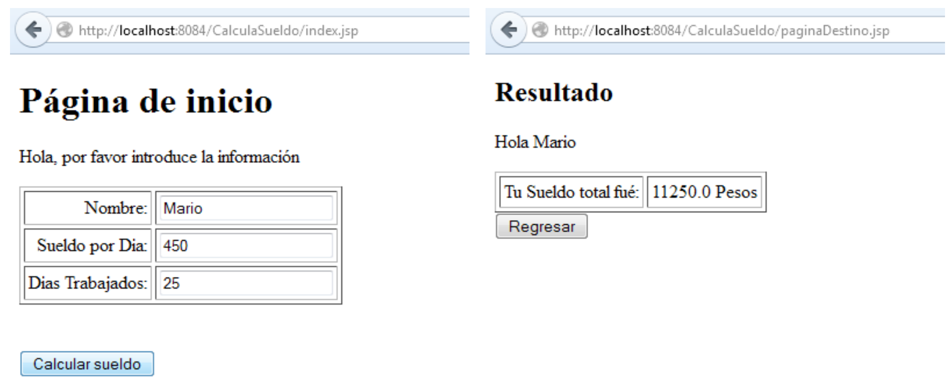


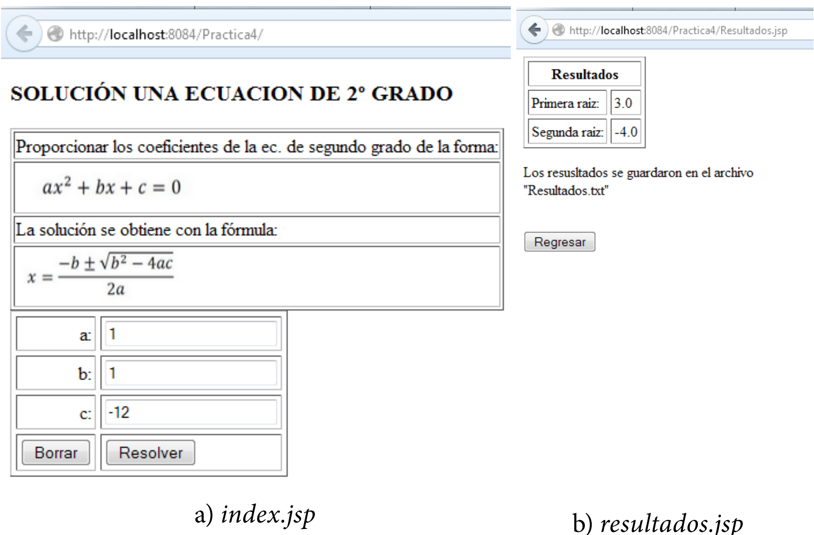
Figura IV-6. Cálculo del sueldo total

4.4.2 Práctica 2

Hacer una aplicación Web que resuelva una ecuación de segundo grado; los datos que debe proporcionar el usuario son los coeficientes *a*, *b* y *c* de la ecuación. Además, se deben guardar los resultados en un archivo de texto. La aplicación debe poder desplegar tanto raices reales, como raíces complejas. Usar una clase Java para resolver la ecuación. En *index.jsp* se incluyeron las siguientes imágenes:

$$ax^2 + bx + c = 0 \qquad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

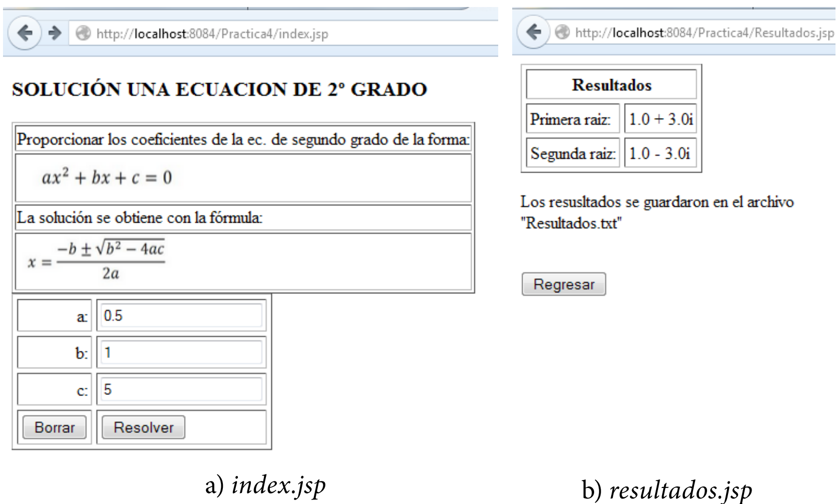
En la Figura IV7 se muestra un ejemplo de la práctica en la que la solución está dada por dos raices reales:



a) *index.jsp* b) *resultados.jsp*

Figura IV-7. Solución de una ecuación de segundo grado con raíces reales

En la Figura IV8 se muestra un ejemplo un ejemplo de la práctica en la que la solución está dada por dos raices complejas:



a) *index.jsp* b) *resultados.jsp*

Figura IV-8. Solución de una ecuación de segundo grado con raíces complejas

4.5 Solución a las prácticas

4.5.1 Solución de la práctica 1

A continuación, el código de *index.jsp*:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
              charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
       <h1> Página de inicio</h1>
    <p> Hola, por favor introduce la información </p>

    <form action="paginaDestino.jsp" method="post">
       <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
              <td align="right"> Nombre: </td>
              <td><input type="text" name="nombre"></td>
          </tr>
          <tr>
              <td align="right"> Sueldo por Dia: </td>
              <td> <input type="text" name="sueldoDia"> </td>
          </tr>
          <tr>
              <td align="right"> Dias Trabajados: </td>
              <td> <input type="text" name="diasTra"> </td>
          </tr>

       </table>
        <br>
        <br>

       <form action="paginaDestino.jsp" method="post">
         <input type="submit" value="Calcular sueldo">
      </form>
    </form>

    </body>
</html>
```

El código de la clase `Calcular` es el siguiente:

```
public class Calcular {
  private final Double sDiario;
    private final Double dTrabajo;
    private Double Stotal;

    public Calcular(String sueldoDia, String diasTrab){
        sDiario = Double.parseDouble(sueldoDia);
        dTrabajo = Double.parseDouble(diasTrab);
    }

    public void sueldototal(){
        Stotal = dTrabajo*sDiario;
    }

    public Double getSueldo(){
        return Stotal;
    }
}
```

Finalmente, la *paginaDestino.jsp*:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
       <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
       <title>Pagina Destino</title>
    </head>
    <body>
        <%@ page import="negocios.Calcular" %>
        <%
           // Extracción de los parámetros recibidos
           String nombre = request.getParameter("nombre");
           String sueldoDia= request.getParameter("sueldoDia");
           String diasTrab = request.getParameter("diasTra");
           Double Stotal;

           Calcular calcula = new Calcular(sueldoDia, diasTrab);
           calcula.sueldototal();
        %>

        <h2> Resultado </h2>
        <p> Hola <%= nombre %> </p>
        <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
              <td align="right"> Tu Sueldo total fué: </td>
              <td> <%= calcula.getSueldo() %> Pesos </td>
          </tr>
         </table>

      <form action="index.jsp" method="post">
         <input type="submit" value="Regresar">
      </form>
    </body>
</html>
```

4.5.2 Solución de la práctica 2

A continuación, el código de *index.jsp*:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
         <meta http-equiv="Content-Type"
                  content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h3>SOLUCIÓN UNA ECUACION DE 2° GRADO</h3>
        <table table border="1">
          <td> Proporcionar los coeficientes de la ec. de segundo
               grado de la forma:
          </td>
          <tr>
            <td>  <img src="ec grado 2.jpg"
                       width="170" heigth="170"
                      align="left"
                      alt="No se pudo mostrar la imagen">
            </td>

          </tr>
            <td> La solución se obtiene con la fórmula:
          </td>
          <tr>
            <td>  <img src="formula.jpg" width="150" heigth="150"
                       align="left"
                       alt="No se pudo mostrar la imagen">
            </td>
          </tr>
        </table>

       <form action="Resultados.jsp" method="POST">
        <table table cellspacing="4" cellpadding="3" border="1">

            <tbody>
              <tr>
                  <td align="right"> a: </td>
                  <td><input type="text" name="VarA"
                             value="" required/></td>
              </tr>
              <tr>
                  <td align="right"> b: </td>
                  <td><input type="text" name="VarB"
                             value="" required/></td>
              </tr>
              <tr>
                  <td align="right"> c: </td>
                  <td><input type="text" name="VarC"
                             value="" required/></td>
              </tr>
              <tr>
```

```
                    <td><input type="reset" value="Borrar"></td>
                    <td><input type="submit"
                               value="Resolver" /></td>
                </tr>
              </tbody>
          </table>
      </form>
    </body>
</html>
```

El código de la clase `ResuelveEc` es el siguiente:

```
package negocios;

import static java.lang.System.out;
public class ResuelveEc {
    private Double a;
    private Double b;
    private Double c;
    private Double discr;

     public ResuelveEc(String a1, String b1, String c2){
        a = Double.parseDouble(a1);
        b = Double.parseDouble(b1);
        c = Double.parseDouble(c2);
        discr= (Math.pow(b, 2) - (4 * a * c));
    }
    public String raiz1() {

        Double raiz;
        Double real, imaginaria;

        if (discr >= 0) {
            raiz = (-b + Math.sqrt(discr) )/( 2 * a);
            return String.valueOf(raiz);
        }else{
            real= b/( 2 * a);
            imaginaria = (Math.sqrt(-discr))/(2*a);
            return (real+" + "+imaginaria+"i");
        }
    }
    public String raiz2() {
        Double raiz;
        Double real, imaginaria;

        if (discr >= 0) {
            raiz = (-b - Math.sqrt(discr) )/( 2 * a);
            return String.valueOf(raiz);
        }else{
            real= b/( 2 * a);
            imaginaria = (Math.sqrt(-discr))/(2*a);
            return (real+" - "+imaginaria+"i");
        }
    }
}
```

El código de la clase `GestorArchivo` es el siguiente:

```
package negocios;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class GestorArchivo {

  File archivo;
  FileWriter fw;
  PrintWriter pw;
  public boolean guardarResultado(String r1, String r2,
                                String path)throws IOException {
      try {
            archivo = new File(path);
            fw = new FileWriter(archivo, true);
            pw = new PrintWriter(fw);
            pw.println("Raiz 1: "+r1 + "   Raiz 2:" + r2);
            return true;
      } catch (Exception e) {
            return false;
      } finally {
            fw.close();
    }
  }
```

Finalmente, la página *resultados.jsp*:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                 content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <%@ page import="negocios.ResuelveEc" %>
        <%@ page import="negocios.GestorArchivo" %>
       <%
          // Extracción de los parámetros recibidos
          String A = request.getParameter("VarA");
          String B = request.getParameter("VarB");
          String C = request.getParameter("VarC");

          ResuelveEc solucion = new ResuelveEc(A,B,C);

          String result1= solucion.raiz1();
          String result2=  solucion.raiz2();
```

```
           ServletContext sc = this.getServletContext();
           String path = sc.getRealPath("/WEB-INF/Resultados.txt");


           GestorArchivo guardar = new GestorArchivo();
           guardar.guardarResultado(result1, result2, path);
         %>
         <form action="" method="POST">
           <table table cellspacing="4" cellpadding="3" border="1">
               <tr align="center">
                   <th colspan="2">Resultados</th>
               </tr>
               <tr>
                   <td> Primera raiz: </td>
                   <td><%= result1%></td>
               </tr>
                <tr>
                   <td> Segunda raiz: </td>
                   <td><%= result2%></td>
               </tr>
           </table>
         </form>
         <p> Los resusltados se guardaron en el archivo  <br>
             "Resultados.txt"
         </p>
         <br>
        <form action="index.jsp" method="post">
           <input type="submit" value="Regresar">
        </form>

    </body>
</html>
```

5. El modelo de tres capas con JSP, servlets, y clases Java

5.1 Objetivos

- Estructurar una aplicación web con tres capas: Modelo, Vista y Controlador (MVC)
- Pasar correctamente la información entre cada una de las capas del MVC.

5.2 Los *servlets* y sus principales métodos

En este capítulo aprenderemos a utilizar las páginas JSP para establecer *la vista*, los *servlets* para construir *el control* y clases Java para conformar *el modelo* de una aplicación Web. Las JSP y las clases Java ya se estudiaron en capítulos anteriores. En esta sección estudiaremos los *servlets*, con los cuales se construyen *los controladores.* Un controlador recibe la información de *la vista*, pide a las clases del *modelo* que procesen la información y, posteriormente, manda desplegar los resultados a la vista correspondiente.

Un *servlet* es una clase Javaque hereda de la clase `HttpServlet`. Los cinco métodos más comunes de un servlet son:

```
public void init()throws ServletException{}

public void service(HttpServletRequest request,
                    HttpServletResponse response)
                    throws ServletException, IOException {

public void doPost (HttpServletRequest request,
                   HttpServletResponse response)
                   throws ServletException, IOException {

public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
                   throws ServletException, IOException {

public void destroy(){};
```

Observar que los métodos `service(), doPost()` y `doGet()` tienen como parámetros los objetos `request` y `response`. El objeto `request` contiene la información que se le envía al *servlet* para que éste se encargue de procesarla, y en el objeto `response` el *servlet* manda la información resultante.

En la Figura V1 se ilustra el ciclo de vida de un *servlet*. Cuando se llama al *servlet* por primera vez, es decir, se hace una petición, éste se instancia (se dice que "se carga") en la memoria del servidor y se invoca al método `init()` el cual se encarga de inicializarlo, posteriormente se invoca al método `service()`. En las siguientes peticiones, el *servlet* ya está cargado en la memoria, por lo que no es necesario invocar a `init()`. Entonces se llama directamente al `service()`, y éste llama a los demás métodos del *servlet*.

Todos los métodos en la Figura V1 ya están declarados en la clase abstracta `HttpServlet`, por lo que, para usarlos, es necesario sobre-escribirlos en el *servlet*.

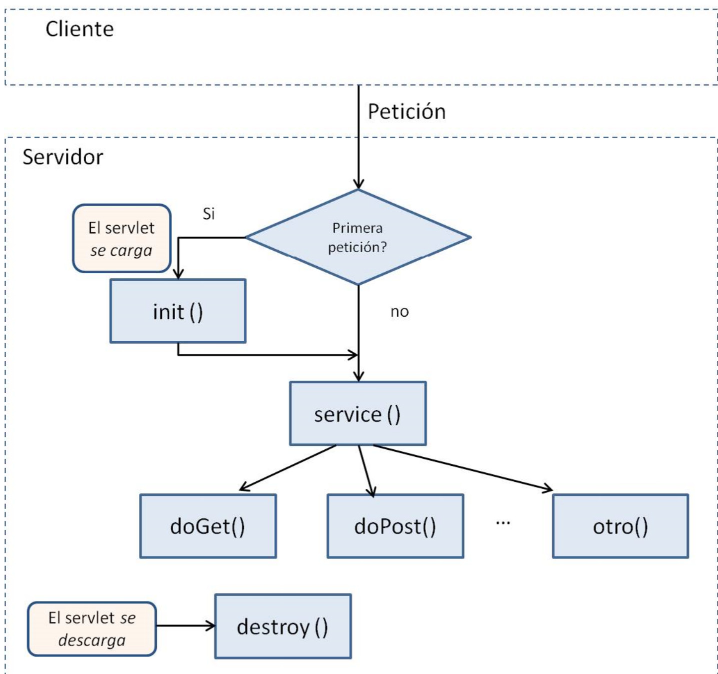


Figura V-1. Ciclo de vida de un servlet

Finalmente, cuando la aplicación finaliza, o el servidor falla, el *servlet* se descarga y se invoca al método `destroy()`:

5.3 Paso de información de una JSP a un *servlet*

Cuando se codifica en un *servlet*, lo aconsejable es sobre-escribir los métodos `doPost()` para atender a una petición POST y `doGet()` para atender a una petición GET, no se considera buena práctica sobre-escribir el `service()`.

En NetBeans se genera automáticamente el método `processRequest()` y los métodos `doGet()` y `doPost()` se sobre-escriben automáticamente para que se invoque a `process-Request()` así, para `doGet()`:

```
@Override
   protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
```

Y para el `doPost()`:

```
@Override
   protected void doPost(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
```

En este ejemplo, la página *index.jsp* de la Figura V2 a) solicita los datos al usuario. Al dar clic en "*Enviar*" se invoca al *servlet* `muestraRegistro` de la Figura V2 b):

a) *index.jsp* b) *muestraRegistro.jsp*

Figura V-2. Página JSP que pasa información a un servlet

index.jsp invoca al *servlet* `muestraRegistro.Java` por medio del `action`, como se muestra en su código a continuación:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Captura Datos</title>
    </head>
```

Se importa el paquete con el *servlet*

```
    <body>
      <%@ page import="controller.muestraRegistro" %>
      <h1>Hola!  Proporciona tus datos:</h1>

      <form action="muestraRegistro" method="post">
        <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
            <td align="right"> Nombre: </td>
            <td><input type="text" name="nombre"></td>
          </tr>
```

Para invocar a un *servlet* únicamente se pone su nombre en el *action*

```
          <tr>
            <td align="right"> Apellidos: </td>
            <td> <input type="text" name="apellidos"> </td>
          </tr>
          <tr>
            <td align="right"> Promedio: </td>
            <td> <input type="text" name="prom"> </td>
          </tr>
        </table>

        <input type="reset" value="Borrar">
        <input type="submit" value="Enviar">
      </form>
    </body>
</html>
```

El *servlet* `muestraRegistro.Java` recibe los atributos de *index.jsp* y construye un objeto de la clase `Alumno` (ver sección IV.2.2) y construye la página HTML de la Figura V2 b) donde muestra el nombre de la aplicación, y despliega los datos del alumno. Su código es el siguiente:

```
package controller;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import negocios.Alumno;

@WebServlet(name = "muestraRegistro", urlPatterns =
                                              {"/muestraRegistro"})
public class muestraRegistro extends HttpServlet {
```

```
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
                  throws ServletException, IOException{
     response.setContentType("text/html;charset=UTF-8");
     PrintWriter out = response.getWriter();

     try {
       String nombre= request.getParameter("nombre");
       String apellidos= request.getParameter("apellidos");
       Double promedio =
                  Double.parseDouble(request.getParameter("prom"));

       Alumno alumno = new Alumno(nombre,apellidos,promedio);

       //TODO output your page here
       out.println("<html>");
       out.println("<head>");
       out.println("<title>Servlet muestraRegistro</title>");
       out.println("</head>");
       out.println("<body>");
       out.println("<h1>Servlet muestraRegistro at " +
                  request.getContextPath () + "</h1>");
       out.println("<h2> Este es el Servlet muestraRegistro.java
                                                        </h2>");
       out.println("<h3> A continuación se muestran los parámetros
                                              recibidos </h3>");
       out.println(alumno);
       out.println("</body>");
       out.println("</html>");

       } finally {
          out.close();
     }
    }




    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);

    }
}
```

Los atributos se extraen de la misma forma que en una JSP

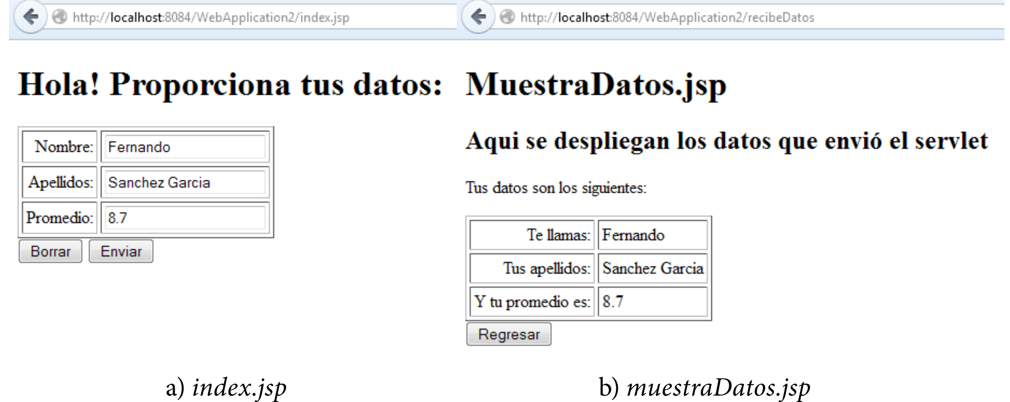
Recordar que automáticamente se invoca a `toString()` de `Alumno`

5.4 Paso de información de un *servlet* a una JSP

Aunque es posible desplegar una página HTML desde un *servlet*, como se ilustró en el ejemplo de la sección anterior, esto no es una buena práctica. Los *servlets* deben enviar la información a una JSP, quien se encargará de mostrar *la vista*, es decir, de desplegar la página para el usuario.

5.4.1 Transferencia de control con el objeto **`request`**

En el ejemplo de esta sección, los datos se capturan en el *index.jsp* de la Figura V3 a). Al dar clic en "Enviar", los datos capturados se pasan al servlet `recibeDatos.java`, el cual a su vez los envía a *muestraDatos.jsp* (Figura V3 b):



a) *index.jsp* b) *muestraDatos.jsp*

Figura V-3. index.jsp a), envía datos a un servlet, que a su vez los envía a la JSP de b).

El código de *index.jsp* es el mismo que el de la sección anterior, sólo cambia el nombre del *servlet* al cual se envía la petición POST: `action = recibeDatos.`

Las dos instrucciones que se usan para enviar datos de un *servlet* a una JSP son

```
   request.setAttribute("nombreAtributo",objetoAEnviar);
   request.getRequestDispatcher("nombre.jsp").forward(request,
response);
```

En el código del *servlet* `recibeDatos.Java` tenemos un ejemplo de su uso.

```
package controller;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import negocios.Alumno;

@WebServlet(name = "recibeDatos", urlPatterns = {"/recibeDatos"})
public class recibeDatos extends HttpServlet {

  protected void processRequest(HttpServletRequest request,
                              HttpServletResponse response)
            throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();

    try {
      String nombre= request.getParameter("nombre");
      String apellidos= request.getParameter("apellidos");
      Double promedio =
                 Double.parseDouble(request.getParameter("prom"));

      Alumno alumno = new Alumno(nombre,apellidos,promedio);

      request.setAttribute("atribAlumn",alumno);
      request.getRequestDispatcher(
                  "/muestraDatos.jsp").forward(request, response);

     } finally {
        out.close();
   }
 }
```

En el objeto `request` guardamos el objeto `alumno`. Utilizamos el atributo llamado `atribAlumn` para guardarlo.

Con esta instrucción se hace un reenvío (forward) de los objetos `request` y response a la JSP *muestraDatos*.

No se incluyen los otros métodos (`@Override`)

El objeto `request` tiene los siguientes métodos para enviar información:

`setAttribute(String` *nombre*, `Object o)` → Sirve para guardar un objeto en un atributo al que se le da un *nombre*.

`getAttribute(String` *nombre*`)` → Sirve recuperar el objeto guardado en el atributo *nombre*.

`getRequestDispatcher( String` *ruta*) → regresa un objeto `RequestDispatcher` para la ruta especificada.

`forward(request, response)` → re-envia los objetos `request` y `response` a otro recurso en el servidor, que normalmente es una JSP o un *servlet*.

Finalmente, *muestraDatos.jsp* recibe la información que le envía el *servlet* `recibeDatos.java`, por medio de un scriptlet, como se muestra en el siguiente código:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
      <meta http-equiv="Content-Type"
               content="text/html; charset=UTF-8">
      <title>JSP Page</title>
  </head>
  <body>
    <%@ page import="negocios.Alumno" %>

    <%
      Alumno alumno = (Alumno) request.getAttribute("atribAlumn");
    %>
    <h1> MuestraDatos.jsp</h1>
    <h2> Aqui se despliegan los datos que envió el servlet</h2>
    <p> Tus datos son los siguientes: </p>

    <table cellspacing="3" cellpadding="3" border="1" >
      <tr>
         <td align="right"> Te llamas: </td>
         <td> <%= alumno.getNombre() %> </td>
      </tr>
      <tr>
         <td align="right"> Tus apellidos: </td>
         <td> <%= alumno.getApellidos() %> </td>
       </tr>
       <tr>
         <td align="right"> Y tu promedio es: </td>
         <td> <%= alumno.getPromedio() %> </td>
       </tr>
    </table>

    <form action="index.jsp" method="post">
        <input type="submit" value="Regresar">
    </form>
  </body>
</html>
```

Recuperación del objeto enviado por el *servlet*

5.4.2 Transferencia de control con el objeto **`response`**

También existe una forma de transferir a otra URL, y es con el siguiente método del objeto `response`:

`sendRedirect(String` *ruta*`)` → se envía al cliente a la ruta especificada.

Este método no transfiere los objetos `request` y `response`, por lo tanto, sólo se utiliza para re-direccionar a una URL específica, usualmente fuera de la aplicación actual.

5.5 Comunicación entre *servlets* y clases Java

Los *servlets* constituyen *el controlador* de la aplicación, éstos utilizan las clases Java para procesar la información. Estas clases conforman *el modelo*. Una vez procesada la información, los *servlets* la mandan desplegar a *la vista*.

El intercambio de información entre *servlets* y clases Javaes es trivial. Sólo es necesario importar en el *servlet*, la o las clases con las que trabaja. Para enviar información a una clase, se instancia un objeto en el *servlet* y se envían los parámetros en el método constructor. El resultado del procesamiento se obtiene con los métodos de la clase diseñados para proporcionar datos y resultados. Un ejemplo claro es el *servlet* `muestraRegistro.Java` de la sección V.2.

En el ejemplo de esta sección, modificaremos el proyecto anterior de tal forma que los datos que introduce el usuario se guarden en un archivo llamado *promedios.txt*, cuando el usuario haga clic en el botón "*Registrar*". Después de que los datos se guardan en el archivo, se despliega la página *muestraDatos.jsp* con los datos que se guardaron, como en la Figura V4 b).

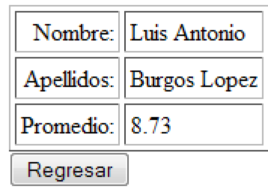
Hola! Proporciona tus datos:

Nombre:	Luis Antonio
Apellidos:	Burgos Lopez
Promedio:	8.73

Borrar Registrar
Ver Alumnos

MuestraDatos.jsp

Se guardaron los siguientes datos:



Nombre:	Luis Antonio
Apellidos:	Burgos Lopez
Promedio:	8.73

Regresar

a) *index.jsp* b) *muestraDatos.jsp*

Figura V-4. Registro de los datos del usuario en un archivo

Para procesar el registro, codificaremos el *servlet* `recibeDatos.java`, el cual recibe los datos proporcionados por *index.jsp* y encarga a la clase `EscribeArchivo.Java` que los guarde en el archivo. Los datos que se guardaron se despliegan en *muestraDatos.jsp*. El código de `recibeDatos.Java` es el siguiente:

```
package controller;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import negocios.*;

@WebServlet(name = "recibeDatos", urlPatterns = {"/recibeDatos"})
public class recibeDatos extends HttpServlet {

  protected void processRequest(HttpServletRequest request,
                              HttpServletResponse response)
            throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
         String nombre= request.getParameter("nombre");
         String apellidos= request.getParameter("apellidos");
         Double promedio =
                 Double.parseDouble(request.getParameter("prom"));

         Alumno alumno = new Alumno(nombre,apellidos,promedio);

         ServletContext sc = this.getServletContext();
         String path = sc.getRealPath("/WEB-INF/Promedios.txt");
         path = path.replace('\\','/');

           // Guardar en archivo
         EscribeArchivo.add(alumno, path);

         request.setAttribute("atribAlumn",alumno);
         request.getRequestDispatcher("/muestraDatos.jsp")
                                      .forward(request, response);
        } finally {
            out.close(); }
    }

    @Override
    protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
}
```

Se importan las clases que se usan (en el paquete negocios)

Clases usadas en este *servlet*

Se envía el objeto `alumno` a *muestraDatos.jsp*

`recibeDatos.Java` hace uso de las clases `Alumno.Java` y `EscribeArchivo.Java` de la sección IV.2.2.

muestraDatos.jsp tiene el siguiente código:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type"
             content="text/html; charset=UTF-8">
    <title>Muestra Datos</title>
  </head>
  <body>
    <%@ page import="negocios.Alumno" %>
    <%
      Alumno alumno = (Alumno) request.getAttribute("atribAlumn");
    %>

    <h3> MuestraDatos.jsp</h3>

    <p> Se guardaron los siguientes datos: </p>

    <table cellspacing="3" cellpadding="3" border="1" >
      <tr>
         <td align="right"> Nombre: </td>
         <td> <%= alumno.getNombre() %> </td>
      </tr>
      <tr>
         <td align="right"> Apellidos: </td>
         <td> <%= alumno.getApellidos() %> </td>
       </tr>
       <tr>
         <td align="right"> Promedio: </td>
         <td> <%= alumno.getPromedio() %> </td>
       </tr>
    </table>

    <form action="index.jsp" method="post">
        <input type="submit" value="Regresar">
    </form>

  </body>
</html>
```

Ahora agregaremos funcionalidad al botón “Ver alumnos”, de index.jsp (ver Figura V4 a) con el cual se mostrarán todos los alumnos registrados en promedios.txt. Con este botón se invoca al servlet `muestraRegistro.java`, el cual pide a la clase `LeeArchivo.Java` que lea los datos del archivo solicitado y los capture en un `ArrayList`. Posteriormente pasa esta información a alumnosRegistrados.jsp, la cual despliega todos los registros recibidos. Como se muestra en la Figura V-5:

Los alumnos que están registrados son:

Nombre	Apellidos	Promedio
Miriam	Fernadez Olvera	7.9
Andres	Sanchez Garcia	9.3
Luis Antonio	Burgos Lopez	8.73

Numero de registros : 3

Regresar

alumnosRegistrados.jsp

Figura V-5. Página que despliega todos los registros leidos en «promedios.txt

El código del *servlet* `muestraRegistro.Java` se muestra a continuación. Cabe observar que la clase `LeeArchivo` es estática, por lo que no se instancia. Se llama al método `clearCont()` para reinicializar el contador a 0. Como la variable `cont` es estática, conserva el valor que tenía cuando se ejecutó el *servlet* con anterioridad. La primera vez que se ejecuta la aplicación: `cont = 0`, pero una vez que ya se leyó el archivo, `cont` tiene el número de registros que hay en el archivo. Si `cont` no se reinicializa a cero, cuando solicitamos leer el archivo una vez más, el número de registros se sumará al valor que ya tenía `cont` anteriormente.

```
package controller;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import negocios.Alumno;
import negocios.LeeArchivo;

@WebServlet(name = "muestraRegistro",
            urlPatterns = {"/muestraRegistro"})
public class muestraRegistro extends HttpServlet {
```

```
  protected void processRequest(HttpServletRequest request,
                                HttpServletResponse response)
                 throws ServletException, IOException {
     response.setContentType("text/html;charset=UTF-8");
     PrintWriter out = response.getWriter();
     try {
         ArrayList <Alumno> alumnos = new ArrayList<Alumno>();
         int cont=0;
         String contador;

         ServletContext sc = this.getServletContext();
         String path = sc.getRealPath("/WEB-INF/Promedios.txt");
         path = path.replace('\\','/');
         alumnos = LeeAlumno.leeAlumnos(path);

         // Resetea la variable estática
         LeeArchivo.clearCont();
         cont = LeeArchivo.getCont();
         contador= String.valueOf(cont);

         request.setAttribute("Alumnos",alumnos);
         request.setAttribute("contador", contador);

         request.getRequestDispatcher("/alumnosRegistrados.jsp")
                                 .forward(request, response);


        } finally {
            out.close();
        }
    }
}
```

Guarda el `ArrayList` en el atributo `Alumnos`

También guarda el contador

Se envían los datos a *alumnosRegistrados.jsp*

La clase para leer de un archivo de texto `LeeArchivo.Java` es la siguiente:

```
package negocios;
import java.io.*;
import java.util.ArrayList;

public class LeeArchivo {
  private static  int cont = 0;
  private static File archivo;
  private static FileReader fr;
  private static BufferedReader br;

  static ArrayList <Alumno> alumnos = new ArrayList<Alumno>();
```

```
  public static ArrayList <Alumno> leeAlumnos(String path){
    try {
      archivo = new File(path);
      fr = new FileReader(archivo);
      br = new BufferedReader(fr);
      String linea=null;
      String [] tokensLinea = null;
      String nombre;
      String apellido;
      Double promedio;
      Alumno alumno;
      linea=br.readLine();
      while( linea!=null){
          tokensLinea = linea.split(",");
          nombre =tokensLinea[0];
          apellido=tokensLinea[1];
          promedio= Double.parseDouble(tokensLinea[2]);
          alumno = new Alumno(nombre, apellido, promedio);
          alumnos.add(alumno);
          cont++;
          linea=br.readLine();
        }
    }catch(Exception e){
           e.printStackTrace();
    }finally{
       try{
          if( null != fr )
              fr.close();
       }catch (Exception e2){
            e2.printStackTrace();}
    }
    return alumnos;
  }

  public static int getCont(){
      return cont;
  }

  public static void clearCont(){
      cont=0;
  }
}
```

Se lee una línea completa, luego se separan los *tokens* y se guardan en un arreglo de *Strings*

Cuando se escribió en el archivo los campos se separaron con coma (permite varios nombres o apellidos)

`cont` cuenta el número de registros leídos. Nótese que cuando el *servlet* pasa el contador a *alumnosRegistrados.jsp*, lo pasa como `String` (no como un entero), por lo que hay que hacer la conversión correspondiente.

La aplicación Web de este ejemplo está organizada con el modelo de tres capas MVC (Modelo Vista Controlador). En la Figura V6 observamos que las páginas JSP forman *la vista* de la aplicación ya que son la interfaz con el usuario, éstas se encargan de capturar y mostrar datos. Los *servlets*: `recibeDatos.Java`y `muestraRegistro.Java` funcionan como *controladores*. Éstos hacen uso de las clases que están en el modelo para procesar los datos recibidos

y transferir el control a una JSP. Y el *modelo* está formado por las clases `EscribeArchivo.java`, `Alumno.Java` y `LeeArchivo.java`.

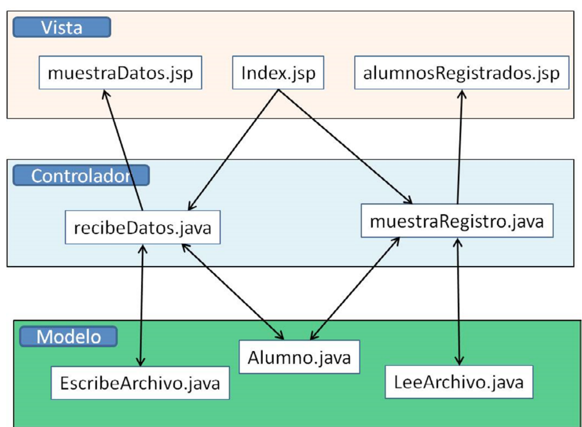


Figura V-6. Modelo de tres capas MVC

A continuación, presentamos el código de la página *alumnosRegistrados.jsp* a la cual el *servlet* `muestraRegistro.Java` transfiere el control cuando ha terminado de leer los datos del archivo.

En este caso en particular, usamos un `ArrayList` para guardar los objetos de la clase `Alumno` leídos en el archivo. Sin embargo, puede usarse también un `LinkedList`.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
      <meta http-equiv="Content-Type"
               content="text/html; charset=UTF-8">
      <title>Alumnos Registrados</title>
  </head>
  <body>
     <%@ page import="negocios.Alumno, java.util.ArrayList" %>
      <h2>Los alumnos que están registrados son: </h2>
      <%
         ArrayList<Alumno> alumnos = null;
         alumnos =
             (ArrayList<Alumno>)request.getAttribute("Alumnos");
         String numReg= (String) request.getAttribute("contador");
         int numRegistros = Integer.parseInt(numReg);
      %>
```

```
    <table border="1">
       <tr>
          <th>Nombre</th>
          <th>Apellidos</th>
          <th>Promedio</th>
       </tr>
      <%
         for (Alumno alumno: alumnos)
         {
      %>
      <tr valign="top">
         <td><%=alumno.getNombre()  %></td>
         <td><%=alumno.getApellidos() %></td>
         <td><%=alumno.getPromedio() %></td>

      </tr>

      <% } %>
      <% alumnos.clear();%>
      </table>

     <p> Numero de registros : <%= numRegistros %></p>
     <br>
     <form action="index.jsp" method="post">
         <input type="submit" value="Regresar">
     </form>

  </body>
</html>
```

Este scriptlet hace una iteración por cada objeto `alumno` de clase `Alumno` contenido en el `ArrayList alumnos`

Borra el contenido de `alumnos`. Si no lo hacemos, entonces cada vez que se solicita la JSP se agrega la nueva lectura a la que se tenía anteriormente.

La organización del proyecto en NetBeans se ilustra en la Figura V7. La *vista* (páginas JSP) siempre van en la carpeta WEB-Pages. Los *servlets* y clases Java deben estar agrupados en paquetes dentro de la carpeta Source Packages. Normalmente, se le llama "*controller*" al paquete en donde se encuentran los *servlets*, ya que es en éste en donde se llevan a cabo las funciones del controlador. Al paquete en donde se realizan las funciones del modelo se le suele llamar "*modelo*"(model) o también "*negocios*" (business).

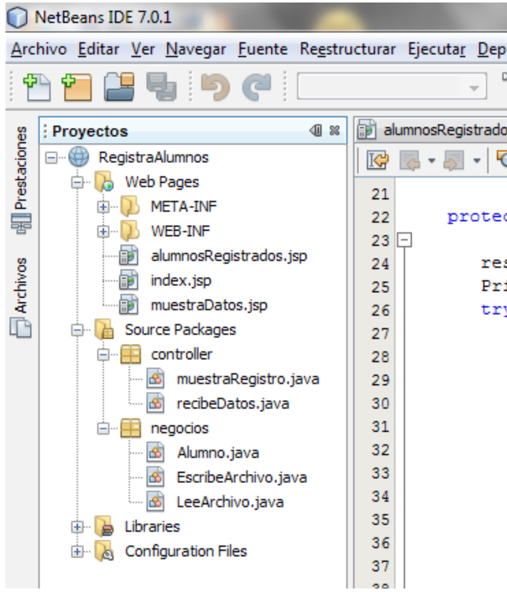


Figura V-7. Estructura del proyecto en NetBeans

5.6 Práctica

5.6.1 Planteamiento

Hacer una aplicación Web que haga lo siguiente:

a) En la Figura V8 a) se muestra la página principal de la aplicación (*index.jsp*). Cuando el usuario introduce los datos de un producto y da clic en el botón “Registrar”, los datos se envían a un *servlet* llamado `registraProducto.java`, este servlet pide a la clase `guardaProducto.Java` que guarde los datos en un archivo llamado *productos.txt*. Finalmente, `registraProducto.Java` envía los datos que se registraron a la página *muestraRegistro.jsp* como se muestra en la Figura V8 b). La página *muestraRegistro.jsp* tiene un botón para regresar a la página principal.

a) *index.jsp* b) *muestraRegistro.jsp*

Figura V-8. Registro de un producto

b) Cuando el usuario oprime el botón "Mostrar Registros" en la página principal de la aplicación (index.jsp), se invoca al *servlet* `muestraProductos.java`. Este *servlet* pide a la clase `leeProductos.Java` que lea todos los productos que están guardados en el archivo *productos.txt*. El *servlet* `muestraProductos.Java` envía un `arrayList` con todos los productos del archivo a la página *despliegaProductos.jsp* como en la Figura V9:

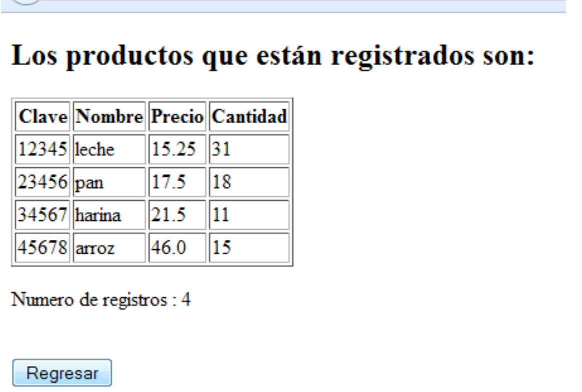


Figura V-9. despliegaProductos.jsp muestra todos los productos registrados en el archivo productos.txt

5.6.2 Solución a la práctica

5.6.2.1 a) Registro de un producto

Primero presentamos la clase Producto, que se encuentra en el paquete *negocios*.

```
package negocios;

public class Producto {
    private int clave;
    private String nombre;
    private Double precio;
    private int cantidad;

    public Producto(int clave, String nombre,
                    Double precio,int cant){
        this.clave = clave;
        this.nombre = nombre;
        this.precio = precio;
        this.cantidad = cant;
    }

    public int getClave(){
        return clave;
    }

    public String getNombre(){
        return nombre;
    }

    public Double getPrecio(){
        return precio;
    }

    public int getCantidad(){
        return cantidad;
    }
}
```

El código de *index.jsp* es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                 content="text/html; charset=UTF-8">
        <title> Index </title>
    </head>
    <body>
      <%@ page import="controller.registraProducto" %>
      <h2>Hola!  Proporciona los datos </h2>
      <h2> del producto</h2>
      <form action="registraProducto" method="post">
        <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
            <td align="right"> Clave: </td>
            <td><input type="text" name="clave"></td>
          </tr>
          <tr>
```

```
            <td align="right"> Nombre: </td>
            <td><input type="text" name="nombre"></td>
          </tr>
          <tr>
            <td align="right"> Precio: </td>
            <td> <input type="text" name="precio"> </td>
          </tr>
          <tr>
            <td align="right"> Cantidad: </td>
            <td> <input type="text" name="cant"> </td>
          </tr>
        </table>

        <input type="reset" value="Borrar">
        <input type="submit" value="Registrar">
      </form>

      <form action="muestraProductos"  method="POST">
          <input type="submit" value="Ver Productos">
      </form>
    </body>
</html>
```

El código del *servlet* `registraProducto.Java` es el siguiente:

```
package controller;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletContext;
import negocios.*;


@WebServlet(name = "registraProducto",
                    urlPatterns = {"/registraProducto"})
public class registraProducto extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
            throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      PrintWriter out = response.getWriter();
      try {
         int clave=
                  Integer.parseInt(request.getParameter("clave"));
         String nombre= request.getParameter("nombre");
         Double precio=
               Double.parseDouble(request.getParameter("precio"));
         int cantidad=
                   Integer.parseInt(request.getParameter("cant"));
```

```
                    Integer.parseInt(request.getParameter("cant"));
          Producto producto =
                     new Producto(clave, nombre, precio, cantidad);

          ServletContext sc = this.getServletContext();
          String path = sc.getRealPath("/WEB-INF/Productos.txt");
          path = path.replace('\\','/');

            // Guardar en archivo
          GuardaProducto.add(producto, path);

          request.setAttribute("atribProd",producto);
          request.getRequestDispatcher("/muestraRegistro.jsp")
                                    .forward(request, response);
         } finally {
             out.close();
         }
     }
}
```

El código de la clase `guardaProducto.Java` es la siguiente:

```
package negocios;
import java.io.*;

public class GuardaProducto {
 public static void add(Producto p, String path) throws IOException{
   File archivo;
   FileWriter fw=null;
   PrintWriter pw=null;
    try{
      //archivo = new File(ruta);
      archivo = new File(path);
      fw = new FileWriter(archivo, true);
      pw = new PrintWriter( fw );
      pw.println(p.getClave()+","+p.getNombre()+
                 ","+p.getPrecio()+","+p.getCantidad() );
    }catch(Exception e){
      e.printStackTrace();
    }finally{
        try{
           if( pw != null)
               pw.close();
        } catch(Exception e2){
           e2.printStackTrace();
        }
    }

  }
}
```

El código de *muestraRegistro.jsp* es el siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                 content="text/html; charset=UTF-8">
        <title>Muestra Registro</title>
    </head>
    <body>
        <%@ page import="negocios.Producto" %>
    <%
      Producto producto =
                     (Producto) request.getAttribute("atribProd");
    %>

    <h3> MuestraRegistro.jsp</h3>

    <p> Se guardaron los siguientes datos: </p>

    <table cellspacing="3" cellpadding="3" border="1" >
      <tr>
         <td align="right"> Clave: </td>
         <td> <%= producto.getClave() %> </td>
      </tr>
      <tr>
         <td align="right"> Nombre: </td>
         <td> <%= producto.getNombre() %> </td>
      </tr>
      <tr>
         <td align="right"> precio: </td>
         <td> <%= producto.getPrecio() %> </td>
       </tr>
       <tr>
         <td align="right"> Cantidad: </td>
         <td> <%= producto.getCantidad() %> </td>
       </tr>
    </table>

    <form action="index.jsp" method="post">
        <input type="submit" value="Regresar">
    </form>

    </body>
</html>
```

5.6.2.2 b) Desplegar productos

El código del *servlet* `muestraProductos.Java` es el siguiente:

```
package controller;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletContext;
import java.util.ArrayList;
import negocios.Producto;
import negocios.LeeProductos;

@WebServlet(name = "muestraProductos",
            urlPatterns = {"/muestraProductos"})
public class muestraProductos extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
           ArrayList <Producto> productos =
                                        new ArrayList<Producto>();
           int cont=0;
           String contador;
           Producto producto;
           ServletContext sc = this.getServletContext();
           String path = sc.getRealPath("/WEB-INF/Productos.txt");
           path = path.replace('\\','/');
           LeeProductos.clearCont();
           productos = LeeProductos.leeProductos(path);
           cont = LeeProductos.getCont();
           contador= String.valueOf(cont);

           request.setAttribute("Productos",productos);
           request.setAttribute("contador", contador);
           request.getRequestDispatcher("/despliegaProductos.jsp")
                                      .forward(request, response);


        } finally {
            out.close();
        }
    }
}
```

El código de la clase `LeeProductos.Java` es el siguiente:

```
package negocios;
import java.io.*;
import java.util.ArrayList;
public class LeeProductos {
  private static  int cont = 0;
  private static File archivo;
  private static FileReader fr;
  private static BufferedReader br;

  static ArrayList <Producto> productos =
                                     new ArrayList<Producto>();
    public static ArrayList <Producto> leeProductos(String path){

    try {
      archivo = new File(path);
      fr = new FileReader(archivo);
      br = new BufferedReader(fr);
      String linea=null;
      String [] tokensLinea = null;
      int clave;
      String nombre;
      Double precio;
      int cant;
      Producto producto;
      linea=br.readLine();
      while( linea!=null){
          tokensLinea = linea.split(",");
          clave= Integer.parseInt(tokensLinea[0]);
          nombre =tokensLinea[1];
          precio= Double.parseDouble(tokensLinea[2]);
          cant = Integer.parseInt(tokensLinea[3]);
          producto = new Producto(clave,nombre, precio, cant);
          productos.add(producto);
          cont++;
          linea=br.readLine();
        }
    }catch(Exception e){
           e.printStackTrace();
    }finally{
       try{
          if( null != fr )
              fr.close();
       }catch (Exception e2){
            e2.printStackTrace();}
    }
    return productos;
  }
  public static int getCont(){
      return cont;
  }
  public static void clearCont(){
      cont=0;
  }
}
```

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
         <meta http-equiv="Content-Type"
                  content="text/html; charset=UTF-8">
        <title>Despliega Productos</title>
    </head>
    <body>
      <%@ page import="negocios.Producto, java.util.ArrayList" %>

      <h2>Los productos que están registrados son: </h2>


      <%
         ArrayList<Producto> productos = null;
         productos =
           (ArrayList<Producto>)request.getAttribute("Productos");
      %>

      <table border="1">
         <tr>
            <th>Clave</th>
            <th>Nombre</th>
            <th>Precio</th>
            <th>Cantidad</th>
         </tr>

        <%
           for (Producto producto: productos)
           {
        %>
         <tr valign="top">
           <td><%=producto.getClave()  %></td>
           <td><%=producto.getNombre()  %></td>
           <td><%=producto.getPrecio() %></td>
           <td><%=producto.getCantidad() %></td>

         </tr>

         <% } %>

      </table>
      <p> Numero de registros : <%= productos.size() %></p>
      <% productos.clear();%>
      <br>
      <form action="index.jsp" method="post">
        <input type="submit" value="Regresar">
      </form>
    </body>
</html>
```

6. Acceso a base de datos

6.1 Objetivo

Construir aplicaciones Web que se conecten con una base de datos para recuperar, guardar y modificar la información.

6.2 Breve repaso de bases de datos

Una base de datos relacional es un conjunto de información relacionada que está estructurada en tablas. Cada tabla contiene varias filas, cada fila está formada por columnas. Se asume que el lector ya está capacitado en el tema de las bases de datos, por lo que en esta sección sólo se presenta un resumen a manera de recordatorio.

Antes de trabajar con una base de datos es necesario crearla y crear sus tablas mediante un gestor de bases de datos. En este curso trabajamos con MySQL porque es un sistema de gestión de base de datos muy aceptado y ampliamente utilizado. Si aún no se tiene instalado MySQL, éste se puede bajar gratuitamente de http://www.mysql.com/downloads/

Para iniciar una sesión de MySQL se puede utilizar la herramienta MySQL línea de Comando (Command Line), a la cual únicamente se le proporciona el password, como en la Figura VI1. Ésta es la forma que utilizaremos en este curso.

El código de *despliegaProductos.jsp* es el siguiente:

```
MySQL 5.5 Command Line Client
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.5.27 MySQL Community Server (GPL)

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> _
```

Figura VI-1. MySQL Command Line

La otra forma de trabajar con MySQL es mediante las herramientas de interfaz gráfica (MySQL GUI tools).

6.2.1 Creación de una base de datos: CREATE DATABASE

Antes de comenzar a trabajar con una base de datos es necesario crearla, esto se hace con el comando:

MySQL> CREATE DATABASE nombre_Base_Datos;

Como ejemplo crearemos la base de datos llamada escuela.

MySQL> CREATE DATABASE escuela;

El comando CREATE no es suficiente, una vez que se crea la base de datos, hay que "usarla" con el comando USE. Entonces, para poder trabajar con ella, es necesario el comando:

MySQL> USE nombre_Base_Datos;

en nuestro caso:

MySQL> USE escuela;

Cuando se trabajen sesiones posteriores, no habrá que crear la base de datos nuevamente, bastará con "usarla" con el comando USE. Nótese, que si se crea la base de datos con CREATE y después no se utiliza el comando USE, será imposible empezar a trabajar con ella.

Leer la base de datos.- Para saber en que base de datos estamos trabajando se utiliza la función DATABASE(), en nuestro caso, como se muestra en la Figura VI2 la base de datos se llama "escuela":

```
MySQL 5.5 Command Line Client
mysql> create database escuela;
Query OK, 1 row affected (0.01 sec)

mysql> use escuela;
Database changed
mysql> select database();
+------------+
| database() |
+------------+
| escuela    |
+------------+
1 row in set (0.00 sec)
```

Figura VI-2. Creación y uso de la base de datos "escuela"

6.2.2 Comandos para crear y modificar tablas

6.2.2.1 Creación de una tabla: CREATE TABLE

Una vez que nuestra base de datos está lista para trabajar, podemos *crear una tabla dentro de esta base de datos* con el comando CREATE y proporcionamos los campos correspondientes:

```
mysql>CREATE TABLE alumnos(nombre      VARCHAR(20),
                           apellidos   VARCHAR(30),
                           promedio    DOUBLE(4,2));
```

Para consultar todas las tablas que tenemos en una base de datos, podemos usar el comando SHOW TABLES. Para consultar los campos de una tabla, usamos el comando DESCRIBE nombre_tabla, esto es útil para acordarnos del nombre, del orden o el tipo de dato de los campos de la tabla, como se muestra en la Figura VI3:

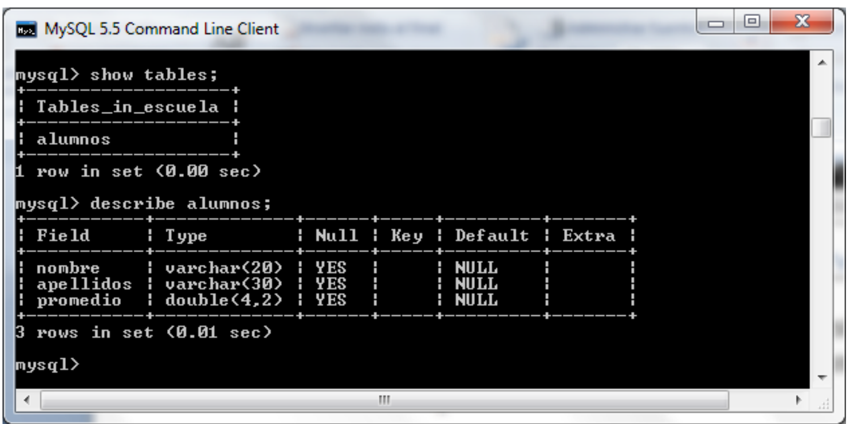


Figura VI-3. Consulta de las tablas en la base de datos y de los campos de una tabla

6.2.2.2 Modificación de la estructura de una tabla: ALTER TABLE.

Borrar una columna de la tabla.

Para borrar una columna de una tabla se usa el comando **ALTER TABLE** añadiendo la cláusula **DROP**. Por ejemplo, si tenemos la tabla **t2** y necesitamos borrar la columna **c**, entonces usamos el comando **ALTER TABLE** con la cláusula **DROP**, como se muestra a continuación:

```
mysql> ALTER TABLE t2 DROP COLUMN c;
```

La palabra COLUMN es opcional y puede omitirse:

```
mysql> ALTER TABLE t2 DROP c;
```

Para agregar una columna de la tabla.

Para agregar una columna a una tabla se usa el comando **ALTER TABLE** añadiendo la cláusula **ADD**. Por ejemplo, si tenemos la tabla **t2** y necesitamos agregarle la columna **d**, de tipo TIME, entonces usamos el comando **ALTER TABLE** con la cláusula **ADD**, como se muestra a continuación:

```
mysql> ALTER TABLE t2 ADD d TIME;
```

Para modificar una columna de la tabla.

Para hacer modificaciones a la columna de una tabla se usa el comando **ALTER TABLE** añadiendo las cláusulas **CHANGE** y/o **MODIFY** según sea el caso.

Para ***cambiar el tipo de dato*** de una columna de la tabla **t2** para que en lugar de **INT** sea **DOUBLE** dejando el mismo nombre (calific), utilizamos la cláusula **MODIFY**.

```
mysql> ALTER TABLE t2 MODIFY calific DOUBLE;
```

Para ***cambiar el nombre del campo*** de una tabla **t2** utilizamos la cláusula **CHANGE**. Por ejemplo, si queremos renombrar la columna de **b** a **d**:

```
mysql> ALTER TABLE t2 CHANGE b d;
```

También es posible cambiar el nombre de la columna al mismo tiempo que su tipo de datos, por ejemplo:

```
mysql> ALTER TABLE t2 CHANGE b  d VARCHAR(20);
```

Modifica la columna **b** para que se llame **d** y además cambia su tipo a VARCHAR(20).

Para agregar una columna a una tabla.

Para agregar una columna a una tabla, usamos el siguiente comando:

```
mysql> ALTER TABLE nombre_tabla ADD nombre_columna  tipo_dato;
```

Por ejemplo:

```
mysql> ALTER TABLE t2 ADD e DATE;
```

6.2.2.3 Renombrar una tabla: ALTER TABLE…RENAME.

Para renombrar una tabla se usa el comando **ALTER TABLE**. Por ejemplo, para renombrar una tabla de **t1** a **t2** usamos:

```
mysql> ALTER TABLE t1 RENAME t2;
```

6.2.2.4 Borrar una tabla: DROP.

Para borrar una tabla de la base de datos:

```
MySQL> DROP TABLE nombre_tabla;
```

6.2.3 Comando par *consultar datos* en una tabla: SELECT

Para consultar todos los datos de una tabla usamos SELECT. La sintaxis de este comando es la siguiente:

```
MySQL> SELECT  *
              FROM nombreTabla
              [WHERE criterio-de-selección]
              [ORDER BY campo1 [ASCIDESC],
                        campo2 [ASCIDESC],….]  ;
```

La forma más sencilla de este comando es:

```
MySQL> SELECT  * FROM nombreTabla;
```

el cual despliega todos los registros de la tabla. Cuando la tabla está vacía se despliega el mensaje "*empty set*", como se observa en la Figura VI4. En esta figura también se puede ver la *inserción* de tres registros (ver VI.2.4.1). Después, con SELECT * se despliegan todos los registros de la tabla.

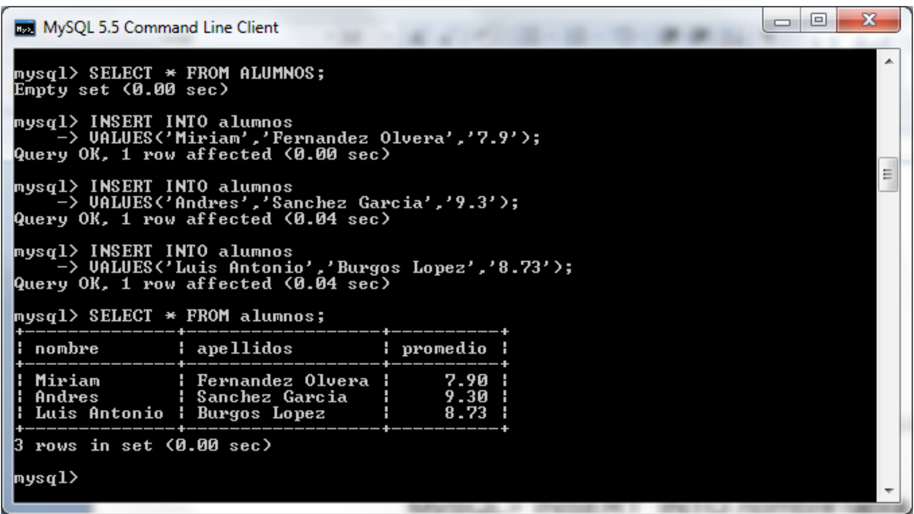


Figura VI-4. Inserción de registros en una tabla y desplegado de todos los datos

El comando SELECT regresa un conjunto de resultados conocido como *result set* (o tabla de resultados).

6.2.4 Comandos para *modificar datos* en una tabla

6.2.4.1 Insertar datos en una tabla: INSERT

Para insertar un registro en una tabla usamos el comando INSERT con la siguiente sintaxis:

```
MySQL> INSERT  INTO nombreTabla
          ->VALUES ('valor del campo1', 'valor del campo2,…,
                    'valor del campoN' );
```

6.2.4.2 Actualizar datos en una tabla: UPDATE

UPDATE, modifica datos uno o varios campos según el criterio de selección.

```
MySQL> UPDATE nombreTabla
          ->SET expresion1 [, expresion2,...]
            WHERE criterio-de-selección
```

6.2.4.3 Borrar datos en una tabla: UPDATE

DELETE, borra renglones de una tabla según el criterio de selección.

```
MySQL> DELETE FROM nombreTabla
       ->WHERE criterio-de-selección
```

6.3 Conexión con la base de datos

La conexión de una aplicación Web con una base de datos es de suma importancia, ya que en la mayoría de los sistemas se requiere operar con la información de una base de datos. Para establecer la conexión es necesario saber:

1.- Cómo preparar el ambiente en la computadora (instalar el conector)
2.- Cómo codificar la conexión
3.- Cómo operar la base de datos desde Java

En las siguientes subsecciones se explica cada uno de estos puntos.

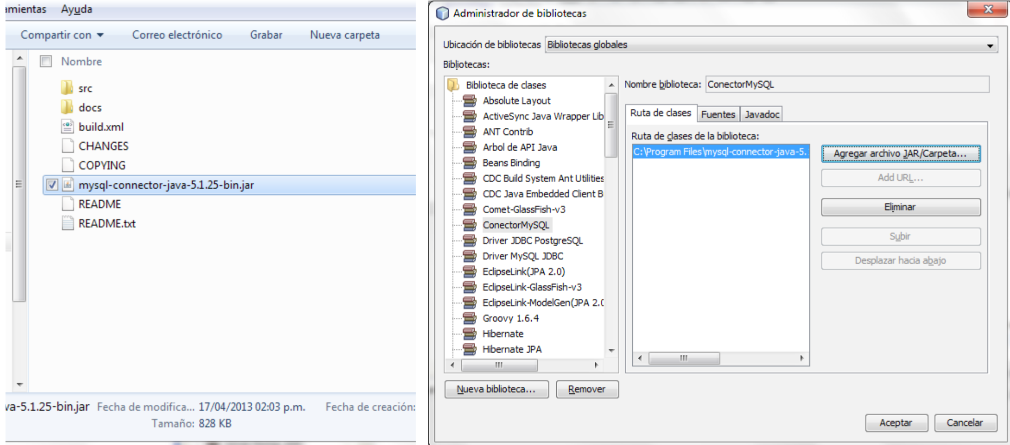
6.3.1 Ambiente

El primer paso para establecer una conexión desde NetBeans con el gestor de bases de datos MySQL es instalar el conector "mysql-connector-java", la última versión se puede descargar del sitio:

http://dev.mysql.com/downloads/connector/j/

La carpeta con el conector contiene varios archivos y subcarpetas, como se muestra en la Figura VI-5 a). El archivo más importante es el jar, en este caso: mysql-connector-java-5.1.25-bin.jar.

En NetBeans, seleccionamos Herramientas/Bibliotecas, y agregamos una nueva biblioteca con el botón "*Nueva Biblioteca*" en la Biblioteca de clases. En la Figura V b) observamos que la llamamos: ConectorMySQL. Proporcionamos la ruta en donde se encuentra mysql-connector-java-5.1.25-bin.jar y damos clic en "*Aceptar*".



a) *Carpeta con el conector* b) *Instalación del conector en NetBeans*

Figura VI-5. Instalación de mysql-connector-java

Una vez instalado el ConectorMySQL, hay que ligarlo al proyecto en donde los vamos a utilizar. Por ejemplo, una aplicación con tres capas llamada *PruebaContectBD* tiene una *index.jsp* que captura los datos, y un *servlet* `registro.Java` que usa la clase `GuardaEnBD.Java` para guardar la información capturada en la tabla "productos" de una base de datos llamada "tienda". El árbol de carpetas y archivos del proyecto es el de la Figura VI-6 a).

Para incorporar el conector de la base de datos al proyecto, hay que agregar el ConectorMySQL en la carpeta de librerías (con clic-derecho-> agregar biblioteca), como se aprecia en la VI-6Figura VI6 b). Con el conector en la librería, ya estamos listos para pasar al siguiente paso: codificar el conector.

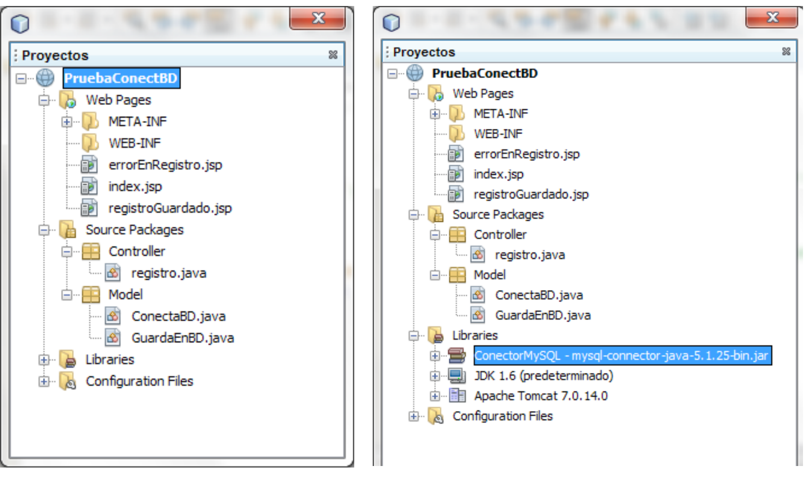


Figura VI-6. Árbol de carpetas y archivos del proyecto PruebaConectBD

6.3.2 Código del conector

Para obtener la conexión a una base de datos se utiliza el método `getConnection()` de la clase `DriverManager`, este método regresa un objeto de la clase `Connection`, y requiere tres parámetros: la URL donde se encuentra la base de datos, el nombre del usuario y el password.

La clase `Class.forName("com.mysql.jdbc.Driver")` sirve para que se cargue el *driver*, en este caso es el *driver* para conectar con MySQL.

A continuación, presentamos la clase `ConectaBD`. Su método `abrir()` regresa el objeto `Connection`. Su método `cerrar()` se asegura de que la conexión quede cerrada.

```
package Model;
import java.sql.Connection;
import java.sql.DriverManager;

public class ConectaBD {
    public static Connection con;
    private static String bd = "tienda";
    public static String usuario = "root";
    public static String passw = "ueadb01";
    public static String url = "jdbc:mysql://localhost/"+bd;

    public static Connection abrir(){
        try{
            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection(url,usuario,passw);

        } catch (Exception e) {
            System.out.println("Error en la conexion...");
            e.printStackTrace();

        }
        return con;
    }

    public static void cerrar(){
        try{
            if(con != null)
                con.close();
        } catch (Exception e){
            System.out.println("Error: No se logro cerrar
                                                  conexion:\n"+e);
        }
    }
}
```

Este tipo de mensajes sirven para hacer "debug", y no salen en el navegador sino en la consola de NetBeans, cuando hay error.

6.3.3 Escribiendo datos en una tabla

Crearemos una aplicación llamada *SistemGestionProds* la cual tendrá tres funcionalidades:

1.- Registrar productos proporcionados por el usuario.
2.- Buscar un producto en la base de datos.
3.- Mostrar todos los productos registrados.

En esta sección estudiaremos cómo registrar productos en la tabla "productos" de la base de datos "tienda".

Lo primero que haremos será crear la base de datos en MySQL, y en seguida la tabla, como se muestra en la Figura VI-7:

```
MySQL 5.5 Command Line Client
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.5.27 MySQL Community Server (GPL)

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE DATABASE tienda;
Query OK, 1 row affected (0.00 sec)

mysql> USE tienda;
Database changed
mysql> CREATE TABLE productos
    -> (clave  INT(4), nombre VARCHAR(12), precio DOUBLE(8,2), cantidad INT(4));

Query OK, 0 rows affected (0.04 sec)

mysql> DESCRIBE productos;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| clave    | int(4)      | YES  |     | NULL    |       |
| nombre   | varchar(12) | YES  |     | NULL    |       |
| precio   | double(8,2) | YES  |     | NULL    |       |
| cantidad | int(4)      | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
4 rows in set (0.01 sec)

mysql> _
```

Figura VI-7. Creación de la tabla "productos" en la base de datos "tienda"

La clase `Producto` en el paquete *model* es la misma que la que usamos para la práctica 6. La captura de datos en *index.jsp* también es igual.

Creamos la clase `GestorBD.Java`en la cual incluiremos métodos para trabajar con la base de datos. El primero que crearemos sirve para *registrar* productos en la base de datos.

El *servlet* `registro.Java` usa el método `registrar()` de la clase `GestorBD.Java` para guardar la información capturada en la tabla “productos” de la base de datos “tienda”. Si el método `registrar()` regresa verdadero significa que los datos se guardaron exitosamente, de lo contrario se redirecciona a una página que muestra un mensaje de error.

Los datos del producto (clave, nombre, precio, cantidad) están dentro del objeto `request` que el *servlet* recibió de *index.jsp*. Para desplegarlos en *registroGuardado.jsp* no es necesario incluir un atributo con el objeto `producto`, basta con el reenvío del objeto `request` (`forward(request,response)`).

Estos parámetros se obtienen en *registroGuardado.jsp* con el método `request.getParameter()`.

El código de `registro.Java` es el siguiente:

```
package Controller;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import Model.*;

@WebServlet(name = "registro", urlPatterns = {"/registro"})
public class registro extends HttpServlet {
  protected void processRequest(HttpServletRequest request,
                                HttpServletResponse response)
            throws ServletException, IOException {
     response.setContentType("text/html;charset=UTF-8");
     PrintWriter out = response.getWriter();
     try {
      int clave = Integer.parseInt(request.getParameter("clave"));
      String nombre= request.getParameter("nombre");
      Double precio =
               Double.parseDouble(request.getParameter("precio"));
      int cant = Integer.parseInt(request.getParameter("cant"));

      GestorBD gestorBD = new GestorBD();

      if (gestorBD.registrar(clave, nombre, precio, cant)){
         request.getRequestDispatcher("/registroGuardado.jsp")
                                    .forward(request, response);
          }
      else
         request.getRequestDispatcher("/errorEnRegistro.jsp")
                                    .forward(request, response);
      } finally {
          out.close();
      }
    }
```

Para entender cómo `gestorBD.registrar()` guarda un producto en la base de datos, primero agregaremos un producto directamente en la tabla de la base de datos con la instrucción INSERT INTO productos, como se muestra en la Figura VI-8:

```
MySQL 5.5 Command Line Client
4 rows in set (0.01 sec)

mysql> INSERT INTO productos
    -> VALUES('25','tenis','900','4');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM productos;
+-------+--------+--------+----------+
| clave | nombre | precio | cantidad |
+-------+--------+--------+----------+
|    25 | tenis  | 900.00 |        4 |
+-------+--------+--------+----------+
1 row in set (0.00 sec)

mysql>
```

Figura VI-8. Guardar un producto en la tabla

Recordar que para guardar un registro en una tabla con el gestor de base de datos usamos la instrucción:

```
MySQL> INSERT  INTO alumnos
         -> VALUES ('valorCol1', 'valorCol2', 'valorCol3', …,'valorColN');
```

Para guardar un nuevo registro (fila) en una tabla de la base de datos, desde Java se ejecutan cuatro pasos:

1.- Establecer la conexión con la base de datos.
2.- Crear un objeto de la clase `Statement` para poder codificar instrucciones.
3.- Codificar la instrucción para guardar un nuevo registro.
4.- Cerrar la conexión con la base de datos.

A continuación, presentamos la codificación de estos cuatro pasos en el método `registrar()` de la clase `GestorBD.java`. Observar que la instruccion para crear un `Statement` debe estar dentro de un `try-catch`, y que para guardar una nueva fila en la base de datos se usa el método `executeUpdate()` del objeto de clase `Statement` con la sintaxis:

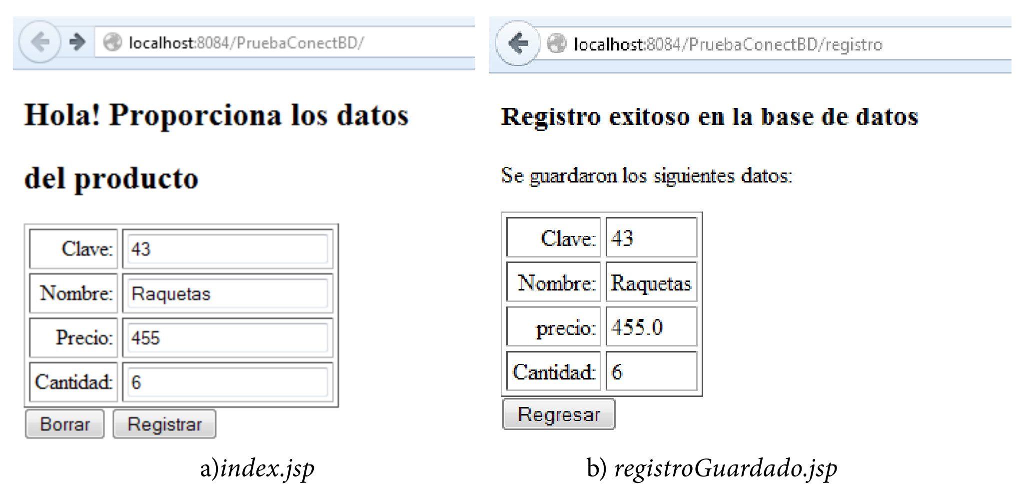
```
stm.executeUpdate("insert into tabla values('" + valorCol1 +
"','" + valorCol2 + "'," + … +"','"+ valorColN + ");");
```

Observar que el parámetro de `executeUpdate()` es la instrucción MySQL en formato `String`. En esta cadena de caracteres se concatenan los valores a insertar en la tabla (no es

sensible a las mayúsculas y minúsculas). Si los valores a insertar son numéricos, entonces no hace falta que vayan entre comillas simples. Sin embargo, las comillas simples son obligatorias para cadenas de caracteres.

```
package Model;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

public class GestorBD {

  public boolean registrar(int clave, String nombre,
                           Double precio, int cant) {
   Connection conn = null;
   Statement stm;
   ResultSet rs;
   int resultUpdate = 0;
   try{
     conn = ConectaBD.obtenConexion();
     stm = conn.createStatement();
     resultUpdate =
             stm.executeUpdate("insert into productos values("
             +clave+",'" + nombre + "'," +precio+","+cant+ ");");
     if(resultUpdate != 0){
         ConectaBD.cerrar();
         return true;
     }else{
         ConectaBD.cerrar();
         return false;
     }

      }catch (Exception e) {
        System.out.println("Error en la base de datos.");
        e.printStackTrace();
        return false;
      }
    }
}
```

En la Figura VI-9 se ilustra cómo funciona la aplicación. En la Figura VI-9 a) el usuario proporciona los datos, cuando da clic en el botón "*Registrar*" se invoca al *servlet* `registro.Java` el cual pide a `gestorBD.registrar()` que registre los datos en la tabla "productos" y si todo salió bien entonces se despliega la página *registroGuardado.jsp* de la Figura VI-9 b). El método `executeUpdate()` de la clase `Statement` regresa un `0` en caso de que no se haya ejecutado exitosamente la actualización (que puede ser un INSERT, DELETE o UPDATE). Cuando la actualización se realizó con éxito regresa un entero indicando el número de renglones afectados.



a)*index.jsp* b) *registroGuardado.jsp*

Figura VI-9. Aplicación que registra productos en una base de datos

Vemos ahora en la Figura VI-10 cómo se ha insertado una fila en la tabla "productos". Con esta inserción el nuevo producto queda registrado.

```
MySQL 5.5 Command Line Client
3 rows in set (0.00 sec)

mysql> select * from productos;
+-------+----------+--------+----------+
| clave | nombre   | precio | cantidad |
+-------+----------+--------+----------+
|    25 | tenis    | 900.00 |        4 |
|    43 | Raquetas | 455.00 |        6 |
+-------+----------+--------+----------+
2 rows in set (0.00 sec)

mysql> _
```

Figura VI-10. El producto proporcionado por el usuario queda registrado en la base de datos

El código del *servlet* `registro.java`, está preparado para cuando hay problema para dar de alta un registro en la base de datos. En este caso, se redirecciona a la página *errorEnRegistro.jsp*. Como ejemplo provocaremos un caso de error, como se muestra en la Figura VI11.

Cuando creamos la tabla "productos", sólo dimos 12 caracteres a la columna "nombre" (ver Figura VI7). El dato "Raquetas de ping-pong" sobrepasa el tamaño permitido. Para arreglar

este error podemos dar más caracteres a la columna con el comando:

```
MySQL> ALTER TABLE productos
        -> MODIFY nombre VARCHAR(30);
```

localhost:8084/PruebaConectBD/index.jsp

Hola! Proporciona los datos del producto

localhost:8084/PruebaConectBD/registro

Hubo un error y no se guardaron los datos!

a) *index.jsp* b) *errorEnRegistro.jsp*

Figura VI-11. Error al guardar el registro en la base de datos

6.3.4 Lectura de un renglón de la tabla (consulta)

La consulta de datos es una de las funciones más importantes en las aplicaciones Web. Como mencionamos en la sección VI.2, el comando MySQL para hacer consultas es SELECT. Para realizar una consulta en una tabla de la base de datos, desde Java se ejecutan cuatro pasos:

1.- Establecer la conexión con la base de datos.
2.- Crear un objeto de la clase `Statement` para poder codificar instrucciones.
3.- Codificar la instrucción con la consulta.
4.- Cerrar la conexión con la base de datos.

Ahora agregamos a la clase `GestorBD.java,` el método `consultar()`, el cual sirve para buscar un producto en la tabla *productos* de la base de datos, y contiene la codificación de los cuatro pasos mencionados. Observar que la instruccion para crear un `Statement` debe estar dentro de un `try-catch`, y que para realizar una consulta en la base de datos se usa el método `executeQuery()` del objeto de clase `Statement` con la sintaxis:

```
stm.executeQuery("select * from tabla where condicion");
```

El parámetro de `executeQuery()` es la instrucción MySQL en formato `String`, y no hace falta terminarlo con punto y coma ";".

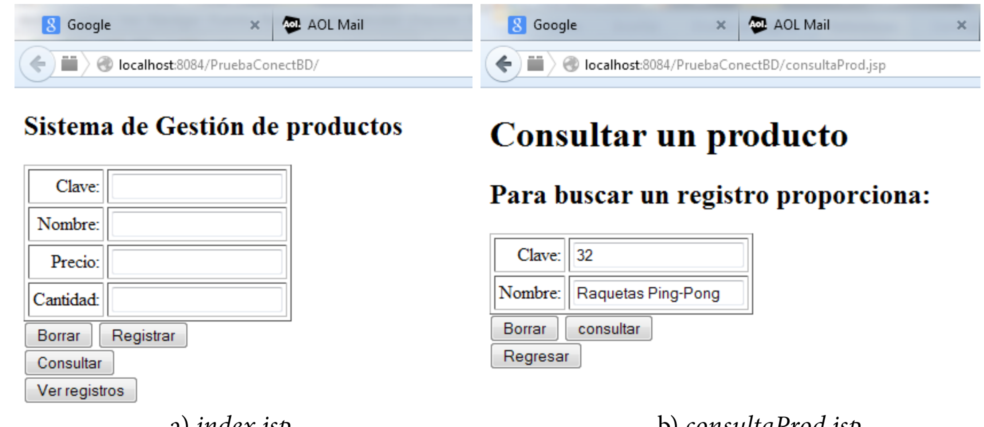
```
package Model;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;

public class GestorBD {
    Connection conn = null;
    Statement stm=null;
    ResultSet productResSet;
    Producto productHallado;
    int clav,cant;
    String nom;
    Double precio;

    public boolean registrar(int clave, String nombre,
                            Double precio, int cant) {
     . . .
    }                                                    1

    public Producto consultar(int clave, String nombre){
      try{
        conn = ConectaBD.obtenConexion();         2             3
        stm = conn.createStatement();
        productResSet = stm.executeQuery("SELECT * FROM productos
                                      WHERE clave='"+clave+
                                 "' and nombre='"+nombre+"'");
        if(!productResSet.next()){
            System.out.println(" No se encontro el registro");
            ConectaBD.cierraConexion();
            return null;                                4
        }else{
            System.out.println("Se encontró el registro");
            clav = productResSet.getInt("clave");
            nom = productResSet.getString("nombre");
            precio = productResSet.getDouble("precio");
            cant = productResSet.getInt("cantidad");
            productHallado = new Producto(clav,nom,precio,cant);

            ConectaBD.cierraConexion();
            return productHallado;
        }
      }catch(Exception e){
         System.out.println("Error en la base de datos.");
         e.printStackTrace();
         return null;
      }
    }

   public ArrayList<Producto> leerTodo(){
      . . .
    }
}
```

Para guardar el resultado de una consulta se utiliza un objeto de clase `ResulSet` el cual es de sólo lectura.

`ResultSet` es una clase Java similar a una lista en la que está el resultado de la consulta. Cada elemento de la lista es uno de los registros de la base de datos. En realidad, `ResulSet` no contiene todos los datos, sino que los va consiguiendo de la base de datos según se van pidiendo. Así, mientras que el método `executeQuery()` tarda poco, el recorrido de los elementos del `ResultSet` no es tan rápido. De esta forma se evita que una consulta que contenga muchos resultados tarde mucho tiempo y llene la memoria del programa java.

`ResultSet` tiene un apuntador interno el cual apunta inicialmente antes del primer renglón que devuelve el método `executeQuery()`. El método `next()` del `ResultSet` hace que el apuntador avance al siguiente renglón. Si lo consigue, el método `next()` devuelve **true**. Si no lo consigue significa que no hay siguiente renglón que leer, y devuelve **false**. Cuando el método `next()` devuelve **false** desde la primera vez que se le invoca significa que el registro está vacío porque no se encontró la información buscada.

Con clic en el botón "*Consultar*", se redirecciona a la página *consultaProd.jsp* que se encarga de capturar los datos para hacer la consulta (ver Figura VI-12 b):



a) *index.jsp* b) *consultaProd.jsp*

Figura VI-12. Consulta de un producto

Con el botón "consultar" de *consultaProd.jsp* el control pasa al *servlet* `consulta.jsp`, cuyo código es el siguiente:

```
package Controller;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import Model.*;
import java.util.ArrayList;

@WebServlet(name = "consulta", urlPatterns = {"/consulta"})
public class consulta extends HttpServlet {
  protected void processRequest(HttpServletRequest request,
                                HttpServletResponse response)
            throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      PrintWriter out = response.getWriter();

      int clave = Integer.parseInt(request.getParameter("clave"));
      String nombre= request.getParameter("nombre");

      try {
          Producto producto;
          GestorBD gestorBD = new GestorBD();

          producto = gestorBD.consultar(clave,nombre);

          if(producto != null){
            request.setAttribute("atribProd",producto);

            request.getRequestDispatcher("/resultadoConsulta.jsp")
                                         .forward(request, response);
          }else
            request.getRequestDispatcher("/noEncontrado.jsp")
                                         .forward(request, response);

        } finally {
            out.close();
        }
    }
}
```

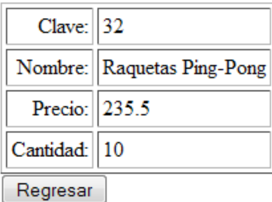
La página *noEncontrado.jsp* despliega un mensaje de error indicando que no se encontró el elemento buscado. La página *resultadoConsulta.jsp* despliega el elemento encontrado. Dejamos al lector la codificación de estas páginas, que se muestran en la Figura VI13. ResultadoConsulta.jsp debe recibir en un *scriptlet* el objeto Producto como se muestra a continuación:

Posteriormente utilizar las *expresiones*, para tener acceso al contenido de la clase:

```
<%= producto.getClave()    %>
<%= producto.getNombre()   %>
<%= producto.getPrecio()   %>
<%= producto.getCantidad() %>
```

localhost:8084/PruebaConectBD/consulta

Resultado de la Consulta



Clave:	32
Nombre:	Raquetas Ping-Pong
Precio:	235.5
Cantidad:	10

Regresar

a) *resultadoConsulta.jsp*

localhost:8084/PruebaConectBD/consulta

No se encontró el registro!

b) *noEncontrado.jsp*

Figura VI-13. Respuestas posibles cuando se pide una consulta

6.3.5 Lectura de un conjunto de renglones de la tabla

En esta sección completaremos la funcionalidad de la página *index.jsp* (Figura VI-12 a), que es la opción de ver los registros que están dados de alta en la base de datos. Cuando el usuario elige "*Ver registros*" se despliegan todos los renglones encontrados en la tabla de la base de datos, como se muestra en la página *listaProductos.jsp* de la Figura VI-14 a). Cuando no hay productos registrados se despliega la página *noHayRegistros.jsp* (Figura VI-14 b).

Los productos que están registrados son:

Clave	Nombre	Precio	Cantidad
34	pelota	15.0	25
543	aros	24.0	7
45	hamaca	735.0	8

No hay productos registrados!

```
<%
  Producto producto = (Producto)request.getAttribute("atribProd");
%>
```

a) *listaProductos.jsp* b) *noHayRegistros.jsp*

Figura VI-14. Listado de los productos registrados

Con en el botón "*Ver registros*", el control pasa al *servlet* `muestraProductos.jsp`, cuyo código es el siguiente:

```
package Controller;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import Model.*;


@WebServlet(name = "muestraProductos",
            urlPatterns = {"/muestraProductos"})
public class muestraProductos extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
            throws ServletException, IOException {
     response.setContentType("text/html;charset=UTF-8");
     PrintWriter out = response.getWriter();
     try {
       ArrayList <Producto> productos = new ArrayList<Producto>();
       Producto producto;
       GestorBD gestorBD = new GestorBD();

       productos = gestorBD.leeTodo();

       if (productos != null){
           request.setAttribute("Productos",productos);
           request.getRequestDispatcher("/listaProductos.jsp")
                                   .forward(request, response);
       }else
           request.getRequestDispatcher("/noHayRegistros.jsp")
                                   .forward(request, response);
       } finally {
            out.close();
       }
    }
```

Guardamos el resultado de la consulta en un `ArrayList`

La recepción del ArrayList productos en *listaProductos.jsp* se hace también con un *scriptlet*:

```
<%
  ArrayList<Producto> productos = null;
  productos= (ArrayList<Producto>)request.getAttribute("Productos");
%>
```

Y el listado se hace con un `for`:

```
<%
    for (Producto producto: productos)
             {
%>
    <tr valign="rigth">
      <td><%=producto.getClave()  %></td>
      <td><%=producto.getNombre()  %></td>
      <td><%=producto.getPrecio() %></td>
      <td><%=producto.getCantidad() %></td>
    </tr>

<% } %>
```

El método `leerTodos()` de la clase `GestorBD.Java` sigue los mismos pasos: establecer conexión, crear objeto de la clase `Statement`, codificar la instrucción con la consulta y cerrar la conexión. El código es el siguiente:

```
public ArrayList<Producto> leerTodos(){
  ArrayList<Producto> productos = new ArrayList<Producto>();
  try{
     conn = ConectaBD.obtenConexion();
     stm = conn.createStatement();
     productResSet = stm.executeQuery("select * from productos");
     if(!productResSet.next()){
           System.out.println(" No se encontraron registros");
           ConectaBD.cierraConexion();
           return null;
     }else{
          do{
            clav = productResSet.getInt("clave");
            nom = productResSet.getString("nombre");
            precio = productResSet.getDouble("precio");
            cant = productResSet.getInt("cantidad");
            productHallado = new Producto(clav,nom,precio,cant);
            productos.add(productHallado);
          }while(productResSet.next());
          ConectaBD.cierraConexion();
          return productos;
     }

   }catch(Exception e){
       System.out.println("Error en la base de datos.");
       e.printStackTrace();
       return null;
   }
}
```

En la Figura VI-15 se muestra el árbol de carpetas y archivos en NetBeans del proyecto completo. Sus tres capas son:

1.- *Vista*.- Las páginas JSP están dentro de la carpeta "Web Pages".
2.- *Controlador*.- Los servlets están dentro del paquete "Controller".
3.- *Modelo*.- Las clases que dan servicio a los *servlets* están dentro del paquete "Model".

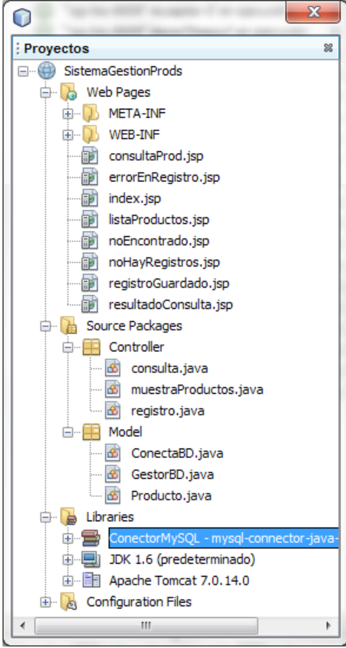


Figura VI-15. Estructura del proyecto:"Sistema de Gestión de Productos"

6.4 Errores más comunes

1.- Nombres erróneos.

a) Hay que tener especial cuidado con el nombre de los atributos en las páginas JSP, ya que no pasan por un proceso de compilación. Si el nombre de un atributo está mal, suceden cosas difíciles de detectar. Por ejemplo, no se encuentra un registro en la base de datos porque el nombre de la columna es erróneo. Así que, lo primero que hay que hacer, es verificar que todos los nombres de los atributos estén correctos.

b) Los nombres equivocados de las JSP tampoco se detectan, así que cuando se requiere la página al ejecutar la aplicación, el navegador muestra el error de "recurso no disponible".

2.- Si no hay conexión con la base de datos, es muy probable que:

a) El nombre de la URL, el usuario o el password esté incorrecto.

b) Falta incluir la librería con el conector a la base de datos en el proyecto.

3.- Es importante contemplar los casos no exitosos. En la aplicación que desarrollamos en este capítulo están incluidos los ejemplos de cómo se manejan (cuando no se encontró un registro, cuando no se pudo guardar un registro).

6.5 Práctica

6.5.1 Planteamiento

Hacer una aplicación wWeb para la gestión del "login".

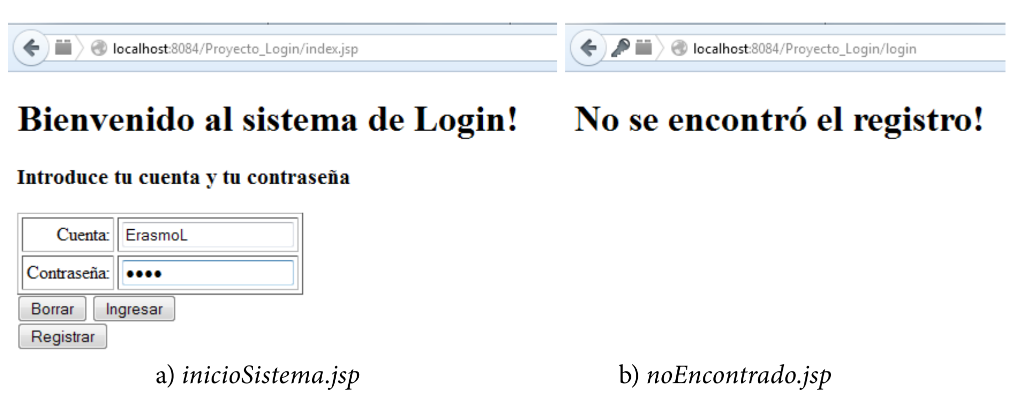
1.- En la página de inicio, usuario debe poder:

1.a.- Proporcionar su cuenta y su contraseña para ingresar a la página del sistema (Figura VI-16 a). La página inicial del sistema debe saludar utilizando el nombre asociado a la cuenta y contraseña (Figura VI-16 b):

a) *index.jsp* b) *inicioSistema.jsp*

Figura VI-16. Ingreso exitoso al sistema

En caso de que el usuario no esté registrado, se enviará un mensaje de error (Figura VI-17 b):



a) *inicioSistema.jsp* b) *noEncontrado.jsp*

Figura VI-17. Ingreso no exitoso

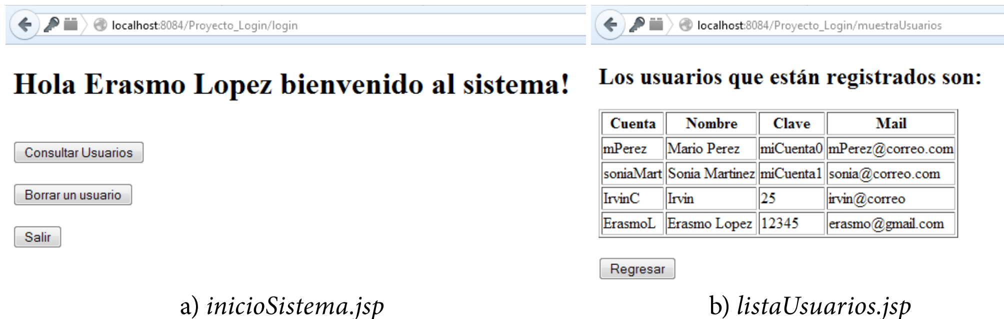
1.b.- Registrar un nuevo usuario con los siguientes datos: nombre, cuenta, contraseña, mail. Al elegir el botón “Registrar” (Figura VI-17 a) debe salir una página para capturar los datos del usuario (Figura VI-18 a). Cuando el registro se hizo con éxito se despliega una página como la de la Figura VI-18b.

a) *llenaRegistro.jsp* b) *registroGuardado.jsp*

Figura VI-18. Registro exitoso

2.- En la página del sistema hay un mensaje de bienvenida y tres botones:

2.a.- Con el botón “Ver usuarios” se despliegan todos los usuarios registrados (Figura VI-19 b). (Con el botón “Salir” se redirecciona a la página de inicio).



Cuenta	Nombre	Clave	Mail
mPerez	Mario Perez	miCuenta0	mPerez@correo.com
soniaMart	Sonia Martinez	miCuenta1	sonia@correo.com
IrvinC	Irvin	25	irvin@correo
ErasmoL	Erasmo Lopez	12345	erasmo@gmail.com

a) *inicioSistema.jsp* b) *listaUsuarios.jsp*

Figura VI-19. Ver usuarios registrados

2.b.- Con el botón "*Borrar usuario*" se podrá eliminar a un usuario registrado proporcionando su *cuenta* y su *clave*. Al elegir el botón "*Borrar un usuario*" (Figura VI-19 b) se despliega una página para capturar la cuenta y la contraseña del usuario que se quiere borrar (Figura VI-20). Al dar clic en "*Borrar un usuario*" se despliega una página con el mensaje apropiado (se borró con éxito o no). En la Figura VI-20 se muestra un caso de borrado con éxito y uno sin éxito:

localhost:8084/Proyecto_Login/borraUsuario

El resgistro se borró con éxito

localhost:8084/Proyecto_Login/borraUsuario

No se pudo borrar el registro

a) *borrado exitoso* b) *borrado no exitoso*

Figura VI-20. Borrar un usuario de la lista de usuarios registrados

6.5.2 Solución

Página de ingreso al sistema *index.jsp*:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Ingreso al sistema</title>
    </head>
    <body>
      <h1>Bienvenido al sistema de Login!</h1>
      <%@ page import="Controller.login" %>
      <h3> Introduce tu cuenta y tu contraseña</h3>
      <form action="login" method="post">
        <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
            <td align="right"> Cuenta: </td>
            <td><input type="text" name="cuenta"></td>
          </tr>
          <tr>
            <td align="right"> Contraseña: </td>
            <td><input type="password" name="clave"></td>
          </tr>
        </table>

        <input type="reset" value="Borrar">
        <input type="submit" value="Ingresar">
      </form>

      <form action="llenaRegistro.jsp" method="post">
        <input type="submit" value="Registrar">
      </form>

    </body>
</html>
```

La clase `Usuario.java`:

```
package Model;

public class Usuario {
    private String cuenta;
    private String  nombre;
    private String  clave;
    private String  mail;

    public Usuario(String cuenta, String nombre,
                   String clave, String mail){
        this.cuenta = cuenta;
        this.nombre = nombre;
        this.clave = clave;
        this.mail = mail;
```

```
    }

    public String getCuenta(){
        return cuenta;
    }

    public String getNombre(){
        return nombre;
    }

    public String getClave(){
        return clave;
    }

    public String getMail(){
        return mail;
    }
}
```

6.5.2.1 El acceso a la base de datos

La conexión y desconexión a la base de datos con `ConectaBD.java`:

```
package Model;
import java.sql.Connection;
import java.sql.DriverManager;

public class ConectaBD {
    public static Connection con=null;
    private static String bd = "sistemalogin";
    public static String usuario = "root";
    public static String passw = "ueadb01";
    public static String url = "jdbc:mysql://localhost/"+bd;

    public static Connection abrir(){
        try{
            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection(url,usuario,passw);
            System.out.println("Conexión exitosa:"+con);

        } catch (Exception e) {
            System.out.println("Error en la conexion...");
            e.printStackTrace();
            return null;
        }
        return con;
    }

    public static void cerrar(){
        try{
            if(con != null)
                con.close();
        } catch (Exception e){
            System.out.println("Error: No se logro cerrar
                                                  la conexion:\n"+e);
        }
    }
}
```

6.5.2.2 El registro

La página *llenaRegistro.jsp*:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Llena Registro</title>
    </head>
    <body>

      <%@ page import="Controller.registro" %>
      <h3> Registra tus datos</h3>
      <form action="registro" method="post">
        <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
            <td align="right"> Cuenta: </td>
            <td><input type="text" name="cuenta"></td>
          </tr>
           <tr>
            <td align="right"> Nombre: </td>
            <td><input type="text" name="nombre"></td>
          </tr>
          <tr>
            <td align="right"> Contraseña: </td>
            <td><input type="text" name="clave"></td>
          </tr>
           <tr>
            <td align="right"> Mail: </td>
            <td><input type="text" name="mail"></td>
          </tr>
        </table>

        <input type="reset" value="Borrar">
        <input type="submit" value="Guardar">
      </form>

      <form action="index.jsp" method="post">
        <input type="submit" value="Regresar">
      </form>

    </body>
</html>
```

El servlet `registro.java`:

```
package Controller;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import Model.*;

@WebServlet(name = "registro", urlPatterns = {"/registro"})
public class registro extends HttpServlet {
  protected void processRequest(HttpServletRequest request,
                              HttpServletResponse response)
         throws ServletException, IOException {
     response.setContentType("text/html;charset=UTF-8");
     PrintWriter out = response.getWriter();
     try {
       String cuenta = request.getParameter("cuenta");
       String nombre= request.getParameter("nombre");
       String clave= request.getParameter("clave");
       String mail= request.getParameter("mail");

       GestorBD gestorBD = new GestorBD();

       if (gestorBD.registrar(cuenta, nombre, clave, mail)){
           request.getRequestDispatcher("/registroGuardado.jsp")
                                    .forward(request, response);
       }
       else
           request.getRequestDispatcher("/errorEnRegistro.jsp")
                                    .forward(request, response);
     } finally {
         out.close();
     }
  }
```

El método `registrar()` de la clase `GestorBD.java`:

```
package Model;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;

public class GestorBD {
    Connection conn = null;
    Statement stm=null;
    ResultSet usuarioResultSet;
    Usuario usuarioHallado;
    String cuent, nom, passw, mail;
```

```
    public boolean registrar(String cuenta, String nombre,
                             String clave, String mail) {
        int resultUpdate = 0;
        try{
        conn = ConectaBD.abrir();
        stm = conn.createStatement();

        resultUpdate=stm.executeUpdate("INSERT INTO usuarios
                           VALUES('" + cuenta + "','" + nombre +
                                "','" + clave +"','"+mail+ "');");
        if(resultUpdate != 0){
            ConectaBD.cerrar();
            return true;
        }else{
            ConectaBD.cerrar();
            return false;
        }

       } catch (Exception e) {
            System.out.println("Error en la base de datos.");
            e.printStackTrace();
            return false;
       }
    }
```

La página *registroGuardado.jsp*:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                 content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <%@ page import="Model.Usuario" %>
        <%
          String cuenta = request.getParameter("cuenta");
          String nombre= request.getParameter("nombre");
          String clave= request.getParameter("clave");
          String mail = request.getParameter("mail");
        %>

        <h3> Registro exitoso en la base de datos</h3>

        <p> Se guardaron los siguientes datos: </p>

        <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
             <td align="right"> Cuenta: </td>
             <td> <%= cuenta %> </td>
          </tr>
          <tr>
              <td align="right"> Nombre: </td>
              <td> <%= nombre %> </td>
```

```
            </tr>
            <tr>
               <td align="right"> Contraseña: </td>
               <td> <%= clave %> </td>
            </tr>
            <tr>
               <td align="right"> Mail: </td>
               <td> <%= mail %> </td>
            </tr>
          </table>

          <form action="index.jsp" method="post">
              <input type="submit" value="Regresar">
          </form>
    </body>
</html>
```

6. 5.2.3 La entrada al sistema (login)

El servlet `login.java`:

```
package Controller;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import Model.*;
import java.util.ArrayList;

@WebServlet(name = "login", urlPatterns = {"/login"})
public class login extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
            throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      PrintWriter out = response.getWriter();
      String cuenta = request.getParameter("cuenta");
      String clave= request.getParameter("clave");

      try {
         Usuario usuario;
         GestorBD gestorBD = new GestorBD();

         usuario = gestorBD.consultar(cuenta,clave);

         if(usuario != null){
           request.setAttribute("nombre",usuario.getNombre());
           request.getRequestDispatcher("/inicioSistema.jsp")
                                   .forward(request, response);
```

```
          }else
            request.getRequestDispatcher("/noEncontrado.jsp")
                                      .forward(request, response);

       } finally {
            out.close();
       }
    }
```

El método `consultar()` de la clase `GestorBD.java`:

```
public Usuario consultar(String cuenta, String clave){
  try{
      conn = ConectaBD.abrir();
      stm = conn.createStatement();
      usuarioResultSet = stm.executeQuery("SELECT * FROM
                                   usuarios WHERE cuenta='"+cuenta+
                                          "' and clave='"+clave+"';");
      if(!usuarioResultSet.next()){
          System.out.println(" No se encontro el registro");
          ConectaBD.cerrar();
          return null;
      }else{
          System.out.println("Se encontró el registro");
          cuent = usuarioResultSet.getString("cuenta");
          nom = usuarioResultSet.getString("nombre");
          passw = usuarioResultSet.getString("clave");
          mail = usuarioResultSet.getString("mail");
          usuarioHallado = new Usuario(cuent,nom,passw,mail);

          ConectaBD.cerrar();
          return usuarioHallado;
      }
    }catch(Exception e){
       System.out.println("Error en la base de datos.");
       e.printStackTrace();
       return null;
    }

  }
```

La página *llenaRegistro.jsp*:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
       <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
       <title>Inicio Sistema</title>
    </head>
    <body>
      <%@ page import="Controller.registro" %>
      <%
         String nombre= (String) request.getAttribute("nombre");
      %>
      <h1> Hola <%= nombre %> bienvenido al sistema!</h1>
      <br>
      <form action="muestraUsuarios" method="post">
          <input type="submit" value="Consultar Usuarios">
      </form>
      <br>

      <form action="capturaBorrado.jsp" method="post">
        <input type="submit" value="Borrar un usuario">
      </form>
      <br>
      <form action="index.jsp" method="post">
        <input type="submit" value="Salir">
      </form>

    </body>
</html>
```

6.5.2.4 Borrar un registro

La página *capturaBorrado.jsp*:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                 content="text/html; charset=UTF-8">
        <title>Captura para borrar</title>
    </head>
    <body>
      <%@ page import="Controller.borraUsuario" %>
      <h3> Datos del usuario a borrar</h3>
      <form action="borraUsuario" method="post">
        <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
            <td align="right"> Cuenta: </td>
            <td><input type="text" name="cuenta"></td>
          </tr>
```

```
          <tr>
            <td align="right"> Contraseña: </td>
            <td><input type="password" name="clave"></td>
          </tr>

        </table>

        <input type="reset" value="Borrar">
        <input type="submit" value="Borrar usuario">
      </form>

      <form action="inicioSistema.jsp" method="post">
        <input type="submit" value="Regresar">
      </form>
    </body>
</html>
```

El *servlet* borraUsuario.jsp:

```
package Controller;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import Model.*;

@WebServlet(name = "borraUsuario",
            urlPatterns = {"/borraUsuario"})
public class borraUsuario extends HttpServlet {

  protected void processRequest(HttpServletRequest request,
                                HttpServletResponse response)
            throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
      String cuenta = request.getParameter("cuenta");
      String clave= request.getParameter("clave");

      GestorBD gestorBD = new GestorBD();

      if (gestorBD.borrar(cuenta, clave)){
           request.getRequestDispatcher("/registroBorrado.jsp")
                                     .forward(request, response);
      }else
           request.getRequestDispatcher("/noBorroRegistro.jsp")
                                     .forward(request, response);
    } finally {
         out.close();
  }
}
```

El método `borrar()` de la clase `GestorBD.java`:

```
public boolean borrar(String cuenta, String clave){
  int resultUpdate = 0;
  try{
      conn = ConectaBD.abrir();
      stm = conn.createStatement();

      resultUpdate= stm.executeUpdate("DELETE FROM usuarios
                                    WHERE(cuenta='" + cuenta +
                                 "'and clave='"+ clave +"');");
      if(resultUpdate != 0){
          ConectaBD.cerrar();
          return true;
      }else{
          ConectaBD.cerrar();
          return false;
      }
   } catch (SQLException e) {
         System.out.println("Error en la base de datos.");
         e.printStackTrace();
         return false;
   }
  }
```

RegistroBorrado.jsp:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                  content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>El resgistro se borró con éxito</h1>
    </body>
</html>
```

noBorroRegistro.jsp:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
                  content="text/html; charset=UTF-8">
        <title>No se borro</title>
    </head>
    <body>
        <h1>No se pudo borrar el registro</h1>
    </body>
</html>
```

6.5.2.5 Desplegar todos los usuarios registrados

El *servlet* `muestraUsuarios.java`:

```
package Controller;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import Model.*;


@WebServlet(name = "muestraUsuarios",
            urlPatterns = {"/muestraUsuarios"})
public class muestraUsuarios extends HttpServlet {
  protected void processRequest(HttpServletRequest request,
                                HttpServletResponse response)
          throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
         ArrayList <Usuario> usuarios = new ArrayList<Usuario>();
         Usuario usuario;
         GestorBD gestorBD = new GestorBD();

         usuarios = gestorBD.leeTodos();
         if (usuarios != null){
            request.setAttribute("Usuarios",usuarios);
            request.getRequestDispatcher("/listaUsuarios.jsp")
                                          .forward(request, response);
         }else
            request.getRequestDispatcher("/noHayRegistros.jsp")
                                          .forward(request, response);
     } finally {
         out.close();
     }
  }
```

El método `leeTodos()` de la clase `GestorBD.java`:

```
public ArrayList<Usuario> leeTodos(){
  ArrayList<Usuario> usuarios = new ArrayList<Usuario>();
  try{
    conn = ConectaBD.abrir();
    stm = conn.createStatement();
    usuarioResultSet = stm.executeQuery("SELECT * FROM usuarios");
    if(!usuarioResultSet.next()){
        System.out.println(" No se encontraron registros");
        ConectaBD.cerrar();
        return null;
    }else{
       do{
          cuent = usuarioResultSet.getString("cuenta");
          nom = usuarioResultSet.getString("nombre");
          passw = usuarioResultSet.getString("clave");
          mail = usuarioResultSet.getString("mail");
          usuarioHallado = new Usuario(cuent,nom,passw,mail);
          usuarios.add(usuarioHallado);
       }while(usuarioResultSet.next());
          ConectaBD.cerrar();
          return usuarios;
    }
  }catch(Exception e){
     System.out.println("Error en la base de datos.");
     e.printStackTrace();
     return null;
 }
}
```

La página *listaUsuarios.jsp*:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
       <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
       <title>Listado de Usuarios</title>
    </head>
    <body>
      <%@ page import="Model.Usuario, java.util.ArrayList" %>

      <h2>Los usuarios que están registrados son: </h2>

      <%
         ArrayList<Usuario> usuarios = null;
         usuarios =
             (ArrayList<Usuario>)request.getAttribute("Usuarios");
      %>
```

```
        <table border="1">
          <tr>
             <th>Cuenta</th>
             <th>Nombre</th>
             <th>Clave</th>
             <th>Mail</th>
          </tr>

          <%
            for (Usuario usuario: usuarios)
            {
          %>
          <tr valign="rigth">
             <td><%=usuario.getCuenta()  %></td>
             <td><%=usuario.getNombre()  %></td>
             <td><%=usuario.getClave() %></td>
             <td><%=usuario.getMail() %></td>

          </tr>

          <% } %>

        </table>

        <br>
        <form action="inicioSistema.jsp" method="post">
           <input type="submit" value="Regresar">
        </form>

     </body>
</html>
```

En la Figura VI-21 se muestra el árbol de carpetas y archivos del proyecto:

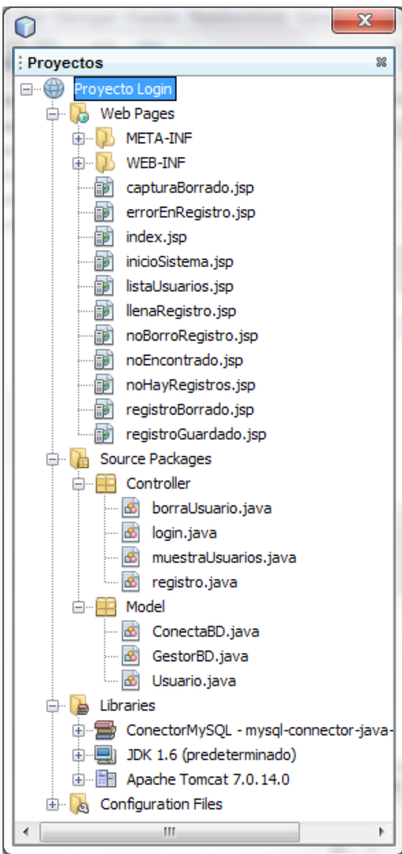


Figura VI-21. Estructura del proyecto: Login

7. Algunos aspectos adicionales

7.1 Objetivos

- Utilizar variables de sesión para evitar la pérdida de información.
- Saber validar datos de entrada.
- Conocer los Java Beans.
- Entender qué son y para qué sirven las secuencias de escape.
- Saber cómo pasar una base de datos de una computadora a otra.

7.2 Introducción

En el capítulo anterior hemos aprendido a desarrollar una aplicación Web con conexión a una base de datos que tiene las funcionalidades básicas: buscar un registro, darlo de alta, darlo de baja y desplegar todos los registros de una tabla. Sin embargo, aún faltan por estudiar aspectos adicionales que brindan robustez a estas aplicaciones. En este capítulo explicamos algunos de los más sencillos. El mundo de las aplicaciones Web es bastante extenso. En este curso hemos visto sólo una introducción que permitirá posteriormente comprender cursos más avanzados.

7.3 Seguimiento de una sesión

En ocasiones es muy útil guardar datos que estén disponibles para todas las páginas y *servlets* de una sesión. Esto evita tener que pasar constantemente la información de un lado a otro.

Por ejemplo, en la práctica del capítulo anterior, la página *index.jsp* pasa a *inicioSistema.jsp* los datos del usuario. Y en *inicioSistema.jsp*, que es la página de bienvenida, se escribe el nombre del usuario, como se observa en la Figura VII-1 b):

a) *index.jsp* b) *inicioSistema.jsp*

Figura VII-1. Ingreso exitoso, se despliega el nombre en la página de bienvenida

Pero si después accesamos otra página, por ejemplo la lista de todos los usuarios registrados (Figura VII-2 a), y regresamos a la página de bienvenida, entonces el nombre del usuario se pierde, como se observa en la Figura VII-2 b:

a) *listaUsuarios.jsp* b) *inicioSistema.jsp*

Figura VII-2. Ver usuarios registrados y regresar

Si guardamos el nombre del usuario en el objeto `session` de clase `HttpSession`, podemos acceder a éste en cualquier momento.

Hemos modificado el código del *servlet* `login.Java` para guardar el atributo “`nombre`” en una sesión, en lugar de guardarlo en el `request`.

```
package Controller;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import Model.*;
import java.util.ArrayList;
import javax.servlet.http.HttpSession;

@WebServlet(name = "login", urlPatterns = {"/login"})
public class login extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        String cuenta = request.getParameter("cuenta");
        String clave= request.getParameter("clave");

        try {
           Usuario usuario;
           GestorBD gestorBD = new GestorBD();

           usuario = gestorBD.consultar(cuenta,clave);

           if(usuario != null){
             HttpSession session = request.getSession();
             session.setAttribute("nombre", usuario.getNombre());
             //request.setAttribute("nombre",usuario.getNombre());
             request.getRequestDispatcher("/inicioSistema.jsp")
                    .forward(request, response);
           }else
             request.getRequestDispatcher("/noEncontrado.jsp")
                    .forward(request, response);

        } finally {
            out.close();
        }
    }
```

La página *inicioSistema.jsp* ahora debe obtener el `nombre` del objeto `sesión`:

```
    <%
       String nombre= (String) session.getAttribute("nombre");
    %>
    <h1> Hola <%= nombre %> bienvenido al sistema!</h1>
```

Si ejecutamos ahora la aplicación, podremos comprobar que siempre se desplegará el nombre del usuario, sin importar desde donde la accesemos.

7.4 Validación de datos de entrada

En la aplicación que desarrollamos en el capítulo anterior para la gestión de productos, se partió de la base de que el usuario llena por completo el formulario solicitado antes de seleccionar el botón "*Registrar*". Pero si el formulario no tiene todos los campos llenos, cuando seleccionamos "*Registrar*", la aplicación queda "atorada". Una de las aplicaciónes más útiles en la validación de los datos de entrada es verificar que todos los campos estén llenos para evitar que el sistema falle. En esta sección utilizaremos JavaScript para hacer esta validación.

JavaScript es un lenguaje que se usa para extender las capacidades del HTML. JavaScript funciona del lado del cliente (lo interpreta el navegador) y se incluye adentro del código de una página Web. JavaScript está orientado a documento, es decir, se usa principalmente para hacer algunas adiciones en las páginas Web. El estudio de JavaScript es tema para otro libro, aquí sólo lo aplicaremos para validar que los datos de un formulario estén completos. Podremos comprender cómo lo hace porque su sintaxis es muy similar a la de Java.

Los *tags* de inicio y fin para JavaScript son:

```
<script>
. . .
</script>
```

Los JavaScripts pueden ir en cualquier parte del código de la página Web, nosotros lo escribiremos en el <body> antes de iniciar con el código HTML.

Para accesar los campos del formulario en JavaScript, es necesario dar nombre al formulario. En nuestro ejemplo, le llamaremos `forma`:

```
<form name="forma" action="registro" method="post">
```

Y para accesar a cada uno de los campos del formulario se utiliza la siguiente sintaxis:

```
document.nombreFormulario.nombreCampo.value
```

El siguiente código es el de la página *index.jsp* del Sistema Gestor de Productos visto en el capítulo anterior. Le hemos agregado un JavaScript que verifica que todos los campos del formulario estén llenos. Los navegadores ya tienen implementada la ventana de alerta de JavaScript, para lanzarla sólo es necesario invocarla con:

```
alert("mensaje");
```

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<link rel="stylesheet" href="css/jquery.min.js">
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>

    <body>
      <%@ page import="Controller.registro" %>
      <h2> Sistema de Gestión de productos</h2>
      <script>
         function valida(form) {
            if(document.forma.clave.value =="")
               alert("falta introducir la clave");
            else{
                if(document.forma.nombre.value =="")
                   alert("falta introducir el nombre");
                else{
                    if(document.forma.precio.value =="")
                       alert("falta introducir el precio");
                    else{
                       if(document.forma.cant.value =="")
                         alert("falta introducir la cantidad");
                       else
                          form.submit();
                     }}}
         }
      </script>

      <form name="forma" action="registro" method="post">
        <table cellspacing="3" cellpadding="3" border="1" >
          <tr>
            <td align="right"> Clave: </td>
            <td><input type="text" name="clave"></td>
          </tr>
          <tr>
            <td align="right"> Nombre: </td>
            <td><input type="text" name="nombre"></td>
          </tr>
          <tr>
            <td align="right"> Precio: </td>
            <td> <input type="text" name="precio"> </td>
          </tr>
          <tr>
            <td align="right"> Cantidad: </td>
            <td> <input type="text" name="cant"> </td>
          </tr>
        </table>

        <input type="reset" value="Borrar">

        <input type="button" value="Registrar"
               onClic="valida(this.form)">
```

Recibe como parámetro una referencia al formulario

Sólo envía el formulario cuando todos sus campos están llenos

Ahora el botón "*registrar*" invoca a la función `válida` del *script*.

```
      </form>



      <form action="muestraProductos" method="post">
        <input type="submit" value="Ver registros">
      </form>

    </body>
</html>
```

En la Figura VII-3 se muestra un ejemplo de la ventana que aparece con la instrucción “alert” de JavaScript:

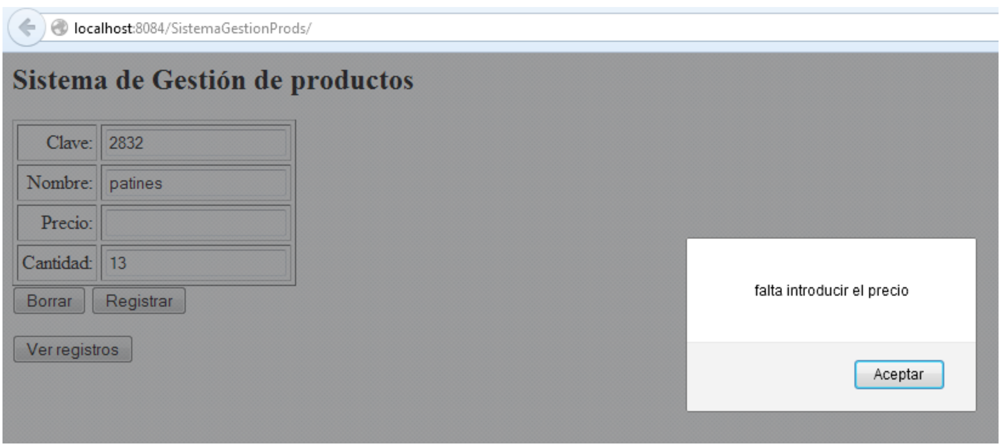


Figura VII-3. Ventana de alerta de JavaScript

7.5 Introducción a los JavaBeans

Los JavaBeans son clases Java con las siguientes reglas establecidas:

1.- Tienen un constructor que no contiene argumentos (parámetros).
2.- Todos los atributos de la clase son privados.
3.- La clase contiene métodos *getters* y *setters* para acceder y modificar los atributos.
4.- La clase debe implementar a la interfaz `serializable`.

Los JavaBeans son componentes reutilizables que ofrecen un determinado servicio. El programador que los implementa hace públicos los métodos que sirven para utilizarlo, y privados los que contienen el “cómo lo hace”, es decir, la lógica interna. A continuación, presentamos

como ejemplo el Bean llamado `producto`:

```
package negocios;
import java.io.Serializable;

public class Producto implements Serializable
{
  private int clave;
  private String nombre;
  private double precio;
  private int cantidad;

  public Producto(){
     clave = 0;
     nombre = " ";
     precio = 0.0;
     cantidad = 0;
  }

  public void setClave(int c){
     clave = c;
  }

  public void setNombre(String n){
     nombre = n;
  }

  public void setPrecio(double p){
     precio = p;
  }

  public void setCantidad(int cant){
     cantidad = cant;
  }

  public int getClave(){
     return clave;
  }

  public String getNombre(){
     return nombre;
  }

  public double getPrecio(){
     return precio;
  }

  public int getCantidad(){
     return cantidad;
  }
}
```

El JavaBean `producto` sólo tiene los métodos para acceder a sus atributos, faltaría implementar métodos que hicieran operaciones sobre sus atributos.

Existen *tags* especializados para trabajar con JavaBeans, éstos abrevian el código de los *scriptles* y las expresiones de las JSP.

Para hacer uso del Bean en una JSP existen los siguientes *tags*:

- **jsp:useBean**

Este *tag* sirve para construir un nuevo Bean, y su sintaxis es la siguiente:

```
<jsp:useBean id="nombreDelBean" class="paquete.Clase" />
```

Ejemplo:

```
<jsp:useBean id="producto" class="negocios.Producto" />
```

- **jsp:getProperty**

Este *tag* sirve para leer el atributo (propiedad) de un Bean. El valor leido se guarda en el nombre del atributo (property name). La sintaxis es la siguiente:

```
<jsp:getProperty name="nombreDelBean" property="nombreAtribu
to" />
```

Ejemplo:

```
<jsp:getProperty name="producto" property="precio" />
```

- **jsp:setProperty**

Este *tag* sirve para modificar un atributo (*property*) de un Bean, y su sintaxis es la siguiente:

```
<jsp:setProperty name="nombreDelBean" property="nombreAtributo"
value="valorAtributo" />
```

Ejemplo:

```
<jsp:setProperty name="producto" property="cantidad"
value="7" />
```

Si queremos poner en el atributo del Bean el valor que se recibe en el objeto `request`, entonces en lugar de utilizar `value`, usamos `param`.

```
<jsp:setProperty name="nombreDelBean" property="nombreAtributo"
param="nombreParametro" />
```

Ejemplo:

```
<jsp:setProperty name="producto" property="cantidad"
     param="cant" />
```

El *scriptlet* equivalente al código anterior es:

```
<% producto.setCantidad(request.getParameter("cant")); %>
```

Si el nombre del parámetro en el objeto request tiene el mismo nombre que el del atributo del Bean, entonces no será necesario usar param.

Ejemplo:

```
<jsp:setProperty name="producto" property="cantidad" />
```

El *scriptlet* equivalente al código anterior es:

```
<% producto.setCantidad(request.getParameter("cantidad")); %>
```

Cuando todos los nombres de los atributos del Bean son los mismos que los nombres en el *request*, se puede hacer la siguiente abreviación, y nos permite apreciar el porqué trabajar con Beans puede llegar a ser más práctico que los *scriptlets*.

```
<jsp:setProperty name="nombreBean" property="*" />
```

Ejemplo:

```
<jsp:setProperty name="producto" property="*" />
```

El *scriptlet* equivalente al código anterior es:

```
<% producto.setClave(request.getParameter("clave"));
   producto.setNombre(request.getParameter("nombre"));
   producto.setPrecio(request.getParameter("precio"));
   producto.setCantidad(request.getParameter("cantidad"));
 %>
```

En proyectos grandes, el uso de los JavaBeans es una práctica común. El ejemplo anterior nos permite apreciar el porqué trabajar con Beans puede llegar a ser más práctico que con *scriptlets*.

7.6 Secuencias de escape

Una *secuencia de escape* es la forma en la que se desactiva el efecto de uno o varios caracteres que forman parte de un comando. Cuando se usa la secuencia de escape, se escribe literalmente el o los caracteres.

Por ejemplo, si en una página HTLM queremos que se despliege “<html>” literalmente en la página, en lugar de que se interprete como un *tag* entonces ponemos: <code><html></code>

Con <code></html></code> en la página se desplegará: </html>

En la siguiente tabla presentamos la secuencia de escape de los caracteres especiales más utilizados:

Carácter	Secuencia de escape
'	\'
"	\"
\	\\
<%	<\%
%>	%\>

Tabla VII-1. Secuencias de escape de caracteres especiales en una JSP

7.7 Cómo pasar una base de datos de una computadora a otra

Cuando estamos desarrollando una aplicación Web que hace acceso a base de datos, es práctico saber cómo pasar la base de datos con la que estamos trabajando a otra computadora. Algunos ambientes para manejar bases de datos tienen la opción para exportar/importar bases de datos. Nosotros proporcionamos la manera generál de hacerlo.

7.7.1 Exportar la base de datos

Lo primero que hay qué hacer es localizar el directorio en donde se encuentra el comando `mysqldump`, por regla general se encuentra en el directorio **bin** de la instalación de mysql. Es necesario abrir la consola del sistema operativo y situarse en este directorio, como en el ejemplo de la Figura VII-4:

```
Símbolo del sistema
C:\Program Files (x86)\MySQL\MySQL Server 5.5\bin>_
```

Figura VII-4. La pantalla de comandos del sistema operativo

A continuación, hay que proporcionar el siguiente comando:

```
mysqldump -u usuario -p NombreDeBaseDatos > archivoSalida.sql
```

Es muy importante proporcionar una ruta de acceso en donde se tenga permiso para escribir, ya que, si no damos esta ruta, el comando intentará escibir en el directorio **bin** y se recibirá el error "acceso denegado", ya que normalmente no tenemos permiso para escribir en este subdirectorio. En la Figura VII-5 exportamos la base de datos llamada *escuela*. Nótese que en el comando de la Figura VII-5, `usuario` es el nombre del usuario con el que se accesa a la base de datos, en este caso `root`, y que el password se proporciona después de dar el comando.

```
Símbolo del sistema
C:\Program Files (x86)\MySQL\MySQL Server 5.5\bin>mysqldump -u root -p escuela >
 c:\users\usuario\dropbox\escuela_dump.sql
Enter password: *******

C:\Program Files (x86)\MySQL\MySQL Server 5.5\bin>
```

Figura VII-5.Comando *mysqldump* para exportar una base de datos a un script

El *script* con los comandos para crear la base de datos en otra computadora se escribió en el archivo *escuela_dump.sql*

7.7.2 Para importar la base de datos

Hay varias formas de importar una base de datos, puede hacerse desde la ventana del sistema operativo, o desde la ventana de comandos MySQL, (también desde un ambiente manejador de base de datos). A continuación, importaremos la base de datos que exportamos en el ejemplo de la sección anterior: *escuela_dump.sql*.

Primero hay que entrar a la ventana de comandos de MySQL y proporcionar el password. Cuando no existe la base de datos que vamos a importar, hay que crearla primero y entrar a ésta mediante los comandos CREATE DATABASE y USE, como se muestra en la Figura VII-6 a continuación:

```
MySQL 5.5 Command Line Client
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7
Server version: 5.5.27 MySQL Community Server (GPL)

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE DATABASE escuela;
Query OK, 1 row affected (0.00 sec)

mysql> USE escuela;
Database changed
mysql>
```

Figura VII-6. Ventana de comandos de MySQL

Una vez que ya nos encontramos dentro de la base de datos, ejecutamos el comando `sour-ce` y el archivo con extensión *.sql* que es donde se encuentra el *script*. Es muy importante notar que las diagonales de la ruta de acceso no deben estar invertidas (voltearlas a mano, si es necesario).

La ventana de la figura Figura VII-7 muestra la ejecución exitosa del script:

```
MySQL 5.5 Command Line Client
Server version: 5.5.27 MySQL Community Server (GPL)

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE DATABASE escuela;
Query OK, 1 row affected (0.00 sec)

mysql> USE escuela;
Database changed
mysql> SOURCE C:/Users/usuario/Dropbox/escuela_dump.sql;
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

Figura VII-7. Importación de una base de datos

8. Los principios de JavaServer Faces

8.1 Objetivos

- Conocer en qué consiste el *Framework* "JavaServer Faces" y los servicios que éste proporciona y principales componentes.
- Saber en que consisten los JavaBeans administrados y entender su alcance.
- Entender cómo se comunican las vistas con la lógica de la aplicación.
- Saber navegar desde una página Web hacia un Bean y desde una clase Java hacia una página Web.
- Conocer los elementos básicos de la interfaz de usuario en JSF.

8.2 Introducción a JavaServer Faces

JavaServer Faces (JSF) es otro entorno de desarrollo de aplicaciones Web en Java, diferente al de los *Servlets* que hemos visto hasta ahora. Esta tecnología está diseñada para simplificar el desarrollo Web con Java, y fomenta la separación de la presentación de las interfaces de usuario con la lógica de la aplicación. JSF utilizaba JSPs, añadiendo librerías que contienen componentes de alto nivel (menús, paneles, campos de texto,…), cada uno de estos componentes puede interactuar con el servidor de forma independiente. En la actualidad, se utilizan "*Facelets*" que son páginas con extensión xhtml que sirven para lo mismo que las JSF pero son una forma más sencilla de trabajar. Como JavaServer Faces es un "*framework*" (marco de referencia), simplifica el diseño de la estructura de la aplicación y también proporciona librerías que hacen más facil el desarrollo de la aplicación.

Con JavaServer Faces es posible utilizar Ajax (Asynchronous JavaScript and XML). Ajax es un enfoque en el que las diferentes acciones que solicita el usuario se realizan dentro de una misma página, de tal manera que el servidor no genera una nueva página sino sólo los datos. En las aplicaciones tradicionales, cada petición al servidor hace que éste genere una nueva página HTML/XHTML. Las aplicaciones RIA (*Rich Internet Applications*), como lo es JSF, intentan simular las aplicaciones de escritorio y, cuando usan Ajax, son más rápidas que las aplicaciones tradicionales.

La tecnología JavaServer Faces proporciona:

- Un conjunto de componentes de Interfaz de Usuario predefinidos (botones, menús, campos de texto,…), listos para agregarse a una página Web. Estos componentes se representan como objetos con un estado.
- Oyentes y manejadores de eventos (pulsación de un botón, cambio en el valor de un campo,…), los cuales permiten conectar fácilmente los eventos generados del lado del cliente con código de la aplicación en el servidor.
- Validador y convertidor de datos en el lado del servidor. Validar los datos de los componentes individuales permite informar de cualquier error antes de que éstos se envíen a procesar al servidor. El convertidor permite al usuario trabajar con datos en diferentes unidades.

JSF contiene dos importantes componentes de software: las *Tag Libraries* y los *Managed Beans*.

8.2.1 Tag Libraries

Los *Facelets* son páginas .xhtml con librerías adicionales llamadas *Tag libraries*. Las librerías de etiquetas (*tag libraries*) son componentes especiales de software que encapsulan funcionalidad dinámica y compleja. Estas librerías las construyen especialistas en ciertos servicios, como, por ejemplo: el acceso a la base de datos, o el manejo de botones, menús, desplegado de listas, etc. La idea principal es que el diseñador de las vistas pueda incorporar poderoso contenido dinámico en sus páginas sin necesidad de saber cómo se codifican los detalles.

JSTL (*JavaServer pages Standard Tag Library*) es una librería que contiene la funcionalidad más comun de las aplicaciones Web. Los diseñadores de páginas Web sólo deben saber cómo incluir los tags. JSF proporciona componentes para construir una aplicación con GUIs (Graphical User Interfaces: Interfaces de Usuario Gráficas). El desarrollador personaliza objetos como menús, check box, botones, etc. manipulando los atributos de los *tags* de cada uno de estos componentes.

Cuando creamos un proyecto en JavaServer Faces en NetBeans, se generan XHTML, las cual se llaman *facelets*, y ya tienen incluidas las *Tag libraries*.

8.2.2 JavaBeans Administrados (*managed Beans*)

Un Bean administrado (*Managed Bean*) es un *JavaBean* que puede ser accedido desde una página Web. El *managed Bean* se inventó para que el usuario remoto pueda comunicarse con un programa en Java que está en el servidor. A diferencia de los *JavaBeans*, los *managed Beans* no están obligados a implementar la interfaz serializable.
También existen los Beans CDI (*Context and Dependency Injection*). Los CDI poseen un mo-

delo un poco más avanzado para administrar los Beans, sin embargo, aún no se ha demostrado su ventaja sobre los *Managed Beans*, por lo tanto, en este curso, trabajaremos con los *Managed Beans*.

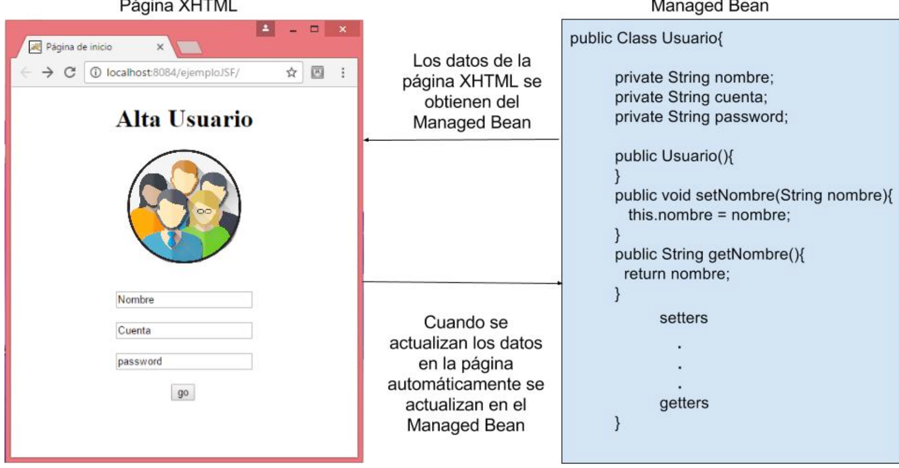


Figura VIII-1. Conexión de una página web y su managed Bean

En la Figura VIII-1 se muestra la conexión de una página wWeb con su *managed Bean* asociado. JavaServer Faces usa el mecanismo de los *managed Beans* para establecer la conexión entre los datos que se despliegan al usuario y lo que está registrado en el código Java.

8.2.3 El Model-View-Controller (MVC) con JavaServer Faces

La forma de modelar la arquitectura Model-View-Controller con JSF se puede apreciar en la Figura VIII-2, y es la siguiente:

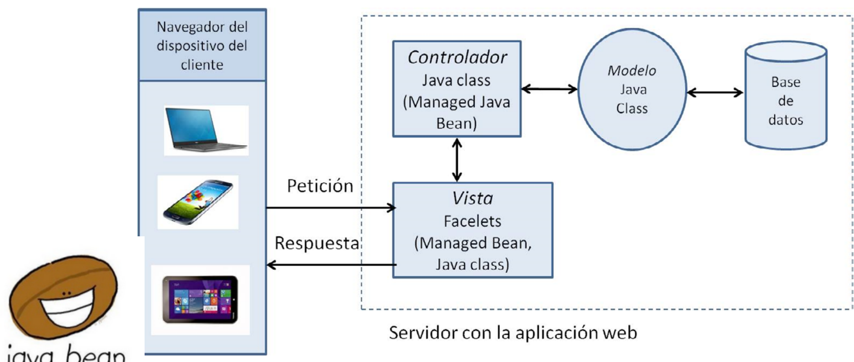


Figura VIII-2. Modelo web de tres capas con JavaServer Faces

- ***Vistas***.- Se implementan con páginas *.xhtml* a las que se les llama *facelets*. Los *Managed Beans* también pueden usarse como parte de la vista. Incluso, si una clase Java organiza las páginas Web y sus correspondientes *Managed Beans* (de la vista) entonces la clase Java también es parte de la vista.

- ***Controladores***.- Los controladores se pueden implementar con los *JavaBeans* administrados (*managed JavaBeans*) o con clases Java. Recordar que un controlador realiza las siguientes funciones: recepción/validación de peticiones, selección de la lógica a ejecutar y selección de la vista a en la que se presentará el resultado.

- ***Modelo***.- Las clases con la lógica de la aplicación se construyen con clases Java.

En la sección VIII.7 estudiaremos ejemplos de cómo implementar estas tres capas en un proyecto JavaServer Faces, en NetBeans.

8.2.4 Las anotaciones Java y la tecnología XML

XML (*Extensible Markup Language*) es un lenguaje de marcado de texto, que permite definir etiquetas personalizadas que describen y organizan datos. Se dice que es un lenguaje de etiquetas, porque cada paquete de información está delimitado por dos etiquetas como se hace en HTML, con la diferencia de que las etiquetas XML describen el significado de la información que contiene cada etiqueta, mientras que las etiquetas HTML se ocupan de la presentación del contenido y no de su significado.

El siguiente es un ejemplo de XML. Al igual que en HTML, el fin de la etiqueta se indica con el caracter "/". Nótese que las etiquetas se anidan para que el código sea más claro:

```
<session-config>
    <session-timeout>
        30
    </session-timeout>
</session-config>
```

Con la etiqueta `<session-config>` sabemos que los datos que ésta encierra se refieren a la configuración de la sesión. Con la etiqueta anidada: `</session-timeout>` podemos intuir que el dato se refiere al tiempo que una sesión puede permanecer inactiva.

Los archivos escritos en este XML son archivos de texto con la extensión XML. Estos archivos .XML siempre están presentes en las aplicaciones *Javaweb*, ya que sirven para indicar su configuración. Como el XML es un archivo clave, se genera automáticamente cuando se crea

un proyecto Web (en NetBeans, eclipse, etc…) y, aunque algunas *dependencias* se generan automáticamente, es necesario agregar las dependencias que no son autogeneradas en este archivo.

Se les llama *dependencias* a todos aquellos elementos sw que el sistema requiere para funcionar (librerías Java, librerías JSF, conector a base de datos, etc.). Escribir directamente las dependencias en los archivos XML, da pié a que se cometan errores, ya que hay que escribir mucho código repetitivo. Las anotaciones en el contexto de las aplicaciones Web, surgieron como una forma de simplificar la codificación, reducir los errores y facilitar el mantenimiento de los sistemas Web.

Las anotaciones Java permiten indicar en el código fuente cómo debe comportarse el software. También sirven para añadir *metadatos* al código fuente. Los *metadatos* son "información acerca de la información". Por eso se dice que con las anotaciones se asocia la meta-información.

Una anotación contiene el caracter "@", seguido de la instrucción correspondiente. En este curso, estudiaremos particularmente algunas de las anotaciones que se utilizan para el desarrollo de aplicaciones Web con JSF. Las anotaciones son una alternativa que simplifica en gran medida los archivos de configuración XML.

8.2.5 XHTML

El lenguaje XHTML (*Extensible HyperText Markup Languaje*) es otro lenguaje de marcado de texto que se caracteriza por la separación de la *estructura* de la información con la *presentación* de la página. XHTML se basa en HTML, pero utiliza un archivo adicional con extensión CSS (*Cascading Style Sheets*) para definir la presentación de la página. La presentación es la especificación del lugar en donde el navegador presenta cada elemento de la página Web, del *font*, y de los colores de cada uno de los elementos a desplegar.

Una de las características más importantes del XHTML es que tanto el texto como las imágenes pueden ser hipervínculos. Se pueden consultar varios manuales de XHTML en español en la Web. También recomendamos [Deitel P., Deitel H., *Internet & World Wide Web*, 2008] que tiene un capítulo dedicado a XHTML con ejemplos muy prácticos. Y otro capítulo dedicado a las hojas de estilo (CSS).

8.3 Características importantes de los JavaBeans Administrados

Hay dos maneras de declarar un *JavaBean* administrado, con anotaciones y con comandos especiales en un archivo *faces-config.xml*, en este curso trabajaremos con anotaciones.

8.3.1 Anotaciones para establecer el ámbito de los Beans

Existen diferentes ámbitos en los que puede operar un *Bean*. Los ámbitos más representativos, de menor a mayor alcance, son: petición, vista, sesión y aplicación. Con la anotación de ámbito, se establece el alcance de los métodos y atributos de un Bean. Las anotaciones para los ámbitos más utilizados son las siguientes:

@RequestScoped.- Persiste sólo durante la petición (*request*) del usuario. Cuando se envía la respuesta correspondiente (*response*), se elimina la instancia del *Bean*. El constructor se ejecuta cada vez que la página se solicita.

@ViewScoped.- Es algo intermedio entre *RequestScoped* y *SessionScoped*. El *Bean* existirá mientras la vista está activa. El *Bean* permanece activo cuando una petición no requiere cambiar a otra vista, y desaparece cuando el usuario navega hacia otra página. Esto es útil cuando se trabaja con Ajax.

@SessionScoped .- El *Bean* está activo durante toda la sesión del usuario. En otras palabras, mientras la sesión exista, existe el *Bean*.

@ApplicationScoped .- El *Bean* existe mientras la aplicación esté corriendo en el servidor.

A continuación, presentamos un ejemplo de cómo se declara un managed *Bean* con anotaciones.

8.3.2 Declarando un *Managed Bean* con anotaciones

Para declarar un managed *Bean* con anotaciones es necesario poner antes del nombre de la clase, la anotación `@ManagedBean`. En seguida se pone una anotación para declarar el alcance del *Bean*, posteriormente se declara el nombre del *Bean*, con sus atributos privados y el constructor vacío. Cada atributo de un *Bean* debe tener su método *getter* y *setter*.

JavaServer Faces administra los managed *Beans* automáticamente, por eso deben cumplir con estas características mencionadas. Las tres acciones que JSF hace de manera automática con los *managed Beans* son:

1.- Los instancia (por eso deben tener el constructor vacío).
2.- Controla su ciclo de vida (por eso deben incluir una declaración de ámbito).
3.- Llama a los métodos *getters* y *setters* (lo explicaremos en la siguiente sección).

Para declarar el managed Bean `EjemploBean` tenemos:

```
@ManagedBean
@RequestScoped
public class EjemploBean{
. . .

}
```

Cuando renombramos un *Bean*, se puede hacer referencia a él con otro nombre desde las páginas JSF, la sintaxis para renombrar es la siguiente:

```
@ManagedBean(name="nombreBean")
```

Por ejemplo:

```
@ManagedBean (name="Ejemplito")
@RequestScoped
public class EjemploBean{
. . .

}
```

8.3.3 Los tres objetos *JavaBean* en la aplicación Web

Toda aplicación Web hecha con *JavaServer Faces* tiene tres objetos que se instancian automáticamente:

1. `RequestBean`.- Es un objeto que persiste sólo durante la petición del usuario (@RequestScoped).
2. `SessionBean`.- Es un objeto que existe durante toda la sesión del usuario. Sólo hay un *sessionBean* por cada usuario.
3. `ApplicationBean`.- Es un objeto que existe mientras la aplicación se esté ejecutando en el servidor. Este objeto lo comparten todas las instancias de la aplicación.

8.4 Comunicación de las vistas con la lógica de la aplicación

8.4.1 El lenguaje EL (*Expression Languaje*)

El *JavaExpression Languaje* (EL) es un mecanismo compacto y muy poderoso para establecer la comunicación entre las páginas Web y los *JavaBeans*. Con EL se pueden enviar los datos proporcionados por el usuario en una pagina Web a la lógica de la aplicación (en un HTTP *request*), y también se pueden mostrar al usuario los datos procesados (en un HTTP *response*). Las expresiones en lenguaje EL se codifican en las páginas JSP, JSF, facelets y en archivos XML. En este curso aprenderemos algunas de las instrucciones EL más utilizadas. Para una referencia completa de EL se puede consultar el documento de especificación de EL en:

http://download.oracle.com/otndocs/jcp/el-3_0-fr-eval-spec/index.html

Las expresiones de valor (*value expressions*) son probablemente las expresiones EL más utilizadas porque son las que se utilizan para hacer referencia a los métodos y los atributos de un objeto o de un *managed Bean*. EL proporciona un conjunto de objetos implícitos que sirven para obtener valores de parámetros y atributos de diferentes ámbitos. Con las *value expressions* se pueden acceder fácilmente los métodos y atributos de un *JavaBean*, las colecciones y los datos de tipo enum.

Existen dos tipos de *value expressions*:

- rvalue.- Son aquellas que pueden leer los datos, pero no pueden sobreescribirlos. Se encierran entre llaves, precedidas por el carácter `$`, es decir, tienen la sintaxis: `${ }`
- lvalue.- Son las que pueden leer y escribir en los datos. Se encierran entre llaves, precedidas por el carácter #, es decir, tienen la sintaxis: `#{ }`

Así, para acceder al atributo de un *Bean*, se usa la notación punto:

```
#{nombreBean.atributoBean}
```

Y para acceder al método de un Bean, los paréntesis son obligatorios sólo cuando el método recibe parámetros:

```
#{nombreBean.metodoBean} // Cuando no tiene parámetros
#{nombreBean.metodoBean(…, …,…)} // Cuando tiene parámetros
```

Nótese que se utilizó letra minúscula en `nombreBean`, esto se debe a que JSF instancia automáticamente un objeto de la clase `NombreBean`, y cuando se desea acceder a un atributo del Bean con EL *hay que usar el objeto, no la clase*.

Ejemplo:

Si tenemos el *managed Bean* `EjemploBean` declarado de la siguiente forma:

```
@ManagedBean
@RequestScoped

public class EjemploBean{
   private String atributo1;
   private Integer atributo2;

   public EjemploBean(){
   }

   public void setAtributo1(String atributo1){
     this.atributo1 = atributo1;
   }

   public String getAtributo1(){
     return atributo1;
   }

   public void setAtributo2(Integer atributo2){
     this.atributo2 = atributo2;
   }

   public Integer getAtributo2(){
     return atributo2;
   }

   // otros métodos del Bean
   . . .
}
```

Entonces, usando EL podremos acceder a los atributos de la clase `EjemploBean` desde un *facelet*, con el objeto `ejemploBean`, de la siguiente manera:

`#{ejemploBean.atributo1}, #{ejemploBean.atributo2}`

8.4.1.1 Las propiedades de anidamiento en EL

Supongamos ahora que también tenemos el *Bean* `Direccion`:

```
@ManagedBean
@RequestScoped

public class Direccion{
  private String calle;
  private int numero;
  private int codigoPostal;

  public Direccion(){

  }

  public void setCalle(String calle){
     this.calle = calle;
  }

  public void setNumero(int numero){
     this.numero = numero;
  }

  public void setCodigoPostal(int codogoPostal){
     this.codigoPostal = codigoPostal;
  }


  public String getCalle(){
     return calle;
  }

  public int getNumero(){
     return numero;
  }

  public int getCodigoPostal(){
     return codigoPostal;
  }
}
```

Agregaremos un tercer atributo a `EjemploBean` de clase `Direccion`:

```
@ManagedBean
@RequestScoped

public class EjemploBean{
   private String    atributo1;
   private Integer   atributo2;
   private Direccion atributo3;

   public EjemploBean(){
   }
```

```
    public void setAtributo1(String atributo1){
      this.atributo1 = atributo1;
    }

    public String getAtributo1(){
      return atributo1;
    }

    public void setAtributo2(Integer atributo2){
      this.atributo2 = atributo2;
    }

    public Integer getAtributo2(){
      return atributo2;
    }

    public Direccion getAtributo3(){
      return atributo3;
    }

    // otros métodos del Bean
    . . .
}
```

Cuando queremos hacer referencia a uno de los atributos de `Direccion`, utilizamos la notación de anidamiento, que es la siguiente:

`#{`**`ejemploBean`**`.atributo3.calle}, #{`**`ejemploBean`**`.atributo3.numero}, #{`**`ejemploBean`**`.atributo3.codigoPostal}`

JSF invocará automáticamente a los *getters* y *setters* de `Direccion`.

Como pudimos apreciar en los ejemplos anteriores, EL nos permite enlazar la vista con el modelo.

8.4.2 Las etiquetas de JSF

En *JavaServer Faces* todo se maneja con etiquetas **<h:** …. /> y **<f:** …./>.

Como JSF se basa en la filosofía de componentes que interactúan con el servidor de forma independiente, no se manejan las etiquetas HTLM, como <form>, <input>, <select>, sino etiquetas para los componentes de JSF, como son por ejemplo <h:form>, <h:inputText>, <h:selectOneListBox>. Los componentes JSF brindan funcionalidad del lado del servidor, como conversiones de tipo, formateo, la posibilidad de definir campos de texto obligatorios, entre otras.

En cualquier etiqueta se pueden accesar los métodos y los atributos de un Bean con la siguiente sintaxis:

```
#{nombreBean.atributoBean}
 #{nombreBean.metodoBean()}
```

Las tres etiquetas básicas son:

<h: inputText …. /> para campos de texto. En el siguiente ejemplo, junto al campo de texto se pone la etiqueta *nombre del campo.*

```
<h: inputtext pt:placeholder =”nombre del campo”  />
```

<h: commandButton …. /> para botones.

Es común utilizar los botones para la navegación implícita, es decir, en el *action* se pone el nombre de la vista a la que se va a transferir el control. No se necesita poner la extensión, como en el siguiente ejemplo, en el que el botón con la etiqueta “ir a página 2” nos lleva a la vista que se llama “pagina2”:

```
 <h: commandButton value =”ir a pagina 2” action =”pagi-
na2”>
```

El otro uso de *commandButton* es transferir el control hacia el método de un *Bean*, como en el siguiente ejemplo, en el que al seleccionar el botón “Ingresar” el control pasa al método `validarUsuario()` del *Bean* `loginBean`:

```
<h:commandButton value=”Ingresar”
                 action=”#{loginBean.validarUsuario}”/>
```

<h: commandLink action=”*PaginaDestino*” value=”*Etiqueta*” />

para poner ligas a otras páginas, en *action* se pone el nombre de la página a la que se transfiere el control cuando se da clic en el enlace, y en *value* las palabras que aparecen en el enlace

Más adelante, en la sección de los componentes de la interfaz de usuario, estudiaremos más ejemplos con otros componentes.

8.4.3 Ligado de datos de la página Web con el *managed Bean*

Cuando se instancia una página web (.xhtml), se instancia también su *managed Bean* asociado. En la Figura VIII-3 se ilustra el caso en el que se despliega un *facelet* en el navegador, en ese momento se instancian todos los *managed Beans* asociados con ese *facelet*, lo que implica que en ese momento se ejecutan los constructores de los *managed Beans*.

Cada vez que el usuario modifica el dato de uno de los campos de la página Web, automáticamente se modifica su atributo correspondiente en el *managed Bean* asociado. A esto se le conoce como ligado de datos (*data binding* en inglés).

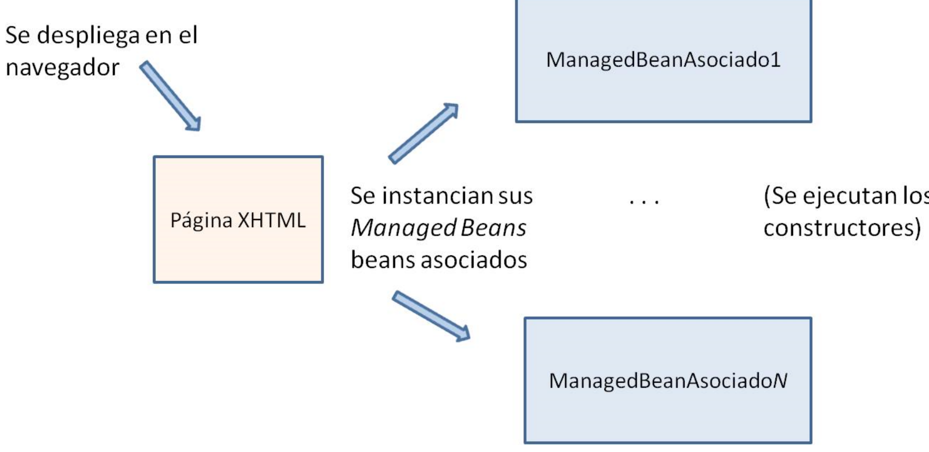


Figura VIII-3. Los Managed Beans se instancian cuando se despliega la página a la que están asociados

Es muy importante comprender lo que sucede con la vida de los *managed Beans* en función de su alcance, para poder llevar un control adecuado de la información.

@RequestScoped.- El constructor del *Managed Bean* se ejecuta cada vez que se hace un *request* desde la página asociada al *Bean* (desplegar, pulsar un botón,…). Los datos del *Bean* se actualizan cada vez que se construye Bean y se pierden cuando el *managed Bean* desaparece.

En la Figura VIII-4 se ilustra en qué momentos se activa y se desactiva el *ManagedBean* que tiene un alcance "*request*".

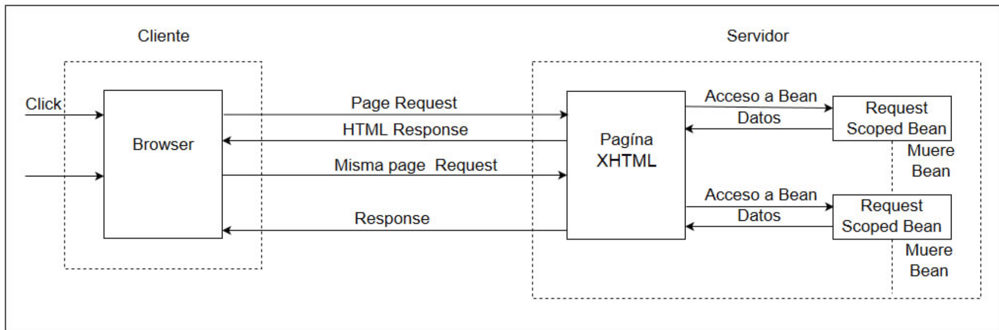


Figura VIII-4. Ciclo de vida del managed Bean "RequestScoped"

@ViewScoped.- Su constructor se ejecuta cada vez que se despliega la página asociada al *Bean*. Los datos están presentes mientras se despliegue la página y se pierden cuando la página desaparece, ya que también desaparece el *managed Bean*.

En la Figura VIII-5 se ilustra en qué momentos se activa y se desactiva el *ManagedBean* que tiene un alcance "*view*".

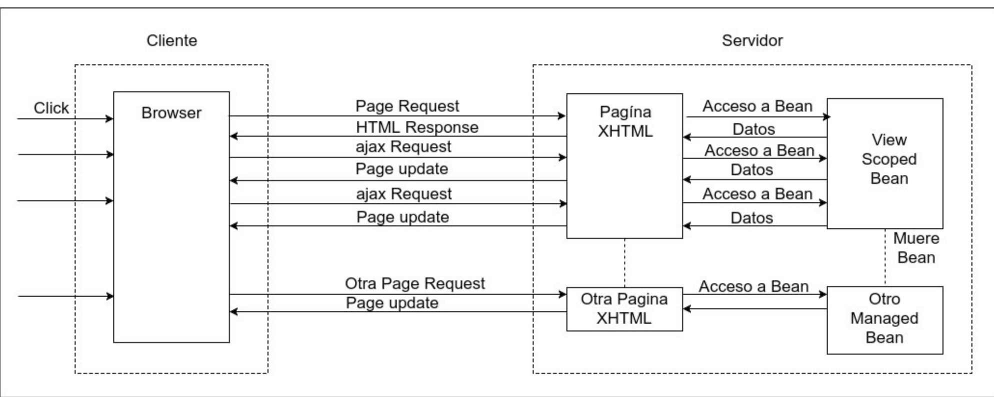


Figura VIII-5 Ciclo de vida del managed Bean "ViewScoped"

@SessionScoped.- El *Bean* se instancia la primera vez que se despliega su página asociada y permanece activo durante la sesión del usuario. Esto tiene la ventaja de que los datos del Bean permanecen durante toda la sesión, pero hay que tomar en cuenta que los datos de la página asociada no se actualizan automáticamente cada vez que ésta se visita, y que el constructor solamente se ejecuta una vez.

En la Figura VIII-6 se ilustra en qué momentos se activa y se desactiva el *ManagedBean* que tiene un alcance "*sesion*".

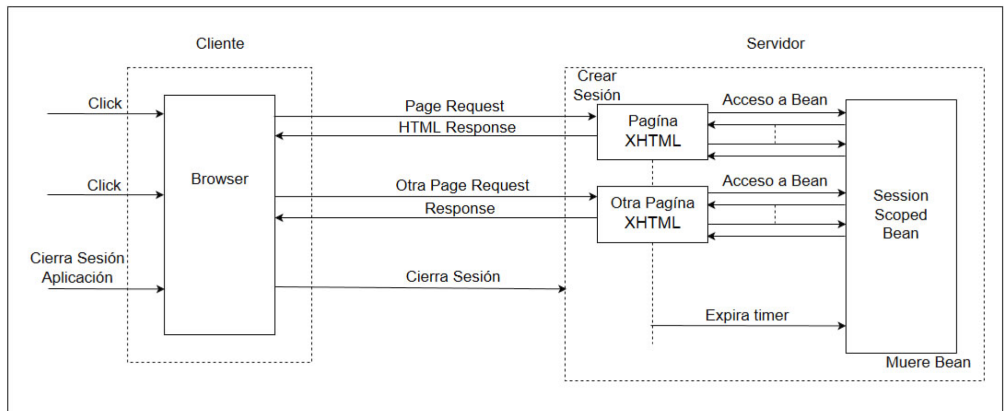


Figura VIII6. Ciclo de vida del managed Bean "SesionScoped"

8.5 La navegación en JSF

8.5.1 La navegación estática y dinámica

La *navegación estática* es la navegación más sencilla, consiste en pasar de una página a otra. En el siguiente código de ejemplo se asocia la página "`homeSistema`" a la acción del botón de comando, es decir, cuando el usuario seleccione el botón "Entrar" se desplegará la página *homeSistema.xhtml*.

```
<h: commandButton label="Entrar" action="homeSistema" />
```

Nota: no se recomienda la navegación estática en proyectos grandes, ya que se pierde la filosofía Modelo-Vista-Controlador. La decisión de desplegar una página Web debe tomarse en el controlador, no en la vista.

En la *navegación dinámica* se ejecuta un método del Managed Bean asociado, cuando el usuario selecciona un botón de la página Web. En este *Bean* se inicia el proceso de la información, y la página destino se determina después de que se finaliza este proceso.

8.5.2 Flujo desde una página Web hacia un *Bean*

Para enviar el control a un Bean es necesario indicar el nombre del *Bean* y el método al que se enviará el control. En el siguiente ejemplo, se transfiere el control al método `registrar` del Bean `login`. Como el método `registrar` no requiere parámetros, no se usan paréntesis.

```
<h: commandButton label="Registrar"
    action ="#{login.registrar}" />
```

8.5.3 Flujo desde una clase Java hacia una página Web

Hay dos opciones para desplegar una página desde una clase Java; la primera es invocar al método de un *managed Bean* que regresa el nombre de la página destino, y la segunda es con el comando *redirect*. En el primer caso, el método toma la decisión de que vista se desplegará, pero la transferencia se hace desde la página Web que lo invoca. En el segundo caso, la transferencia se hace desde la clase Java con el comando *redirect*.

8.5.3.1 Primer caso: invocar al método de un managed Bean *que regresa el nombre de la página destino*

Primero hay que hacer un *managed Bean* que implemente la interfaz `Serializable`. Dentro de este *Bean* se codifican los métodos que regresan un `String` con el nombre de la página a la que se hará la transferencia. Este `String` es el que espera el parámetro "action" del `commandButton`.

En el siguiente ejemplo, el *managed Bean* `ControladorNavegación`, al que renombramos como "navegador", contiene dos métodos. El `método1` transfiere el control a la *pagina1.xhtml* (no es necesario poner la extensión) y el `método2` transfiere a la *pagina2.xhtml*.

```
package Controller;

import javax.faces.Bean.ManagedBean;
import javax.faces.Bean.RequestScoped;
import java.io.Serializable;

@ManagedBean (name="navegador")
@RequestScoped
public class ControladorNavegacion implements Serializable {

    public ControladorNavegacion() {
    }

    public String metodo1(){
        return "pagina1";
    }

    public String metodo2(){
        return "pagina2";
    }
}
```

Se renombró el Bean.

Nombre de la página destino

En el código de *index.xhtml* se llama a un controlador de navegación, como se indica a continuación. En este caso el controlador se llama `navegador`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
    <title>Página de inicio</title>
 </h:head>
 <h:body>
   <h:form id="form">
     <center>
      <img src="images/User_Computer.png" width="156" height="156"
           alt=" no se pudo mostrar User Computer"/>
       <h2 class="title" >Hola</h2>

       <h:commandButton value="ir a página 1"
                  action="#{navegador.metodo1()}" /> <br/><br/>
       <h:commandLink value="ir a página 2"
                      action="#{navegador.metodo2()}"/>
     </center>
   </h:form>
 </h:body>
</html>
```

navegador se instancia cuando se despliega *index.xhtml.* Cada método regresa el nombre de su vista asociada

8.5.3.2 Segundo caso: transferencia desde una clase con el comando “redirect”

También se puede usar el comando “**redirect**”, para desplegar una página Web desde cualquier clase. Su sintaxis es la siguiente:

```
FacesContext.getCurrentInstance().getExternalContext().redirect("nombreVista");
```

O, de manera más general:

```
FacesContext.getCurrentInstance().getExternalContext().redirect(nombre.nombreMetodo(parámetro));
```

En este último comando, el método invocado debe devolver un *String* con el nombre de la vista. A continuación, presentamos un ejemplo:

Nota: El comando **redirect** siempre debe estar dentro de un **try-catch**. NetBeans genera automáticamente el try-catch con “clic” derecho.

Ahora la página *index.xhtml* contiene el siguiente código:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
    <title>Página de inicio</title>
 </h:head>
 <h:body>
  <h:form id="form">
    <center>
      <img src="images/User_Computer.png" width="156" height="156"
           alt=" no se pudo mostrar User Computer"/>
      <h2 class="title" >Hola</h2>

      <h3> Elige una interfaz</h3>
      <h:commandButton value="ir a sonrisa"
                 action="#{navegador.metodo2()}"/>  <br/><br/>
      <h:commandButton value="Ejemplos de elementos IU"
                 action="#{navegador.metodo3()}" /> <br/><br/>
      <h:commandButton value="ir a un Facelet"
                 action="#{navegador.metodo4()}" /> <br/><br/>

     </center>
   </h:form>
 </h:body>
</html>
```

index.xhtml despliega la página de la Figura VIII-7.

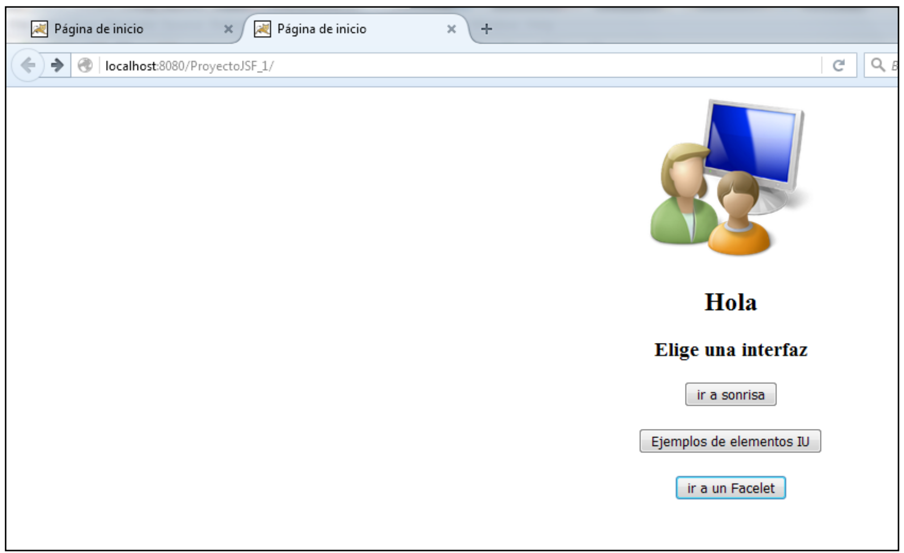


Figura VIII-7. Desplegando la vista index.xhtml

El código del *managed Bean* `navegador`:

```
package Controller;

import javax.faces.Bean.ManagedBean;
import javax.faces.Bean.RequestScoped;
import java.io.Serializable;

@ManagedBean (name="navegador")
@RequestScoped

public class ControladorNavegacion implements Serializable {
    private AdministraVista administrador;

    public ControladorNavegacion() {
        /* Crea una instancia de administrador
        this.administrador = new AdministraVista();
    }

    public void metodo1(){
        administrador.presentarInterfaz(1);
    }

    public void metodo2(){
       administrador.presentarInterfaz(2);
    }
     public void metodo3(){
        administrador.presentarInterfaz(3);
    }

    public void metodo4(){
       administrador.presentarInterfaz(4);
    }
}
```

El administrador es el que redirecciona a una interfaz

Número de la interfaz que se desea desplegar

El objeto `administrador`, instancia al objeto `catalogoInterfaces`, que tiene el método `obtenerNombreInterfaz(interfaceId)`. Este último recibe como parámetro la identidad de la vista que se va a desplegar y devuelve un `String` con el nombre de la vista requerida. La clase `AdministraVista` es la que redirecciona a la página destino por medio del comando **redirect**.

A continuación, presentamos el código de la clase `AdministraVista`.

```
package Controller;

import java.io.IOException;
import javax.faces.context.FacesContext;
import Model.Interfaces;

public class AdministraVista {
  private Interfaces catalogoInterfaces;

  public void presentarInterfaz( int interfaceId ) {
    try{
     Interfaces listaInterfaces = new Interfaces();
     FacesContext.getCurrentInstance().getExternalContext()
    .redirect(listaInterfaces.obtenerNombreInterfaz(interfaceId));

    } catch (IOException ex) {
         System.out.println("Error");
    }
 }
}
```

Recibe un número de interfaz y devuelve su nombre

La clase `Interfaces` es la siguiente:

```
package Model;

public class Interfaces {
  private String interfaces[];

    public Interfaces(){
        this.interfaces = new String[10];
        this.interfaces[1] = "index.xhtml";
        this.interfaces[2] = "sonrisa.xhtml";
        this.interfaces[3] = "muestraElementosInterfaz.xhtml";
        this.interfaces[4] = "unFacelet.xhtml";
    }

    public String obtenerNombreInterfaz(int numeroInterfaz){
        return this.interfaces[numeroInterfaz];
    }

}
```

8.6 Elementos básicos de la Interfaz de Usuario en JSF

8.6.1 Forma, botón de comando e imagen

- **Form**.- Encapsula un grupo de controles que envían datos de la página Web a la aplicación, es análogo a la etiqueta </Form> de HTML. Su sintaxis es:

```
<h: form id = "form"> … </h:form>
```

- **CommandButton**.- Muestra un botón con el texto especificado en `value`, la acción puede ser, ejecutar el método de un Bean, o navegar hacia otra página:

a) ejecutar el método de un Bean:

```
<h: commandButton value="Registrarse"
               action="{login.registrarUsuario()}" />
```

b) navegar hacia otra página:

```
<h: commandButon value="Ejecutar" action= "nombrePagDestino"
/>
```

- **GraphicImage.**- Muestra una imagen, es similar a la etiqueta <img…> de HTML. Su sintaxis es:

```
<h: graphicImage url="/imagenes/….jpg"/>
```

8.6.2 Enlaces, campos de captura y salida de texto

- **OutputLink.**- despliega un enlace en la página, cuando el usuario da clic en el enlace, el navegador despliega la página indicada en `value`. Su sintaxis:

```
<h: outputLink   value="http://www.……/" />
```

- **InputText.**- captura en el atributo de un *ManagedBean* lo que el usuario intoduce en un campo de texto. Su sintaxis:

```
<h: inputText   value="#{nombreManagedBean.atributo}" />
```

- **InputSecret.**- captura en el atributo de un *ManagedBean* lo que el usuario intoduce en un campo de texto, pero no permite ver lo que el usuario escribió. Su sintaxis:

```
<h: inputSecret   value="#{nombreBean.password}" />
```

- **OutputText.**- despliega un texto en la página. Su sintaxis:

```
<h: outputText   value="Este texto se despliega en la página"
/>
```

8.6.3 Objetos para hacer una selección

- **SelectBooleanCheckbox.**- Crea una casilla con dos estados: activado y desactivado. Su sintaxis:

```
<h:selectBooleanCheckbox
    value="#{formulario.recibirInfo}" />
```

- **SelectManyCheckbox.**- Crea un conjunto de casillas activables. Su sintaxis:

```
<h:selectManyCheckbox value="#{formulario.idiomas}">
  <f:selectItem itemValue="español" itemLabel= "Español"/>
  <f:selectItem itemValue="ingles" itemLabel= "Ingles"/>
  <f:selectItem itemValue="frances" itemLabel= "Frances"/>
  <f:selectItem itemValue="aleman" itemLabel= "Aleman"/>
</h:selectManyCheckbox>
```

- **SelectManyListbox.**- Crea una lista que permite seleccionar múltiples elementos. Su sintaxis:

```
<h:selectManyListbox value="#{formulario.lenguajes}">
  <f:selectItem itemValue="c" itemLabel= "C"/>
  <f:selectItem itemValue="c++" itemLabel= "C++"/>
  <f:selectItem itemValue="java" itemLabel= "Java"/>
  <f:selectItem itemValue="python" itemLabel= "Python"/>
</h:selectManyListbox>
```

- **SelectOneRadio.**- Crea una lista de botones, redondos normalmente, excluyentes. Su sintaxis:

```
<h:selectOneRadio value="#{formulario.genero}">
  <f:selectItem itemValue="masculino" itemLabel= "Masculi
   no"/>
  <f:selectItem itemValue="femenino" itemLabel= "Femenino"/>
</h:selectOneRadio>
```

- **SelectOneMenu.**- Crea una lista desplegable de selección excluyente. Su sintaxis:

```
<h:selectOneMenu value="#{formulario.sistema}">
  <f:selectItem itemValue="windows" itemLabel= "Windows"/>
  <f:selectItem itemValue="linux" itemLabel= "Linux"/>
</h:selectOneMenu>
```

En la figura VIII-8 se muestran ejemplos de los elementos para hacer una selección.

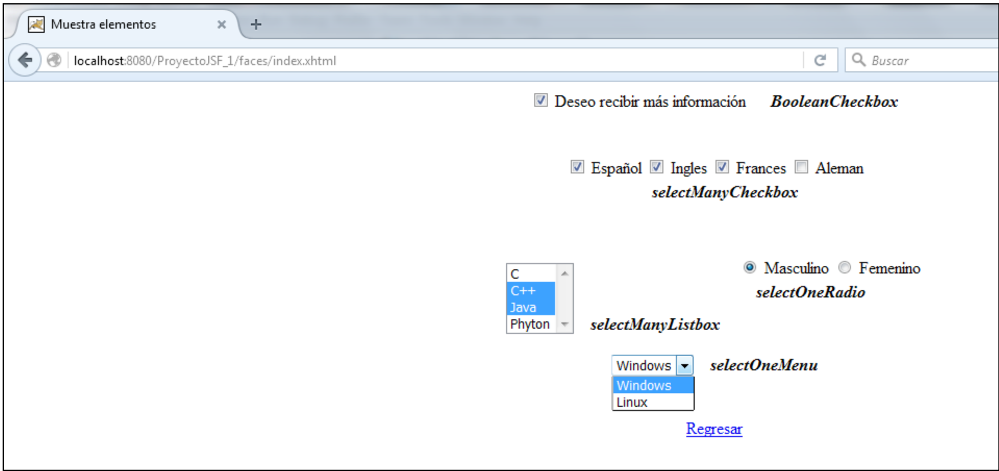


Figura VIII-8. Ejemplos de elementos de interfaz de usuario para hacer una selección

En el siguiente capítulo aprenderemos cómo hacer un proyecto JSF en *NetBeans* y cómo codificar el *facelet Muestra Elementos* y su *managed Bean* asociado.

9. Primer proyecto JSF en *NetBeans*

9.1 Objetivos

- Construir una aplicación Web en el ambiente de desarrollo *NetBeans* implementando la arquitectura MVC.
- Poner en práctica la navegación en JSF y el desplegado de elementos de interfaz de usuario.

9.2 Creación de un proyecto JSF en *NetBeans*

Para crear un proyecto JSF con *NetBeans* hay que crear una aplicación Web (*Web Application*) como hemos hecho hasta ahora, pero además hay que llegar al cuarto paso, para definir el marco de trabajo JSF, como en la Figura IX-1:

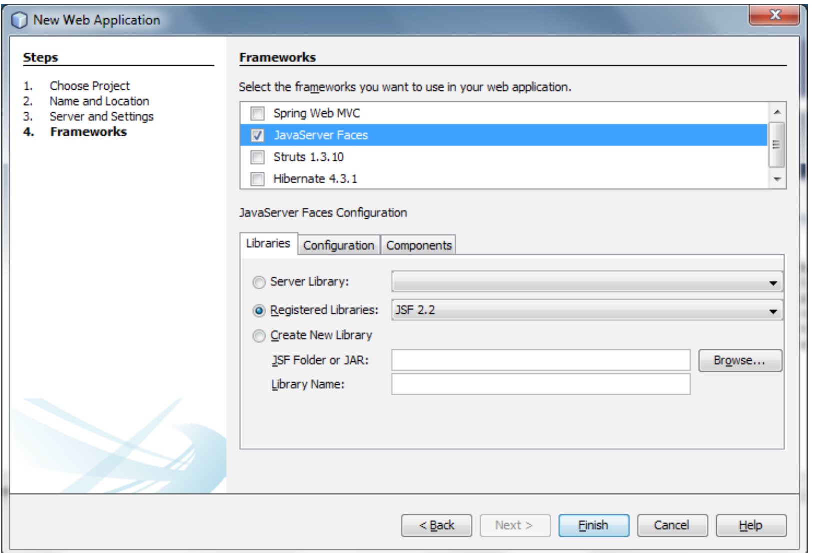


Figura IX-1. Agregando un Framework al proyecto

En la Figura IX-2 se muestra la fotografía de un proyecto *JavaServer Faces* en *NetBeans* al que hemos llamado “ProyectoJSF_1”. El archivo *web.xml* está dentro de la carpeta WEB-INF y

contiene la configuración de JSF. El proyecto se creó con el servidor Apache Tomcat, como se puede apreciar en la carpeta de librerías.

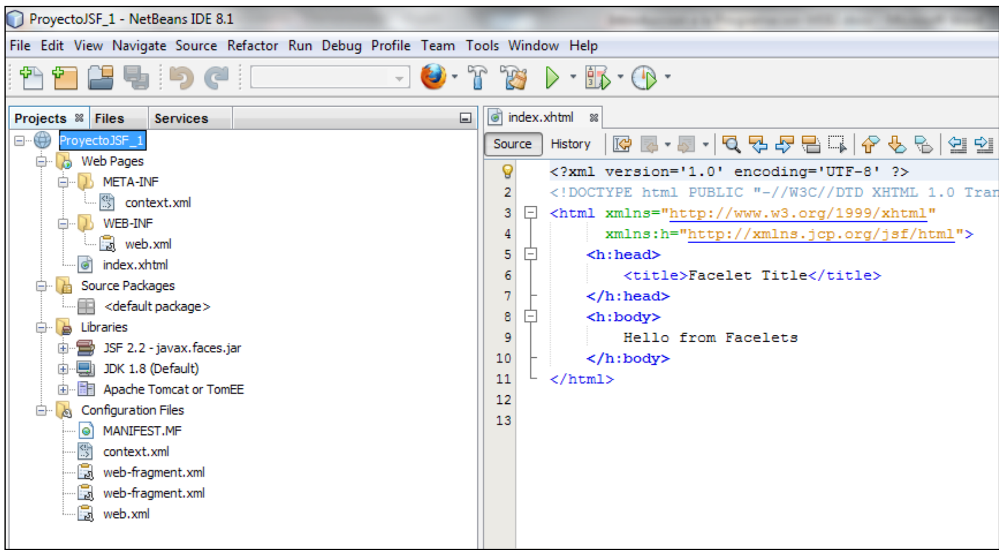


Figura IX-2. Fotografía de un proyecto en JSF

index.xhtml es la "página-home" de la aplicación, y se genera automáticamente. Si corremos el proyecto se desplegará en el navegador el contenido de esta página, como se ilustra en la Figura IX-3:

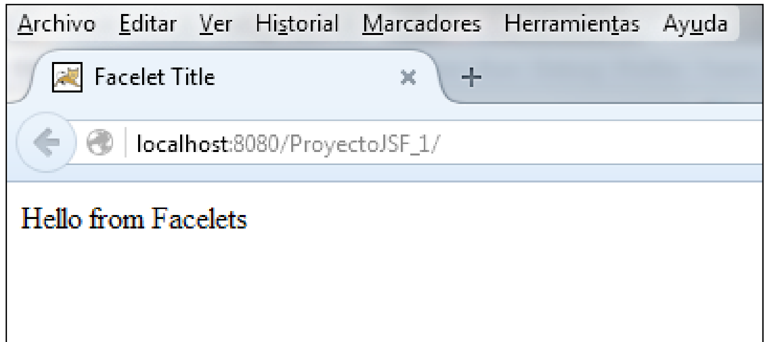


Figura IX-3. Desplegando de un Facelet en el navegador

9.2.1 Organización del proyecto implementando la arquitectura MVC

Las VISTAS.- Se deben poner en la carpeta "WebPages". El *index.xhtml* siempre es la página de inicio. En este caso las vistas son archivos con extensión xhtml, es decir, *facelets*. Para gene-

rar un *facelet* nuevo, hacer clic derecho en WebPages, y luego seleccionar *new/JSF Page*. Como ya se explicó en la sección de ligado de datos, los facelets tienen asociado un *managed Bean* en el que se actualizan los datos que el usuario introduzca en la página Web.

CONTROLADORES.- En la carpeta *Source packages* creamos un paquete al que llamaremos "Controller", donde se ponen los controladores. Es una decisión de diseño incluir los *managed Beans* como parte de la vista (en proyectos grandes) o como controladores. Para nuestros primeros proyectos JSF usaremos los *JavaBeans* administrados como controladores. Para generar un *managed Bean*, primero haremos un paquete "Controller" dentro de "*Source Packages*", luego, hacer clic derecho en "*Controller*", y seleccionar *new/JSF Managed Bean*. También es posible crear controladores como *JavaBeans* simples.

MODELO.- En la carpeta *Source packages* creamos un paquete al que llamaremos "Model", donde se ponen las clases del modelo. Todas las clases del modelo deben ser *JavaBeans*. Para generar un *JavaBean* se selecciona *new/JavaClass*. Con *NetBeans* se puede generar automáticamente el constructor, los *getters* y los *setters* de los atributos de la clase (entre otros). Se selecciona el nombre de la clase y con clic derecho: *Insert Code*, entonces aparece un menú como el de la Figura IX-4:

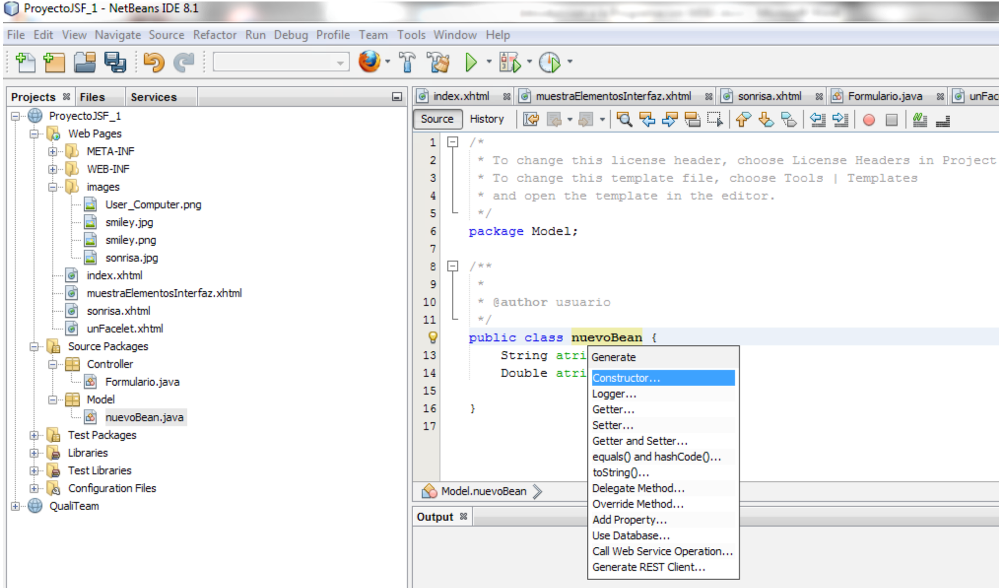


Figura IX-4. Generando código automáticamente

Se pueden generar automáticamente el constructor, los *setters* y los *getters* de los atributos declarados.

9.2.2 Desplegando una página de inicio

Haremos un facelet que despliegue la página de inicio de un sistema, como se muestra en la Figura IX-5. Cada que se hace un *facelet* en *NetBeans*, se incluyen automáticamente las librerías JSF para trabajar con HTML.

Modificaremos el código de la página index.xhtml. Nótese que el nómbre de la página: «Página de inicio» se despliega en la pestaña.

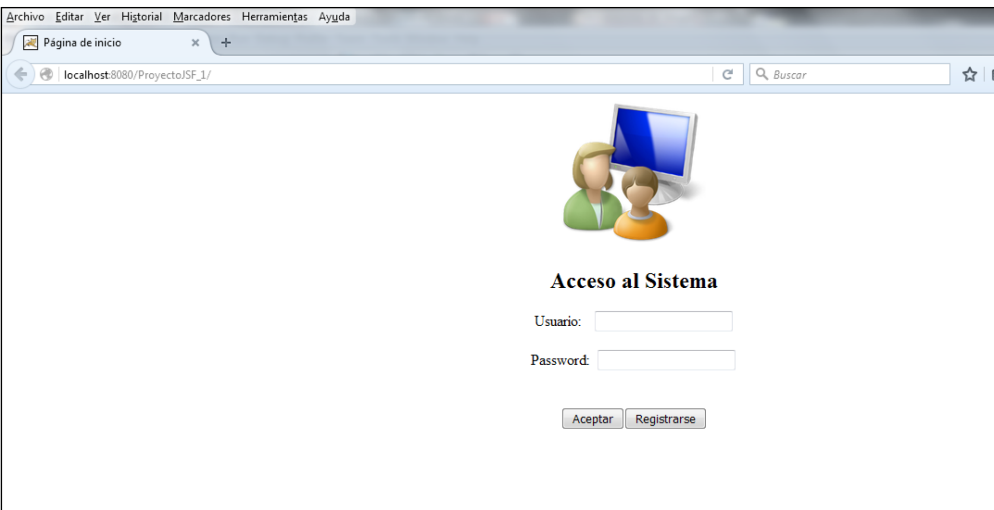


Figura IX-5. Desplegado del facelet: "Página de inicio"

Para que se despliegue la imagen agregamos una nueva carpeta que llamaremos "images". La imagen debe estar en /NetBeansProjects/ProyectoJSF_1/web/images.

Para desplegar una imagen utilizaremos la etiqueta:

h:graphicImage url = " *path donde se encuentra la imagen* "

Cuando se comienza una ruta con la diagonal: "/", significa que la primera parte de la ruta será la necesaria para llegar a la carpeta "Web Pages" del proyecto JSF.

El código de la página de la Figura IX-5 es el siguiente:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE   html   PUBLIC   "-//W3C//DTD   XHTML   1.0   Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Página de inicio</title>
    </h:head>
    <h:body>
      <h:form id="form">
        <center>
         <h:graphicImage url = "/images/User_Computer.png"
                 width="156"
                 height="156" alt="no se pudo mostrar imagen"/>
            <h2 class="title" >Acceso al Sistema</h2>

            <span style="margin-right: 15px;"> Usuario:</span>
            <h:inputText value="#{login.userName}" /> <br/><br/>
            <span style="margin-right: 9px;"> Password:</span>
            <h:inputSecret value="#{login.password}" />
            <br/><br/><br/>




            <h:commandButton actionListener="#{login.aceptar()}"
               value="Aceptar" />
            <h:commandButton value="Registrarse"
               action="{login.registrarUsuario()" />
         </center>
      </h:form>
    </h:body>
</html>
```

Librerias para usar html desde JSF

La imagen debe estar en la carpeta "images" del proyecto

Se usa para agrupar elementos en una misma línea

Atributos y métodos del Bean `login`

Los botones “Aceptar” y “Registrarse” ejecutan dos acciones diferentes. Con “Aceptar” se transfiere el control al método `aceptar()` del *Bean* `login`, mientras que con “Registrarse” el control pasa al método `registrarUsuario()` del mismo *Bean* `login`.

9.3 Una aplicación que contiene un *Managed Bean*

Ahora haremos una aplicación en la que la página principal (Figura IX6) tiene dos botones. Con el botón “Mostrar” se transfiere el control a la página *MostrarElementosInterfaz.xhtml*, con ejemplos de elementos para seleccionar, y con el botón “Ir a sonrisa” se despliega una imagen.

9.3.1 La página principal (index)

La página principal (*index.xhtml*) se ilustra en la Figura IX-6:

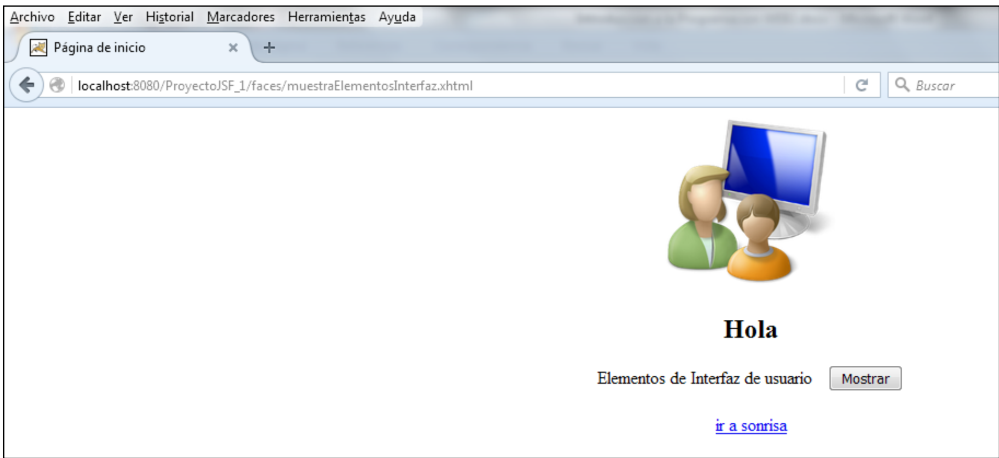


Figura IX-6. Página de inicio del sistema

El código de la página *index.xhtml* de la Figura IX-6 es el siguiente:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE   html   PUBLIC   "-//W3C//DTD   XHTML   1.0   Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
   <title>Página de inicio</title>
 </h:head>
 <h:body>
   <h:form id="form">
       <center>
          < h:graphicImage
              url = "/images/User_Computer.png" width="156"
              height = "156"
              alt = "no se pudo mostrar User Computer"/>
          <h2 class="title" >Hola</h2>

          <span style="margin-right: 15px;"> Elementos de Interfaz
                                             de usuario</span>

          <h:commandButton value="Mostrar"
             action="muestraElementosInterfaz" /> <br/><br/>
          <h:commandLink value="ir a sonrisa" action="sonrisa"/>
       </center>
   </h:form>
 </h:body>
</html>
```

Botón

Liga (enlace)

9.3.2 La página que despliega la sonrisa

Al dar clic en el enlace indicado por la etiqueta `commandLink` se transfiere el control a la página *sonrisa.xhtml*, que sólo contiene un comando para desplegar la imagen, como se aprecia en el siguiente código:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Sonrisa</title>
    </h:head>
    <h:body>
        <center>
        <h:graphicImage url = "/images/sonrisa.jpg"
                        width = "156" height="156"
                        alt = "No se pudo mostrar: Sonrisa"/>
        </center>
    </h:body>
</html>
```

La imagen se despliega como en la Figura IX-7:

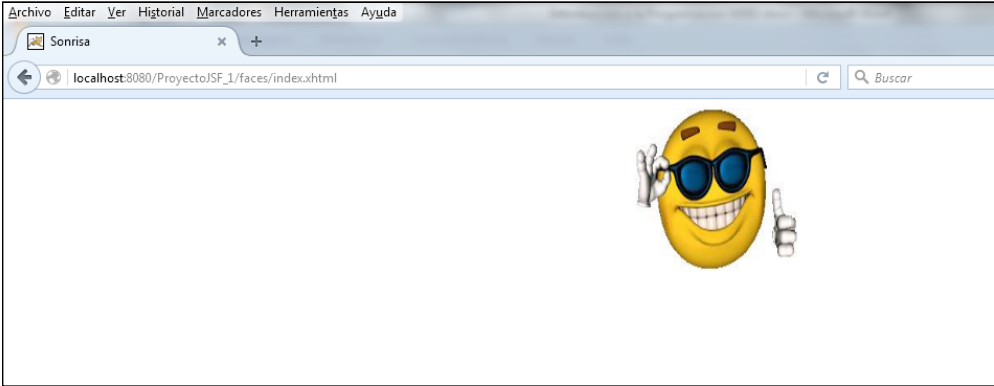


Figura IX-7 página sonrisa.xhtml

9.3.3 Desplegando elementos de interfaz de usuario

Para mostrar una página con algunos elementos para seleccionar (menú, checkbox…) comenzaremos construyendo un *facelet* al que llamaremos *muestraElementosInterfaz.xhtml*, al que agregaremos el botón más sencillo: el BooleanCheckbox.

La página *muestraElementosInterfaz.xhtml* tiene asociado un managed Bean al que llamaremos `Formulario`. Más adelante explicaremos su código.

A continuación, presentamos el código del *facelet* que despliega un BooleanCheckbox. Observa que para desplegar el *checkBox* hace uso del *managed Bean* `Formulario`, especificamente liga el *checkBox* al atributo `recibirInfo` de este *managed Bean*.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

 <h:head>
     <title>Muestra elementos</title>
 </h:head>
 <h:body>
  <h:form id="form">
    <center>
     <h:selectBooleanCheckbox value="#{formulario.recibirInfo}" />
     <span> Deseo recibir más información </span> <br/><br/>
     <h:commandLink value="Regresar" action="index"/>
    </center>
  </h:form>

 </h:body>
</html>
```

Es un *managed Bean* ligado a esta vista

El comando:

```
<h:commandButton value=”Mostrar”
                 action=”muestraElementosInterfaz” />
```

de la página index.xhtml llama a la vista *muestraElementosInterfaz.xhtml*, que hasta lo que hemos codificado sólo desplegará el botón **selectBooleanCheckbox**.

9.3.4 Lo que sucede cuando un *managed Bean* no contiene todos los *setters* y/o *getters* que se usan en la vista

Es muy importante que cada atributo del *managed Bean* que se utilice en la vista a la que está ligado, tenga su *setter* y/o su *getter*. Incluir *getter* cuando en la vista se lee el atributo y *setter* cuando éste se modifica**.** En la Figura IX-8 se muestra el mensaje de error que despliega el navegador, cuando el *Managed Bean* asociado a una página Web, no contiene el *setter* o el *getter* del atributo al cual se pretende acceder desde la página:

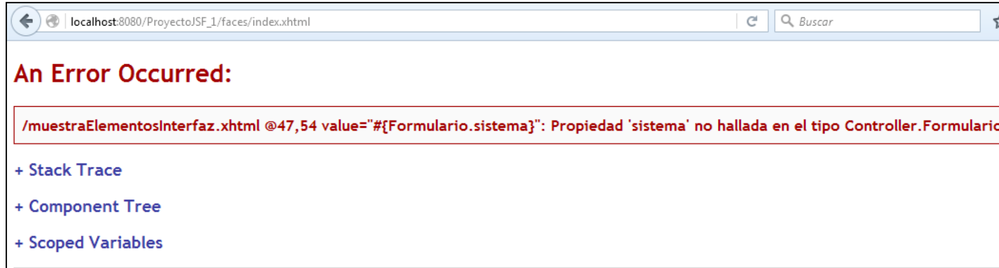


Figura IX-8. Error cuando el Bean no tiene métodos getters y setters

9.3.5 La página *muestraElementosInterfaz.xhtml*

La página que muestra elementos de interfaz de usuario para seleccionar se muestra en la Figura Figura IX-9:

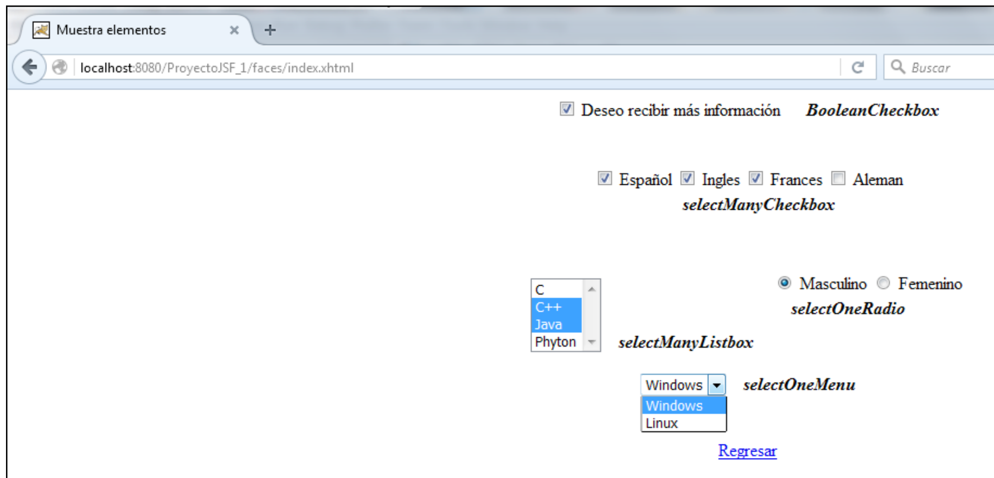


Figura IX-9. Ejemplos de elementos de interfaz de usuario para hacer una selección

El código de esta página es el siguiente:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>Muestra elementos</title>
    </h:head>
    <h:body>
      <h:form id="form">
        <center>
          <h:selectBooleanCheckbox
              value="#{formulario.recibirInfo}" />
          <span> Deseo recibir más información </span>
          <span>     </span>

         <span><i><b>BooleanCheckbox</b></i></span><br/><br/><br/>

          <h:selectManyCheckbox value="#{formulario.idiomas}">
            <f:selectItem itemValue="español"
                          itemLabel= "Español"/>
            <f:selectItem itemValue="ingles" itemLabel= "Ingles"/>
            <f:selectItem itemValue="frances"
                          itemLabel= "Frances"/>
            <f:selectItem itemValue="aleman" itemLabel= "Aleman"/>
          </h:selectManyCheckbox>
          <span>     </span>

          <span><i><b>selectManyCheckbox</b></i></span>
          <br/><br/><br/>

       <table cellspacing="5" cellpadding="5" border="0" >
        <tr>
          <td>
            <h:selectManyListbox value="#{formulario.lenguajes}">
             <f:selectItem itemValue="c" itemLabel= "C"/>
             <f:selectItem itemValue="c++" itemLabel= "C++"/>
             <f:selectItem itemValue="java" itemLabel= "Java"/>
             <f:selectItem itemValue="phyton"
                           itemLabel= "Phyton"/>
            </h:selectManyListbox>
            <span><i><b>  
                        selectManyListbox</b></i></span>
           </td>

          <td>
           <h:selectOneRadio value="#{formulario.genero}">
             <f:selectItem itemValue="masculino"
                           itemLabel= "Masculino"/>
             <f:selectItem itemValue="femenino"
                           itemLabel= "Femenino"/>
            </h:selectOneRadio>
            <span><i><b>    
                      selectOneRadio</b></i></span><br/><br/><br/>
          </td>
        </tr>
      </table>
      <h:selectOneMenu value="#{formulario.sistema}">
```

Hay que agregar esta librería

```
          <f:selectItem itemValue="windows" itemLabel= "Windows"/>
          <f:selectItem itemValue="linux" itemLabel= "Linux"/>
       </h:selectOneMenu>
       <span><i><b>  
                   selectOneMenu</b></i></span> <br/><br/><br/>

        <h:commandLink value="Regresar" action="index"/>
      </center>
    </h:form>

  </h:body>
</html>
```

Nota.- Algunos elementos requieren de la librería:

`xmlns:f=”http://java.sun.com/jsf/core”`

Como es el caso de **SelectManyCheckbox**.

9.3.6 El *managed Bean* `Formulario.java`

El código del *managed Bean* `Formulario.java`, que está asociado a la vista *muestraElementosInterfaz.xhtml*, es el siguiente:

```
package Controller;

import javax.faces.Bean.ManagedBean;
import javax.faces.Bean.RequestScoped;
import java.io.Serializable;

@ManagedBean (name="formulario")
@RequestScoped
public class Formulario{
 Boolean recibirInfo;
 String[] idiomas;
 String[] lenguajes;
 String genero;
 String sistema;

   /**
     * Creates a new instance of Formulario
     */
    public Formulario() {

    }

   public String getSistema() {
        return sistema;
    }

    public void setSistema(String sistema) {
        this.sistema = sistema;
    }
```

```
    public String getGenero() {
        return genero;
    }

    public void setGenero(String genero) {
        this.genero = genero;
    }

    public String[] getLenguajes() {
        return lenguajes;
    }

    public void setLenguajes(String[] lenguajes) {
        this.lenguajes = lenguajes;
    }


    public String[] getIdiomas() {
        return idiomas;
    }

    public void setIdiomas(String[] idiomas) {
        this.idiomas = idiomas;
    }

    public Boolean getRecibirInfo() {
        return recibirInfo;
    }

    public void setRecibirInfo(Boolean recibirInfo) {
        this.recibirInfo = recibirInfo;
    }

}
```

9.3.7 Ejecución en vivo del proyecto de ejemplo

Al final de la lección 5 de SEAWeb 2 podrás operar en vivo el ejemplo que hemos estudiado en este capítulo.

10. Captura, procesamiento y muestra de información en JSF

10.1 Objetivos

- Construir una aplicación Web en la que se capturen los datos del usuario, se procesen y se desplieguen los resultados.
- Comprender en la práctica el alcance de los *Managed Beans* y la disponibilidad de su contenido.

10.2 Introducción

En este capítulo aprenderemos a capturar datos del usuario para luego procesarlos y desplegar el resultado. Construiremos una aplicación que presenta al usuario una página de captura. Cuando el usuario proporciona la información y selecciona el botón "Calcular", se procesa la información y se despliega el resultado.

10.3 Captura de los datos del usuario

Como estudiamos anteriormente, los campos para capturar información en un *facelet* pueden ser de tipo texto o de selección. Recordemos cómo se captura desde un campo de texto:

InputText.- Lo que el usuario intoduce en el campo de texto, se captura en el atributo de un *ManagedBean*. Su sintaxis es la siguiente:

```
<h: inputText   value="#{nombreManagedBean.atributo}" />
```

Trabajaremos con un ejemplo de selección múltiple de opciones, usando **SelectManyCheck-box.** Un ejemplo de su sintaxis es:

```
<h:selectManyCheckbox value="#{formulario.idiomas}">
  <f:selectItem itemValue="español" itemLabel= "Español"/>
```

```
    <f:selectItem itemValue="ingles" itemLabel= "Ingles"/>
    <f:selectItem itemValue="frances" itemLabel= "Frances"/>
    <f:selectItem itemValue="aleman" itemLabel= "Aleman"/>
  </h:selectManyCheckbox>
```

La selección del usuario se guarda en el atributo `idiomas` del *mannaged bean* `formulario`, el cual debe ser un *ArrayList* de `String`, que guardar la lista de cadenas seleccionadas. La página de captura de datos se muestra en la Figura X-1:

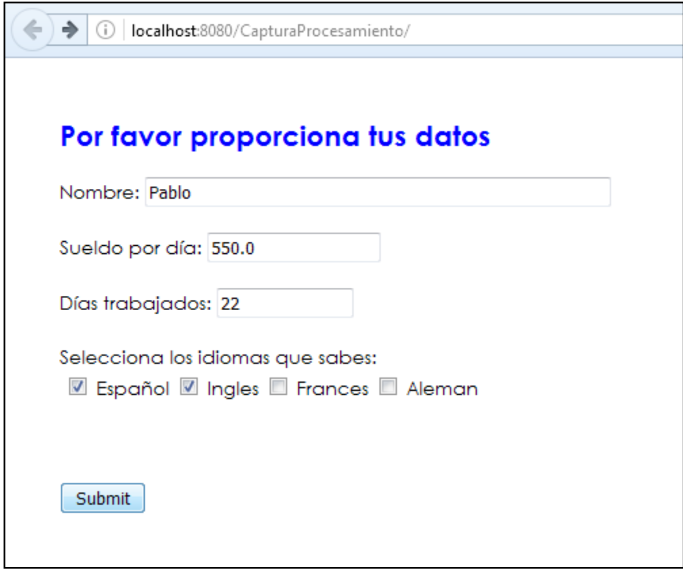


Figura X-1 Pagina de captura de los datos del usuario

10.3.1 Estructura del proyecto

Usaremos la arquitectura MVC, así que construimos un paquete llamado *Controller* para guardar ahí los dos *Managed beans* de la aplicación: el que lleva el control y el del formulario. También hay otro paquete llamado *Model* que contiene una pequeña clase que ejecuta la lógica de la aplicación. Este pequeño ejemplo, contiene sólo una multiplicación en la parte del procesamiento de los datos. La Figura X-2 muestra la estructura del Proyecto:

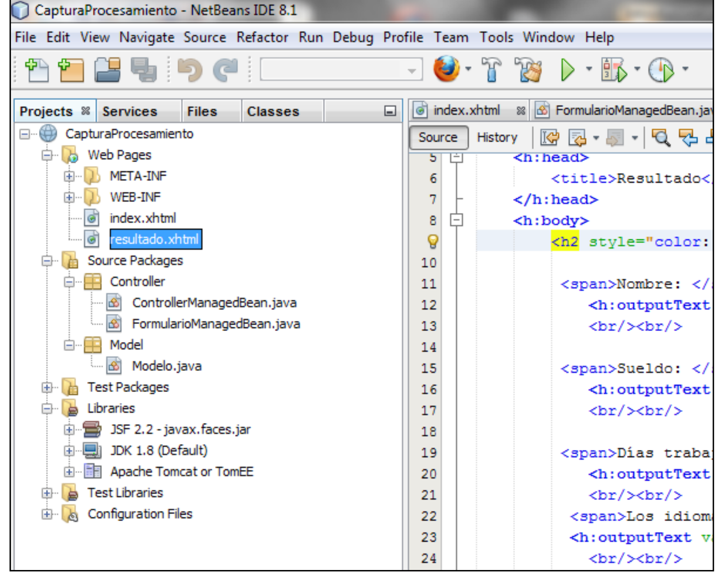


Figura X-2 Fotografía del Proyecto

10.3.2 *Managed Bean* ligado al *facelet* de captura

Para poder capturar los datos que introduce el usuario se construye un *Managed Bean* que tiene como atributos los nombres de los campos de captura del *facelet*. Recuerda que debe tener los métodos *setters* y *getters* para cada uno de los atributos que se capturan en la página Web. El código del *Managed Bean* es el siguiente:

```
@ManagedBean(name="formularioManagedBean")
package Controller;

import javax.faces.bean.ManagedBean;
import java.util.ArrayList;
import javax.faces.bean.SessionScoped;

@SessionScoped
public class FormularioManagedBean {
    private String   nombre;
    private Double   sueldo;
    private int      numDias;
    private ArrayList <String> idiomas;

    public FormularioManagedBean() {
        idiomas = new ArrayList <String>();
    }
    public String getNombre() {
        return nombre;
```

Los datos del *Managed Bean* están disponibles durante toda la sesión

Aquí se capturan cada uno de los idiomas que el usuario selecciona

```
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public Double getSueldo() {
        return sueldo;
    }
    public void setSueldo(Double sueldo) {
        this.sueldo = sueldo;
    }
    public int getNumDias() {
        return numDias;
    }
    public void setNumDias(int numDias) {
        this.numDias = numDias;
    }
    public ArrayList<String> getIdiomas() {
        return idiomas;
    }
    public void setIdiomas(ArrayList<String> idiomas) {
        this.idiomas = idiomas;
    }
}
```

10.3.3 Facelet que captura los datos

A continuación, presentamos el código HTML de la página de captura de la Figura X-1:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE   html   PUBLIC   "-//W3C//DTD   XHTML   1.0   Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
 <h:head>
     <title>Página de captura de datos del u
 </h:head>

 <h:body style="font-family: century gothic;font-size: 14px; ">
   <h:form id="form" style="width: 100%;padding: 20px 33px;">
    <h2 style="color: blue;">Por favor proporciona tus datos</h2>
      <span>Nombre: </span>
         <h:inputText value="#{formularioManagedBean.nombre}"
                      style=" width:25%;" />
      <br/><br/>

      <span>Sueldo por día: </span>
         <h:inputText value="#{formularioManagedBean.sueldo}"
                      style=" width:9%;" />
      <br/><br/>
```

Managed Bean con los datos ligados a la página

```
      <span>Días trabajados: </span>
         <h:inputText value="#{formularioManagedBean.numDias}"
                           style=" width:7%;" />
      <br/><br/>
                                                  Es un ArrayList<String>
      <span>Selecciona los idiomas que sabes: </span>
        <h:selectManyCheckbox
                  value="#{formularioManagedBean.idiomas}">
          <f:selectItem itemValue="español" itemLabel= "Español"/>
          <f:selectItem itemValue="ingles" itemLabel= "Ingles"/>
          <f:selectItem itemValue="frances" itemLabel= "Frances"/>
          <f:selectItem itemValue="aleman" itemLabel= "Aleman"/>
        </h:selectManyCheckbox>
       <br/> <br/> <br/>
       <h:commandButton value="Submit"
                     action="#{controllerManagedBean.ejecutar}" >
       </h:commandButton>
   </h:form>
 </h:body>                                        Managed Bean que
</html>                                           procesa los datos
```

10.4 Procesamiento de los datos

El controller debe tener acceso a los datos que proporcionó el usuario para poder pedir al model que los procese y posteriormente mandar a desplegar el resultado del procesamiento.

En nuestro ejemplo, el `controllerManagedBean.java` pide al *model* que haga el cálculo y después redirecciona a la página `resultados.xhtml`. Los datos del usuario se capturan en `formularioManagedBean.java`, y `controllerManagedBean` los necesita para mandar a hacer los cálculos. Para tener disponible los atributos de un managed bean dentro de otro puede usarse el recurso de la "inyección de dependencias", que consiste en inyectar en un *managed bean*, el *managed bean* que contiene los datos requeridos. En la siguiente sección estudiamos la forma de hacerlo.

10.4.1 Inyección de un *Managed Bean* dentro de otro *Managed Bean*

Para inyectar un *Managed Bean* dentro de otro se utiliza la instrucción **@ManagedProperty** con la siguiente sintaxis:

```
@ManagedProperty(value="#{nombreDelBeanAInyectar}")
```

Posteriormente se agrega un atributo que debe ser de la clase del *bean* que se inyectó. La sintaxis es la siguiente:

```
private NombreManagedBean objetoManagedBean;
```

En nuestro ejemplo, para inyectar `formularioManagedBean` en `controllerManagedBean`, codificamos:

```
@ManagedProperty(value="#{formularioManagedBean}")
private FormularioManagedBean formularioManagedBean;
```

De esta manera, el controller (`controllerManagedBean`) tiene acceso a los datos del formulario mediante el objeto `formularioManagedBean`.

El código del `controllerManagedBean` es el siguiente:

```
package Controller;

import Model.Modelo;
import . . .


@ManagedBean( name="controllerManagedBean" )
@SessionScoped
public class ControllerManagedBean implements Serializable{

    // Inyección del Managed Bean
    @ManagedProperty(value="#{formularioManagedBean}")
    private FormularioManagedBean formularioManagedBean;

    private Double   sueldo;
    private int      numDias;
    private Modelo   calculador;
    private Double   salario;

    public ControllerManagedBean() {
          calculador = new Modelo( );
    }

    public Double getSalario() {
        return salario;
    }

    public FormularioManagedBean getFormularioManagedBean() {
        return formularioManagedBean;
    }

    public void setFormularioManagedBean(FormularioManagedBean
                                         formularioManagedBean) {
        this.formularioManagedBean = formularioManagedBean;
    }


    public void ejecutar(){
        try {
            sueldo = formularioManagedBean.getSueldo();
            numDias = formularioManagedBean.getNumDias();
```

Objeto para accesar los datos del formulario

Objeto que sirve para hacer el cálculo

Para proporcionar a la vista el salario

Acceso a los datos del *Managed Bean* inyectado

```
            // Se pide al model que calcule el salario
            salario = calculador.calculaSalario(sueldo, numDias);
            // Se redirecciona a la página "resultado"
            FacesContext.getCurrentInstance().getExternalContext()
                    .redirect("resultado.xhtml");
        } catch (IOException ex) {
           Logger.getLogger(ControllerManagedBean
                 .class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

10.4.2 El *bean* que se inyecta debe tener su *setter* y *getter*

Es importante subrayar que también es necesario hacer un *setter* y un *getter* para el objeto del bean que se inyecta, de lo contrario saldrá el error de la Figura X-3: "*No existe la propiedad formularioManagedBean para el bean administrado controllerManagedBean*".

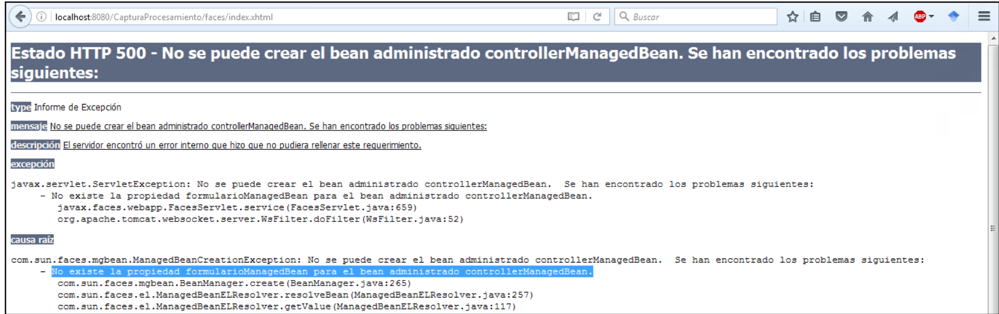


Figura X-3. Error cuando no se tienen el setter y getter del bean que se inyectó

Para poder desplegar datos y resultados del procesamiento, aprenderemos en la siguiente sección, a accesar los atributos de los *Managed Beans* desde un *facelet*.

10.5 Desplegar resultados: usando los datos de los *Managed Beans* en los *facelets*

El código del facelet que despliega los resultados “*resultado.xhtml*” es el siguiente:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
     <title>Resultado</title>
 </h:head>
 <h:body>
   <h2 style="color: blue;">Los datos que diste son
                               los siguientes:</h2>
     <span>Nombre: </span>
         <h:outputText value="#{formularioManagedBean.nombre}"
                       style=" width:25%;" />
     <br/><br/>

     <span>Sueldo: </span>
         <h:outputText value="#{formularioManagedBean.sueldo}"
                       style=" width:25%;" />
     <br/><br/>

     <span>Días trabajados: </span>
         <h:outputText value="#{formularioManagedBean.numDias}"
                       style=" width:25%;" />
     <br/><br/>

     <span>Los idiomas que seleccionaste son: </span>
         <h:outputText value="#{formularioManagedBean.idiomas}"
                       style=" width:25%;" />
     <br/><br/>

   <h2 style="color: blue;">El salario calculado es:</h2>
     <span>Salario total: </span>
        <h:outputText value="#{controllerManagedBean.salario}"
                      style=" width:25%;" />
     <br/><br/>

    </h:body>
</html>
```

Estos datos se obtienen del *Managed Bean*: formulario

Este dato se obtiene del *Managed Bean*: controller

La página *resultado.xhtml* tiene acceso a los datos de los dos *managed beans*: `controllerManagedBean` y `formularioManagedBean`. Esto se debe a que ambos *beans* tienen un alcance de sesión. Es muy importante tener en cuenta el alcance de los *beans*, el cual debe determinarse en función de la necesidad de cada aplicación en particular.

10.5.1 La elección del alcance de los *Managed Beans*

La elección del alcance depende de los datos que contiene el *managed bean*. Se usa `@RequestScoped` cuando sólo se despliegan datos en la página asociada. Se usa `@ViewScoped` cuando se incluyen elementos ajax en la página (validación ajax, diálogos, etc.). Si es necesario tener un mismo dato disponible durante toda la sesión, entonces utilizaremos `@SessionScoped`, por ejemplo: datos específicos del cliente, como el usuario y las preferencias de usuario (idioma, etc). Y se utiliza `@ApplicationScoped` para datos o constantes de uso común, como listas desplegables que son iguales para todos. También puede usarse para *beans* administrados que sólo tienen métodos, sin atributos.

Abusar del alcance `@ApplicationScoped` para los datos que son de sesión, vista, o solicitud (*request*), hará que los datos se compartan entre todos los usuarios, y que cualquiera pueda ver los datos de los demás. El uso abusivo de `@SessionScoped` para datos que son de vista (o *request*) hace que los datos perduren más tiempo del necesario y podría haber inconsistencias al navegar de una vista a otra porque se complica mantener los datos actualizados. Si usamos un bean `@RequestScoped` para datos que deberían ser `@viewScoped` entonces los datos se reinicializan al valor predeterminado en cada *postback* (ajax), y posiblemente no funcionarán bien los formularios.

10.5.2 Compatibilidad en el alcance de los *Managed Beans*: Disponibilidad del contenido

También es importante tener en cuenta la compatibilidad entre los *beans* en función de su alcance. Para ilustrar lo que sucede cuando hay incompatibildad en el alcance de los *beans*, cambiaremos el alcance de `FormularioManagedBean` de `@SessionScoped` a `@ViewScoped`. Con este cambio, los datos de desaparecen cuando se cambia a la página resultado.xhtml, y por lo tanto ya no se puede acceder a sus datos, por lo tanto, no se puede inyectar `FormularioManagedBean` dentro de `ControllerManagedBean`.

En la Figura X-4 se ilustra lo que sucede cuando el sistema intenta desplegar *resultado.xhtml*. El mensaje de error es el siguiente:

```
El ámbito del objeto al que hace referencia la expresión
#{formularioManagedBean}, view, es más corto que el
ámbito de session del bean administrado de referencia
(controllerManagedBean)
```

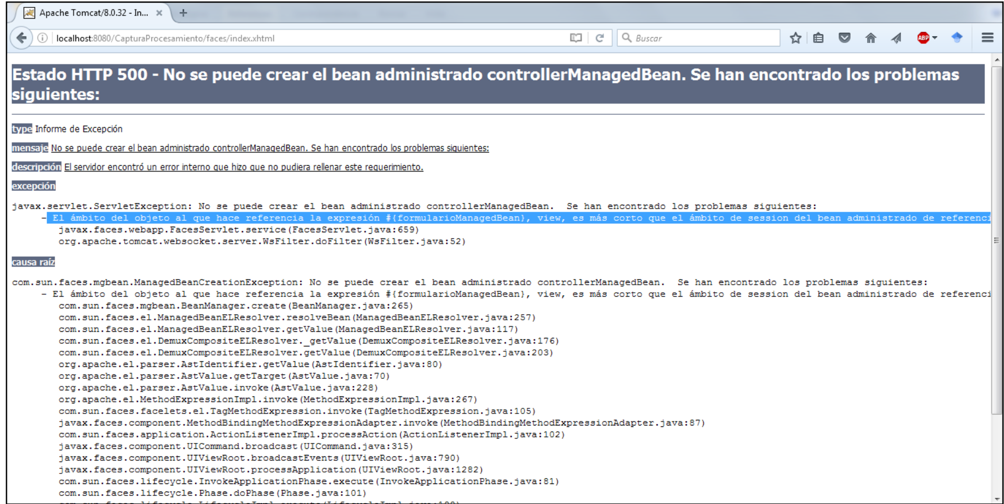


Figura X-4. Cuando dos managed bean tienen diferentes alcances surgen problemas

10.5.3 Usando los datos de un solo *managed Bean* en *resultado.xhtml*

También podemos pasar toda la información de `formularioManagedBean` al `controllerManagedBean`:

```
public void ejecutar(){
  try {
      nombre = formularioManagedBean.getNombre();
      sueldo = formularioManagedBean.getSueldo();
      numDias = formularioManagedBean.getNumDias();
      idiomas = formularioManagedBean.getIdiomas();
      salario = calculador.calculaSalario(sueldo, numDias);
      FacesContext.getCurrentInstance()
                  .getExternalContext()
                  .redirect("resultado.xhtml");
  } catch (IOException ex) {
      Logger.getLogger(ControllerManagedBean
                        .class.getName())
                        .log(Level.SEVERE, null, ex);
   }
}
```

Codificamos *getters* para los atributos del `controllerManagedBean`. Como sólo se desplegará la información (no se captura), no codificamos *setters*, ya que no se necesitan.

```
    public Double getSalario() {
        return salario;
    }

    public String getNombre() {
        return nombre;
    }

    public Double getSueldo() {
        return sueldo;
    }

    public int getNumDias() {
        return numDias;
    }

    public ArrayList <String> getIdiomas() {
        return idiomas;
    }
```

`formularioManagedBean` en la página *resultado.xhtml*. Por ejemplo, en lugar de:

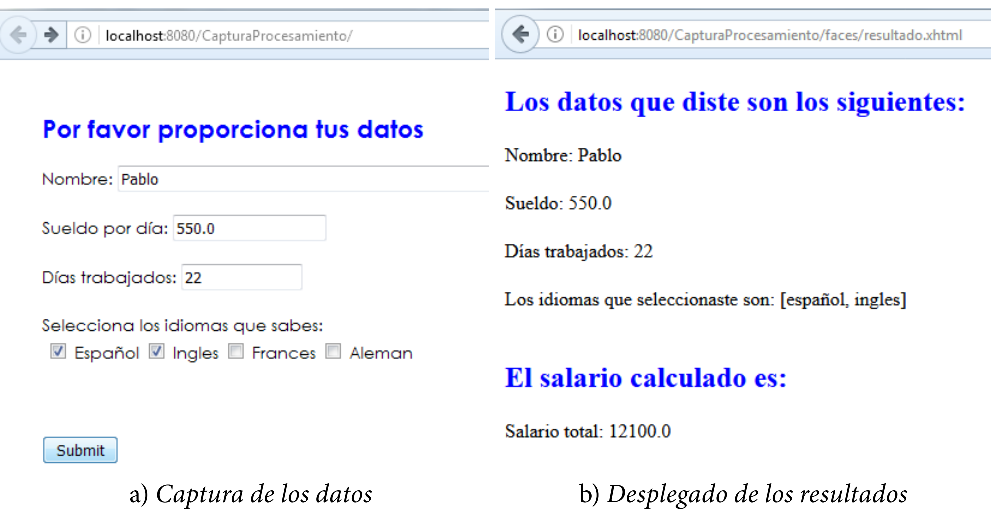
```
<h:outputText value="#{formularioManagedBean.nombre}"
              style=" width:25%;" />
```

Usamos:

```
<h:outputText value="#{controllerManagedBean.nombre}"
              style=" width:25%;" />
```

10.5.4 Ejecución en vivo del proyecto de ejemplo

Al final de la lección 6 de SEAWeb 2 podrás operar en vivo el ejemplo que hemos estudiado en este capítulo. En la figura X-5 se ilustra el funcionamiento del proyecto de ejemplo:



a) *Captura de los datos* b) *Desplegado de los resultados*

Figura X-5. Funcionamiento de captura y procesamiento de datos

10.6 Práctica

Construir una aplicación similar al ejercicio de la sección anterior, pero que solicite los puntos obtenidos en un videojuego y que de cómo resultado el nivel del jugador, de acuerdo con la siguiente tabla:

Número de puntos	Nivel
0-250	Principiante
251-600	Intermedio
601-900	Avanzado
901-1000	Experto

Al final de la lección 6 de SEAWeb 2 podrás operar en vivo esta práctica.

11. Proyecto JSF con Acceso a base de Datos

11.1 Objetivos

- Construir una aplicación Web JSF con acceso a base de datos.
- Construir una aplicación con las operaciones básicas: crear, leer, actualizar y borrar registros.

11.2 Proyecto "CRUD" con JSF

Se utiliza el acrónimo "CRUD" para nombrar a una aplicación con las funciones básicas de base de datos que son, por sus siglas en inglés: *Create*, *Read*, *Update*, *Delete* (crear, leer, actualizar y borrar). En esta sección construiremos una aplicación CRUD para administrar una base de datos que contiene las materias de Ingeniería en Computación.

11.3 Creación de la base de datos

Hacer la base de datos llamada *materiasIngComp* que contiene una tabla llamada "ueas". Recordar que para crear la base de datos se usa el comando:

```
> CREATE DATABASE materiasIngComp;
```

Después hay que entrar a la base de datos por medio del comando:

```
> USE materiasIngComp;
```

En la Figura XI-1 se muestran los comandos para crear la tabla "ueas", sus campos son:

- la clave de la materia: *clave*
- el nombre de la materia: *nombre*

- el trimestre en el que se imparte: *trimestre*
- la materia con la que está seriada: *requiere*

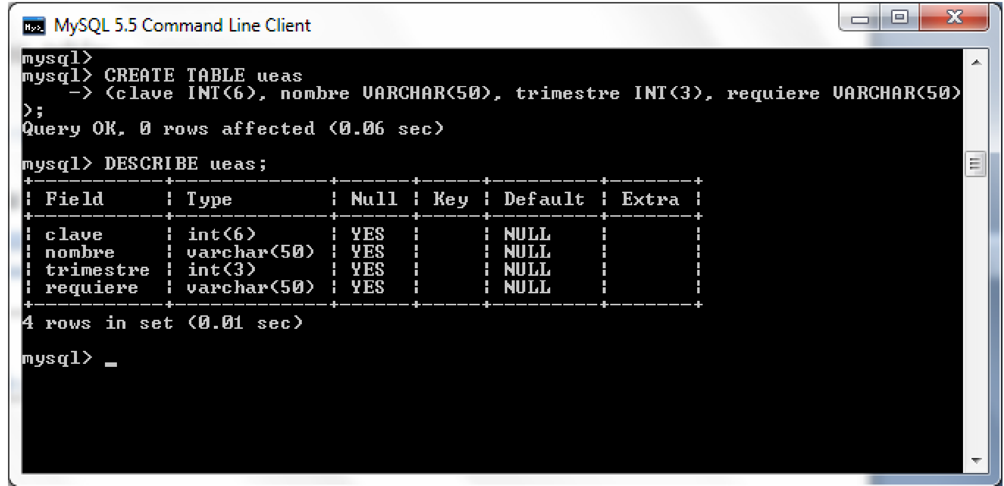


Figura XI-1. Tabla “ueas” de la base de datos “materiasIngComp”

Agregaremos manualmente dos datos a la tabla, con:

> INSERT INTO ueas VALUES(‘460006’, ‘Programacion Orientada a Objetos’, ‘2’, ‘Pro gramacion Estructurada’);

> INSERT INTO ueas VALUES(‘460009’, ‘Estructuras de Datos’ , ‘3’, ‘Programacion Orientada a Objetos’);

11.4 Creación del Proyecto

Ahora prepararemos nuestra aplicación para leer y desplegar lo que hay en la tabla “ueas”. Primero creamos un proyecto JSF llamado *UeasIngComp*, con los paquetes *Controller* y *Model*. En *Model* tenemos las clases `ConectaBD`, `GestorBD` y `Uea`. En el paquete *Controller* el managed bean `ControllerManagedBean`. En la Figura XI2 se muestra la fotografía del proyecto:

Recordar que es muy importante agregar en las librerías del proyecto, el conector a la base de datos.

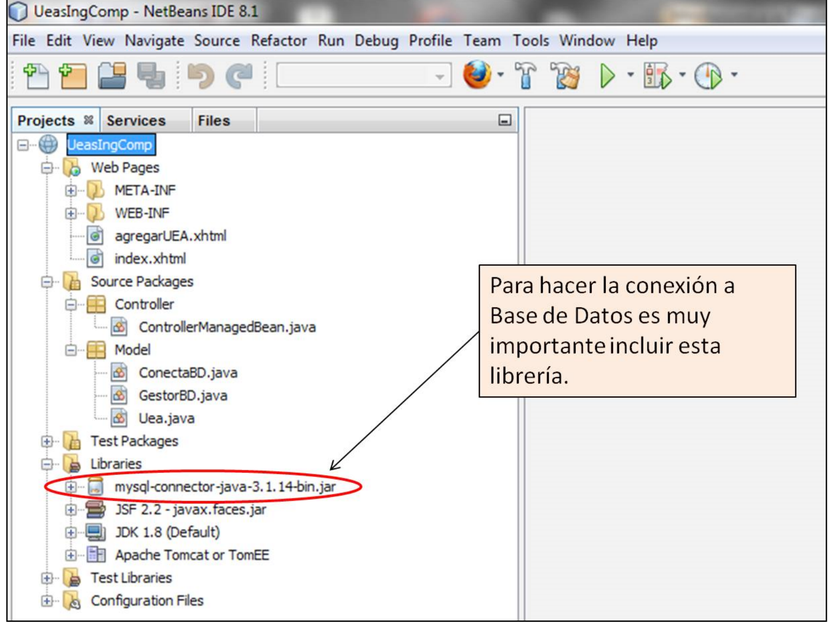


Figura XI-2. Fotografía del proyecto UeasIngComp

La clase `Uea.java` contiene los siguientes atributos:

```
private Integer clave;
private String  nombre;
private Integer trimestre;
private String  requisito;
```

con sus respectivos *setters* y *getters*. En las siguientes secciones analizaremos los demás componentes del proyecto.

11.5 Conexión con la base de datos

El primer paso es agregar el conector a la base de datos (mysql-connector-java) al proyecto. En la carpeta “Librerias” clic derecho “Add JAR/Folder…”. Se recomienda descargar el archivo.jar del conector en la carpeta NetBeansProjects o en un lugar que no olvidemos posteriormente.

Dentro del paquete *Model* del proyecto, creamos la clase `ConectaBD`, la cual contiene el código para conectarse a la base de datos llamada “*MateriasIngComp*”. Su constructor establece la conexión con la base de datos. El método `getConexion()` regresa el objeto `conexion`

de clase `Connection`. Y el método `cerrar()` se asegura de que la conexión quede cerrada. A continuación, presentamos el código de la clase `ConectaBD`:

```
package Model;

import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import javax.faces.context.FacesContext;

public class ConectaBD {
    private Connection conexion=null;
    private String servidor="localhost";
    private String database= "materiasingcomp";
    private String usuario="root";
    private String password="ueadb01";
    private String url="";

    public ConectaBD(){
        try {
            // Establece la conexión con la base de datos
            Class.forName("com.mysql.jdbc.Driver");
            url="jdbc:mysql://"+servidor+"/"+database;
            conexion=DriverManager.getConnection(url, usuario,
                                                 password);
        }
        catch (SQLException | ClassNotFoundException ex) {
            try {
                System.out.println(ex);

            } catch (IOException ex1) {
                System.out.println(ex1);
            }
        }
    }
public Connection getConexion(){
        return conexion;
    }

    public Connection cerrarConexion(){
        try {
            conexion.close();
        } catch (SQLException ex) {
            System.out.println(ex);
        }
        conexion=null;
        return conexion;
    }
}
```

11.6 Leer de la base de datos

Para leer la lista de elementos de la tabla *UEA*, creamos la clase `GestorBD`, la cual contiene un método para leer una lista de *UEA*, como se muestra a continuación:

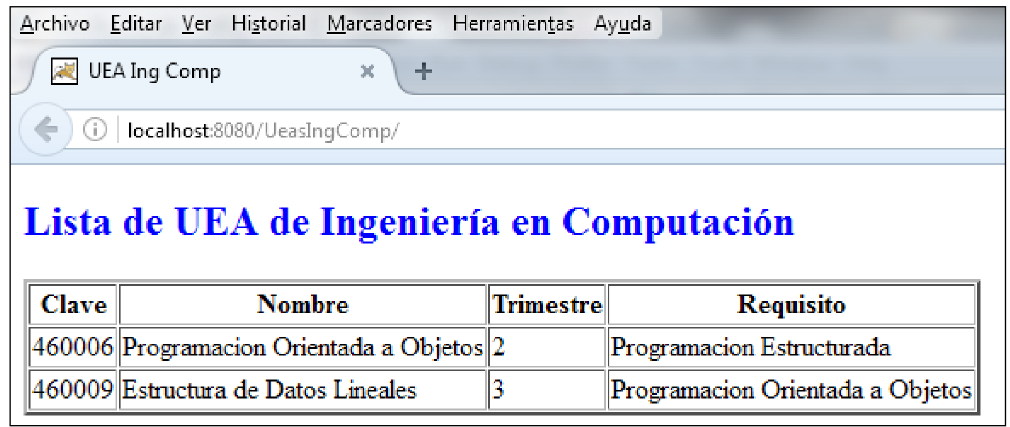
```
package Model;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;

public class GestorBD {
    private Connection conexion=null;
    private Statement stm = null;
    private ResultSet ueaResultSet;
    private Integer clave, trimestre;
    private String nombre, requisito;

    public ArrayList<Uea> leerUeas(){
      ArrayList<Uea> ueas = new ArrayList<Uea>();
      Uea ueaHallada;
      try{
          ConectaBD conectaBD = new ConectaBD();
          conexion = conectaBD.getConexion();
          stm = conexion.createStatement();
          ueaResultSet = stm.executeQuery("select * from ueas");
          if(!ueaResultSet.next()){
             System.out.println(" No se encontraron registros");
             conexion.close();
           return null;
          }else{
            do{
              clave = ueaResultSet.getInt("clave");
              nombre = ueaResultSet.getString("nombre");
              trimestre = ueaResultSet.getInt("trimestre");
              requisito = ueaResultSet.getString("requiere");
              ueaHallada =
                        new Uea(clave,nombre,trimestre,requisito);
              ueas.add(ueaHallada);
          }while(ueaResultSet.next());
          conexion.close();
          return ueas;
         }
      }catch(Exception e){
        System.out.println("Error en la base de datos.");
        e.printStackTrace();
        return null;}
       }
}
```

Regresa una lista con las UEA que encontró en la base de datos

Instrucción para la base de datos

11.6.1 Desplegando los registros leidos en la base de datos

En la Figura XI-3 se muestra la vista que se le presentará al usuario. Ésta despliega una tabla con todos los registros encontrados en la base de datos.



Clave	Nombre	Trimestre	Requisito
460006	Programacion Orientada a Objetos	2	Programacion Estructurada
460009	Estructura de Datos Lineales	3	Programacion Orientada a Objetos

Figura XI-3. Vista de los datos obtenidos de la tabla UEAS

El código del *facelet* (*index.xhtml*) de la Figura XI3 contiene el *tag* `<h:dataTable`, cuya sintaxis es la siguiente:

```
<h:form>
       <h:dataTable value="#{nombreManagedBean.nombreLista}"
                    var="claseDelElemento" >

           <h:column>
              <f:facet name="header">Encabezado 1</f:facet>
              #{claseDelElemento.atributo1}
           </h:column>

           . . .

           <h:column>
              <f:facet name="header">Encabezado N </f:facet>
              #{claseDelElemento.atributoN}
           </h:column>

        </h:dataTable>

   </h:form>
```

Nótese que `<h:dataTable` debe ir dentro de un `<h:form`. A continuación, el código de *index.xhtml*:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
    <h:head>
         <title>UEA Ing Comp</title>
    </h:head>

    <h:body>
      <h2 style="color: blue;">Lista de UEA de Ingeniería en
                                                Computación</h2>

      <h:form>
        <h:dataTable value="#{controllerManagedBean.ueasList}"
                     var="uea"  border="2"
                     cellspacing="1" cellpadding="1">

             <h:column>
                <f:facet name="header">Clave</f:facet>
                #{uea.clave}
             </h:column>

             <h:column>
                <f:facet name="header">Nombre</f:facet>
                #{uea.nombre}
             </h:column>

             <h:column>
                <f:facet name="header">Trimestre</f:facet>
                #{uea.trimestre}
             </h:column>

             <h:column>
                <f:facet name="header">Requisito</f:facet>
                #{uea.requisito}
             </h:column>

        </h:dataTable>

      </h:form>
    </h:body>
</html>
```

11.6.2 El **`controllerManagedBean`** pide al model que lea la lista de *UEA* en la base de datos

Finalmente, sólo falta el `controllerManagedBean`, el cual se muestra a continuación:

```
package Controller;

import Model.GestorBD;
import Model.Uea;
import java.util.ArrayList;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "controllerManagedBean")
@SessionScoped
public class ControllerManagedBean {

    private Integer clave;
    private String  nombre;
    private Integer trimestre;
    private String  requisito;
    private GestorBD gestorBD;

    private static ArrayList<Uea> ueasList;

    public ControllerManagedBean() {
        gestorBD = new GestorBD();
        ueasList = gestorBD.leerUeas();
    }

    public Integer getClave() {
        return clave;
    }

    public String getNombre() {
        return nombre;
    }

    public Integer getTrimestre() {
        return trimestre;
    }

    public String getRequisito() {
        return requisito;
    }

    public ArrayList<Uea> getUeasList() {
        return ueasList;
    }
}
```

La lista de UEAs se obtiene del gestor de la base de datos

Sólo codificamos *getters*, porque los *setters* aún no se necesitan

11.7 Crear un elemento en la base de datos

En la Figura XI-4 añadimos el botón “Agregar UEA” mediante:

```
<h:commandButton value=”Agregar UEA”
                 type=”submit”
               action=”#{controllerManagedBean
                                .pedirDatosUEA_aAgregar()}” >
```

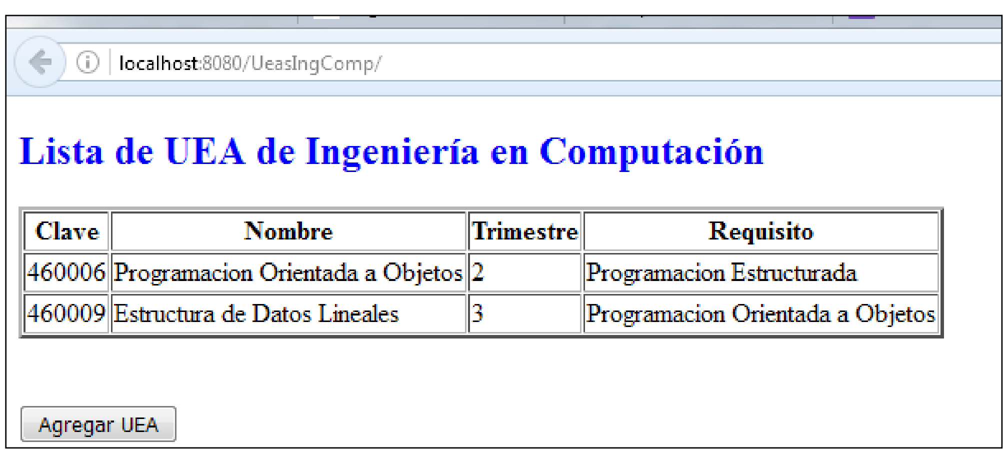


Figura XI-4. Vista con el botón “Agregar Uea”

11.7.1 La página que captura los datos de la nueva *UEA*

El método `pedirDatosUEA_aAgregar()` de `ControllerManagedBean` redirecciona a la página *agregar_UEA.xhtml* (Figura XI-5) para que el usuario introduzca los datos de la nueva *UEA*:

localhost:8080/UeasIngComp/faces/agregarUEA.xhtml

Proporciona los datos de la UEA nueva

Clave: 460022

Nombre: Analisis de Requerimientos

Trimestre: 6

Requisito: Fundamentos de Ing. de SW

Submitir

Figura XI-5. Vista agregar_UEA.xhtml para que el usuario proporcione los datos de la UEA nueva

El código de *agregar_UEA.xhtml* es el siguiente:

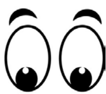
```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
     <title>Captura de UEA</title>
 </h:head>
  <h:body>
    <h:form id="form" style="width: 100%; padding: 20px 33px;">
     <h2 style="color: blue;">Proporciona los datos de la
                                                     UEA nueva</h2>
       <span>Clave: </span>
         <h:inputText value="#{controllerManagedBean.clave}"
                      style=" width:25%;" />
        <br/><br/>

        <span>Nombre: </span>
         <h:inputText value="#{controllerManagedBean.nombre}"
                      style=" width:25%;" />
        <br/><br/>

        <span>Trimestre: </span>
         <h:inputText value="#{controllerManagedBean.trimestre}"
                      style=" width:9%;" />
        <br/><br/>

        <span>Requisito: </span>
         <h:inputText value="#{controllerManagedBean.requisito}"
                      style=" width:25%;" />
        <br/><br/>
        <br/><br/>
          <h:commandButton value="Submitir" type="submit"
                  action="#{controllerManagedBean.guardarUEA()}" >
          </h:commandButton>
    </h:form>
 </h:body>
</html>
```

Método que guarda la UEA en la base de datos

11.7.2 El método `pedirDatosUEA _ aAgregar()` del controller

El método `pedirDatosUEA _ aAgregar()` de `ControllerManagedBean` se muestra a continuación:

```
public void pedirDatosUEA_aAgregar (){
   try{
       FacesContext.getCurrentInstance()
                   .getExternalContext()
                 .redirect("agregar_UEA.xhtml");
}catch(IOException ex) {
     Logger.getLogger(ControllerManagedBean
                 .class.getName()).log(Level.SEVERE, null, ex);
  }
}
```



Para poder capturar los datos del formulario en el `ControllerManagedBean`, ahora sí necesitaremos agregar los métodos *setters* para cada uno de los campos del formulario.

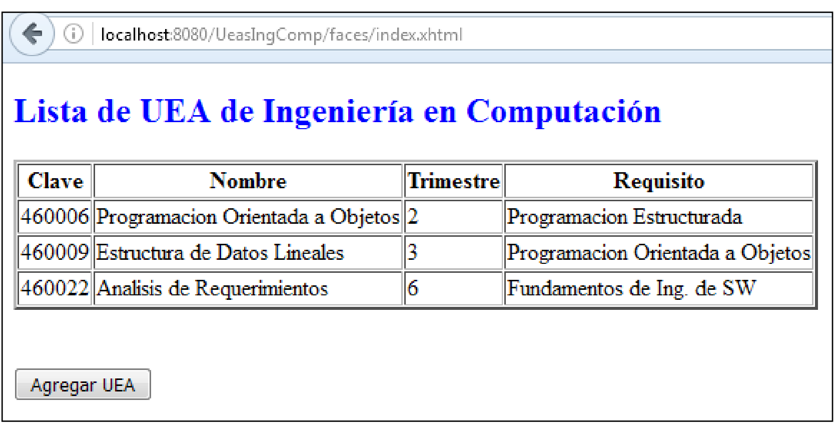
11.7.3 El método para guardar la UEA en el Controller

Ahora codificaremos `guardarUEA()` dentro del `ControllerManagedBean`: Desplegando la lista actualizada de UEA

```
    public void guardarUEA(){

        Uea ueaNueva = new Uea(clave, nombre,
                               trimestre, requisito);

        if (gestorBD.guardarUea(ueaNueva)){
          try{
             ueasList = gestorBD.leerUeas();

             FacesContext.getCurrentInstance()
                         .getExternalContext()
                         .redirect("index.xhtml");
          }catch(IOException ex) {
             Logger.getLogger(ControllerManagedBean
                   .class.getName()).log(Level.SEVERE, null, ex);
          }
        }
    }
```

11.7.4 Desplegando la lista actualizada de UEA

En la Figura XI-6 se observa cómo se despliega la lista actualizada con la nueva *UEA* que agregó el usuario:



Clave	Nombre	Trimestre	Requisito
460006	Programacion Orientada a Objetos	2	Programacion Estructurada
460009	Estructura de Datos Lineales	3	Programacion Orientada a Objetos
460022	Analisis de Requerimientos	6	Fundamentos de Ing. de SW

Figura XI-6. Lista de UEA actualizada

11.7.5 El método para guardar la UEA en el model

Finalmente veamos cómo se codifica el método `guardarUea()` en `gestorBD`:

```
private int resultUpdate = 0;

public boolean guardarUea(Uea ueaNueva) {
        try{
          ConectaBD conectaBD = new ConectaBD();
          conexion = conectaBD.getConexion();
          stm = conexion.createStatement();
          resultUpdate = stm.executeUpdate("INSERT INTO ueas
            VALUES("
                    +ueaNueva.getClave()
                    +",'" + ueaNueva.getNombre()
                    +"'," +ueaNueva.getTrimestre()
                    +",'"+ueaNueva.getRequisito()+ "');");
          if(resultUpdate != 0){
             conexion.close();
             return true;
          }else{
             conexion.close();;
             System.out.println("No se pudo insertar la UEA.");
             return false;
           }
        }catch (Exception e) {
            System.out.println("Error en la base de datos.");
            e.printStackTrace();
            return false;
     }
  }
```

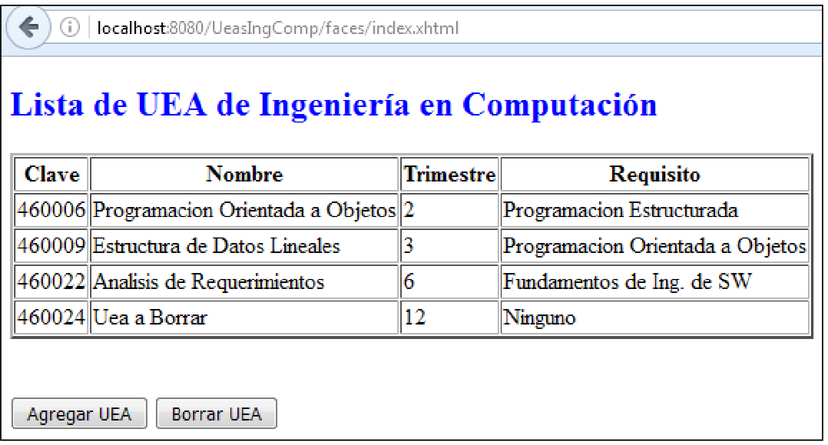
Lo agregamos a los atributos de la clase

11.8 Borrar un elemento en la base de datos

En la Figura XI-1se observa que añadimos el botón "Borrar UEA" mediante:

```
<h:commandButton value="Borrar UEA"
                 type="submit"
               action="#{controllerManagedBean
                                .pedirDatosUEA_aBorrar()}" >
```

Si lo ponemos entre   el botón "Borrar UEA" queda alineado con el botón "Agregar UEA", como se muestra en la Figura XI-7:



Clave	Nombre	Trimestre	Requisito
460006	Programacion Orientada a Objetos	2	Programacion Estructurada
460009	Estructura de Datos Lineales	3	Programacion Orientada a Objetos
460022	Analisis de Requerimientos	6	Fundamentos de Ing. de SW
460024	Uea a Borrar	12	Ninguno

Figura XI-7. Vista con los botones "Agregar UEA" y "Borrar UEA"

11.8.1 La página que captura los datos de la UEA a eliminar

El método `pedirDatosUEA_aBorrar()` redirecciona a borrar_UEA.xhtml (Figura XI-8) para que el usuario introduzca la clave y el nombre de la *UEA* que quiere borrar:

Figura XI-8. Vista borrar_UEA.xhtml para que el usuario ingrese clave y nombre de la UEA a borrar

El código de *borrar_UEA.xhtml* es el siguiente:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
     <title>Borrar UEA</title>
 </h:head>
 <h:body>
   <h:form id="form" style="width: 100%;padding: 20px 33px;">
     <h2 style="color: blue;">Proporciona clave y nombre de la UEA
                          a eliminar</h2>
     <span>Clave: </span>
        <h:inputText value="#{controllerManagedBean.clave}"
                     style=" width:25%;" />
        <br/><br/>

     <span>Nombre: </span>
        <h:inputText value="#{controllerManagedBean.nombre}"
                     style=" width:25%;" />
        <br/><br/>

        <br/><br/>
          <h:commandButton value="Submitir" type="submit"
                   action="#{controllerManagedBean.borrarUEA()}" >
          </h:commandButton>
      </h:form>
    </h:body>
</html>
```

11.8.2 El método pedirDatosUEA _ aBorrar() del controller

El método pedirDatosUEA _ aBorrar() de ControllerManagedBean se muestra a continuación:

```
public void pedirDatosUEA_aBorrar (){
   try{
       FacesContext.getCurrentInstance()
                    .getExternalContext()
                    .redirect("borrar_UEA.xhtml");
}catch(IOException ex) {
     Logger.getLogger(ControllerManagedBean
                  .class.getName()).log(Level.SEVERE, null, ex);
  }
}
```

11.8.3 El método para borrar la UEA en el Controller

Ahora codificaremos borrarUEA() dentro del ControllerManagedBean:

```
public void borrarUEA(){

        Uea ueaABorrar = new Uea(clave, nombre,
                             trimestre, requisito);

        if (gestorBD.borrarUea(ueaABorrar)){
          try{
             ueasList = gestorBD.leerUeas();

             FacesContext.getCurrentInstance()
                         .getExternalContext()
                         .redirect("index.xhtml");
          }catch(IOException ex) {
             Logger.getLogger(ControllerManagedBean
                   .class.getName()).log(Level.SEVERE, null, ex);
          }
        }
    }
```

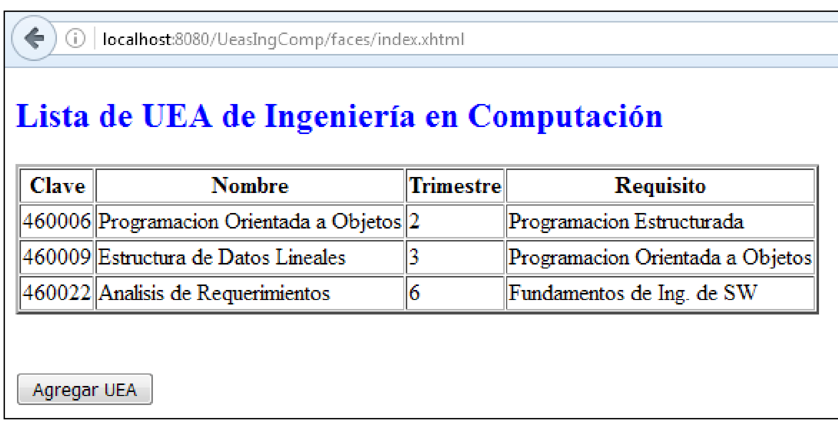
No importa que trimestres y requisisto estén en *Null*, porque no se necesitan

Este método de gestorBD borra la UEA solicitada

Hay que actualizar la lista de ueas antes de volver a desplegarla

11.8.4 Desplegando la lista actualizada de UEA

En la Figura XI-9 se observa como se despliega la lista actualizada y ya no aparece la *UEA* que borró el usuario:



Clave	Nombre	Trimestre	Requisito
460006	Programacion Orientada a Objetos	2	Programacion Estructurada
460009	Estructura de Datos Lineales	3	Programacion Orientada a Objetos
460022	Analisis de Requerimientos	6	Fundamentos de Ing. de SW

Figura XI-9. Lista de UEA actualizada

11.8.5 El método para borrar la UEA en el model

Finalmente veamos cómo se codifica el método `borrarUea()` en `gestorBD`:

```
private int resultUpdate = 0;     // Es uno de los atributos de la clase

public boolean borrarUea(Uea ueaABorrar) {
 try{
    ConectaBD conectaBD = new ConectaBD();
    conexion = conectaBD.getConexion();
    stm = conexion.createStatement();
    resultUpdate = stm.executeUpdate("DELETE FROM ueas
                    WHERE(clave = '"
                      +ueaABorrar.getClave()
                      +"' AND nombre ='"
                      +ueaABorrar.getNombre()+"');");
    if(resultUpdate != 0){
       conexion.close();
       return true;
    }else{
       conexion.close();;
       System.out.println("No se pudo borrar la UEA.");
       return false;
    }
 }catch (Exception e) {
    System.out.println("Error en la base de datos.");
    e.printStackTrace();
    return false;
 }
}
```

Es uno de los atributos de la clase

11.9 Modificar un elemento en la base de datos

En la Figura XI-10 añadimos el botón “Modificar UEA” mediante:

```
<span>
 <h:commandButton value="Modificar UEA"
                  type="submit"
                action="#{controllerManagedBean
                             .pedirDatosUEA_aLocalizar()}">
</span>
```

En la Figura XI-10 se observa la vista que incluye también el botón “Modificar UEA”. El método `pedirDatosUEA_aLocalizar()` redirecciona a la página *localizar_UEA.xhtml* en la que se pide la clave y el nombre de la *UEA* a modificar.

localhost:8080/UeasIngComp/

Lista de UEA de Ingeniería en Computación

Clave	Nombre	Trimestre	Requisito
460006	Programacion Orientada a Objetos	2	Programacion Estructurada
460009	Estructura de Datos Lineales	3	Programacion Orientada a Objetos
460022	Analisis de Requerimientos	6	Fundamentos de Ing. de SW
460000	Uea a Modificar	1	Ninguno

Agregar UEA Borrar UEA Modificar UEA

Figura XI-10. Vista con los botones “Agregar UEA”, “Borrar UEA” y “Modificar UEA”

11.9.1 Validar la existencia de la *UEA* a modificar

Antes de intentar modificar una UEA, es necesario comprobar su existencia en la base de datos. Al seleccionar el botón “Modificar UEA” se despliega la página de la Figura XI-11 (*localizar_UEA.xhtml*) solicitando la clave y el nombre de la *UEA* a modificar. El código de *localizar_UEA.xhtml* es muy similar al de las vistas de los ejemplos anteriores, su botón submitir está codificado de la siguiente forma:

```
<h:commandButton value="Submitir" type="submit"
```

```
                    action="#{controllerManagedBean.localiza-
rUEA()}" >
   </h:commandButton>
```

Proporciona clave y nombre de la UEA a modificar

Clave: 460035

Nombre: Analisis de Algoritmos

Submitir

Figura XI-11. Vista que solicita la clave y nombre de la UEA a modificar

El código de `localizarUEA()` en `controllerManagedBean` es el siguiente:

```
public void localizarUEA(){
 if(gestorBD.localizaUEA(clave,nombre))
     try{
          FacesContext.getCurrentInstance()
                      .getExternalContext()
                      .redirect("modificar_UEA.xhtml");
     }catch(IOException ex) {
          Logger.getLogger(ControllerManagedBean
                   .class.getName()).log(Level.SEVERE, null, ex);
          }
  else
     try{
           ueasList = gestorBD.leerUeas();
           FacesContext.getCurrentInstance()
                       .getExternalContext()
                       .redirect("error.xhtml");
     }catch(IOException ex) {
           Logger.getLogger(ControllerManagedBean
                 .class.getName()).log(Level.SEVERE, null, ex);
     }
}
```

Si en la base de datos no hay una UEA con la clave y nombre proporcionados, se despliega el mensaje de error de la Figura XI-12; al dar clic en "Ok" la aplicación muestra nuevamente la página de inicio: "*index.xhtml*".

Figura XI-12. Mensaje de error

El código de "*error.xhtml*" se muestra a continuación:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
     <title>Borrar UEA</title>
 </h:head>
 <h:body>
   <h:form id="form" style="width: 100%;padding: 20px 33px;">
                
                
     <img src="#{request.contextPath}/images/error.jpeg"
           width="80"  height="80"/>

     <h2 style="color: blue;">ERROR, no se encontró una UEA
                                                    con:</h2>
      <span>Clave: </span>
         <h:outputText value="#{controllerManagedBean.clave}"
                             style=" width:25%;" />
      <br/><br/>

      <span>Nombre: </span>
         <h:outputText value="#{controllerManagedBean.nombre}"
                             style=" width:25%;" />
      <br/><br/>
      <span>
         <h:commandButton value="OK" action="index" />
      </span>

   </h:form>
 </h:body>
</html>
```

Instrucción para que encuentre la ruta relativa donde está la imagen

Regresa al inicio

(En este ejemplo añadimos una imagen que está localizada dentro de la carpeta "images" que a su vez están en la carpeta "Web Pages").

11.9.2 El método en el model para localizar la UEA

El código de `localizaUEA()` en `gestorBD`, es el siguiente:

```
public boolean localizaUEA(Integer clave, String nombre) {
 try{
    ConectaBD conectaBD = new ConectaBD();
    conexion = conectaBD.getConexion();
    stm = conexion.createStatement();
    ueaResultSet = stm.executeQuery("SELECT * FROM ueas
                                    WHERE(clave = '"
                                        +clave+"' AND nombre ='"
                                        +nombre+"');");
    if(!ueaResultSet.next()){
       System.out.println(" No se encontraron registros");
       conexion.close();
       return false;
    }else{
       conexion.close();
       return true; }
 }catch(Exception e){
    System.out.println("Error en la base de datos.");
    e.printStackTrace();
    return false;
 }
}
```

11.9.3 La página que captura los datos de la UEA a modificar

En caso de que si se localice en la base de datos un registro con la clave y el nombre proporcionados, entonces el método `localizarUEA()` redirecciona a la página modificar_UEA.xhtml para que el usuario introduzca las modificaciones que desea. En este ejercicio, el usuario podrá modificar cualquier campo excepto la clave de la UEA. Si requiere cambiar la clave será necesario borrar primero la *UEA* y después darla de alta con la clave correcta. Por esta razón, en la Figura XI-3 se observa que la `clave` se despliega con un `<h:outputText`:

Figura XI-13. Vista modificar_UEA.xhtml para que el usuario ingrese los nuevos datos de la UEA

El botón “submitir” de *modificar_UEA.xhtml* contiene el siguiente código:

```
<h:commandButton value=”Submitir” type=”submit”
                action=”#{controllerManagedBean.modificar
UEA()}”>
</h:commandButton>
```

11.9.4 El método para modificar la UEA en el Controller

El método `modificarUEA()` dentro del `ControllerManagedBean` se muestra a continuación:

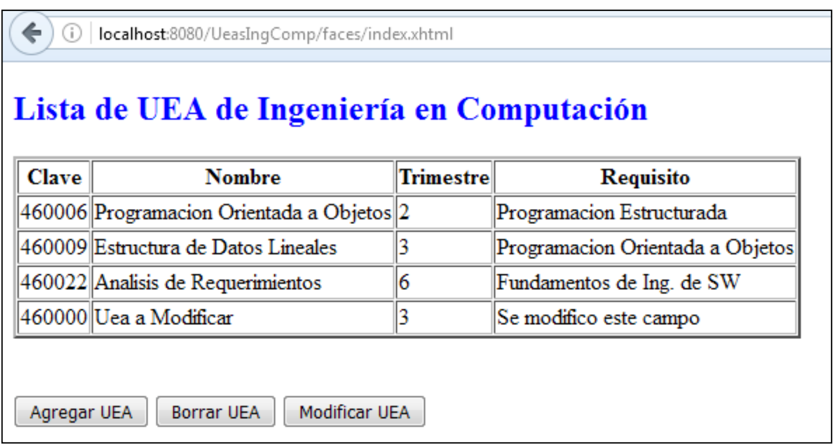
```
public void modificarUEA(){

    Uea uea_a_Cambiar = new Uea(clave, nombre,
                                trimestre, requisito);

    if (gestorBD.modificarUea(uea_a_Cambiar)){
      try{
         ueasList = gestorBD.leerUeas();
         FacesContext.getCurrentInstance().getExternalContext()
                     .redirect("index.xhtml");
      }catch(IOException ex) {
         Logger.getLogger(ControllerManagedBean
               .class.getName()).log(Level.SEVERE, null, ex);}
    }
}
```

Los datos están actualizados conforme a los que se capturó en *modificar_UEA.xhtml*

Este método de `gestorBD` modifica la UEA solicitada

11.9.5 Desplegando la lista actualizada de UEA

En la Figura XI-14 se observa cómo se despliega la lista actualizada, la última *UEA* contiene los cambios que solicitó el usuario:



Clave	Nombre	Trimestre	Requisito
460006	Programacion Orientada a Objetos	2	Programacion Estructurada
460009	Estructura de Datos Lineales	3	Programacion Orientada a Objetos
460022	Analisis de Requerimientos	6	Fundamentos de Ing. de SW
460000	Uea a Modificar	3	Se modifico este campo

Figura XI-14. Lista de UEA con la última actualizada

11.9.6 El método para actualizar la UEA en el model

Finalmente veamos cómo se codifica el método `modificarUea()` en `gestorBD`:

```
public boolean modificarUea(Uea ueaACambiar) {
 try{
    ConectaBD conectaBD = new ConectaBD();
    conexion = conectaBD.getConexion();
    stm = conexion.createStatement();
    resultUpdate = stm.executeUpdate("UPDATE ueas SET nombre = '"
                  +ueaACambiar.getNombre()
                  +"', trimestre = '"+ ueaACambiar.getTrimestre()
                  +"', requiere = '"+ ueaACambiar.getRequisito()
                  +"' WHERE clave = "
                  +ueaACambiar.getClave()+";");
    if(resultUpdate != 0){
       conexion.close();
       return true;
    }else{
       conexion.close();;
       System.out.println("No se pudo borrar la UEA.");
       return false;
    }
 }catch (Exception e) {
    System.out.println("Error en la base de datos.");
    e.printStackTrace();
    return false;
 }
}
```

No lleva paréntesis

No lleva comilla (')

11.10 Práctica

Hacer una aplicación Web para la gestión de videojuegos.

1.- En la página de inicio se debe desplegar la lista de videojuegos registrados en una base de datos. Los campos de cada registro se muestran en la Figura XI-15:

Figura XI-15. Práctica 1: Registro de videojuegos

1.a).- Crear una base de datos en MySQL llamada "videojuegos" con una tabla llamada "existencias". Los campos de la tabla existencias son los siguientes:

- clave int
- nombre varchar
- genero varchar
- plataforma varchar
- precio varchar

1.b).- Con el botón "Agregar Nuevo" se podrá agregar un nuevo registro en la base de datos. En la Figura XI-16 se muestra la página de captura del registro de un nuevo videojuego.

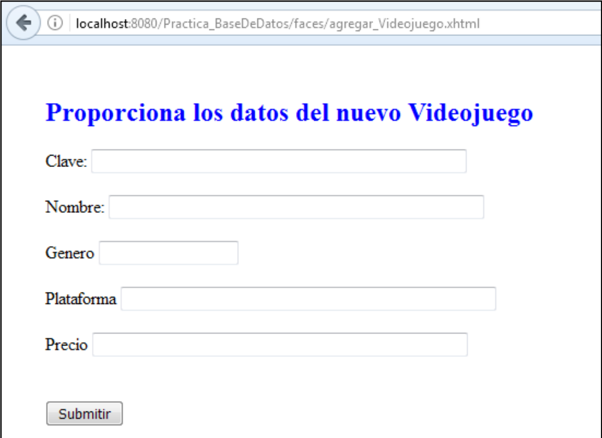


Figura XI-16. Captura de un nuevo videojuego

1.c).- Con el botón "Borrar" se podrá borrar un registro de la base de datos. Para borrar un registro, primero se debe desplegar una página para pedir la clave y el nombre del videojuego, como se muestra en la Figura XI-17:

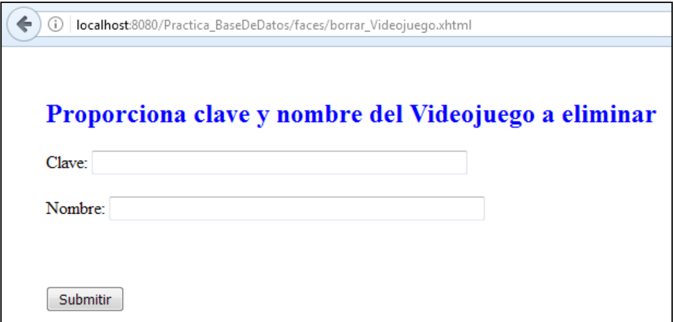


Figura XI-17: Solicitud de la clave y el nombre del videojuego que se quiere borrar

1.d).- Con el botón "Modificar" se podrán modificar los datos de un registro existente. Primero será necesario solicitar la clave y el nombre para validar que el registro se encuentre en la base de datos, como se muestra en Figura XI-18:

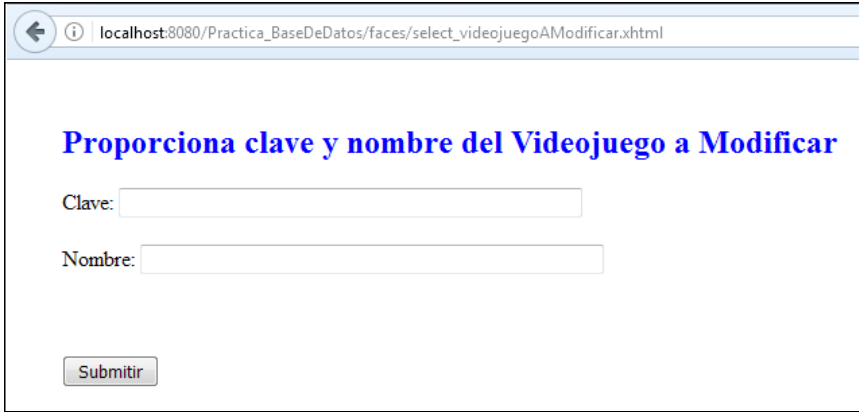


Figura XI-18. Página que pide los datos del videojuego que se quiere modificar

En caso de que éste no se encuentre, se debe desplegar una página con un mensaje de error, indicando que no se encontró el registro. Al seleccionar el botón “Ok” se despliega nuevamente la página de inicio.

Al final de la lección 6 de SEAWeb 2 podrás operar en vivo esta práctica.

12. Validadores y existencia de librerías adicionales

12.1 Objetivos

- Trabajar con los validadores de JSF.
- Conocer la existencia de librerías adicionales que mejoran la presentación de las páginas Web.

12.2 Los validadores

Los validadores evitan que surjan errores de procesamiento debido a datos incompletos o con formato incorrecto que introduce el usuario. Para usar los validadores de *JavaServer Faces* es necesario incluir la librería:

```
xmlns:f="http://java.sun.com/jsf/core"
```

Cuando la validación no es exitosa se despliega un mensaje de error predeterminado.

12.2.1 Campo obligatorio

Cuando es indispensable que el usuario introduzca el dato en un campo, sólo hay que agregar el indicador required=true en el campo de texto, por ejemplo:

```
<h:inputText id="numeroCuenta"
    value=»#{loginBean.numeroCuenta}"
    required=»true»/>
```

Cuando el usuario deja vacío un campo de texto marcado como obligatorio, se despliega un mensaje de texto por *default*. Para el campo "Nombre" del ejemplo de la Figura XII1 se codificó:

```
<h:inputText id="nombre" value="#{registro.nombre}"
             required="true" />
```

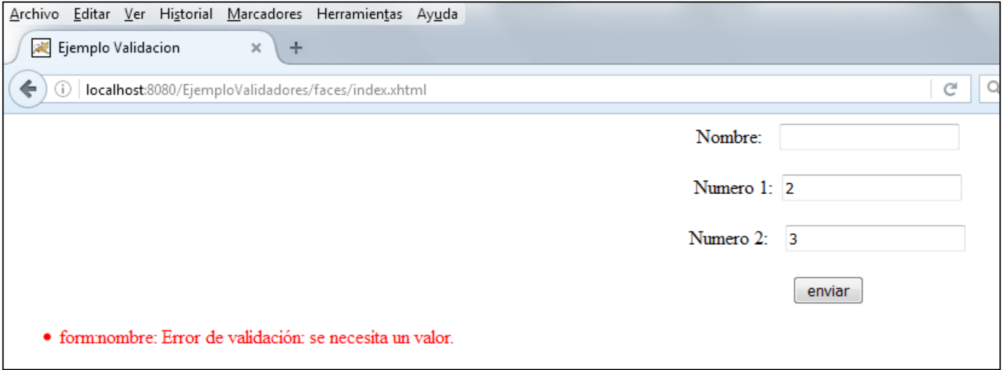


Figura XII-1. Error cuando el usuario no introduce un dato que es obligatorio

12.2.2 Validador de longitud

Determina si un campo de texto contiene un número máximo y/o mínimo de caracteres. En el ejemplo de la Figura XII-2 se codificó:

```
<h:inputText id=”num2”  value=”#{registro.num2}”>
   <f:validateLength maximum=”2”/>
</h:inputText>
```

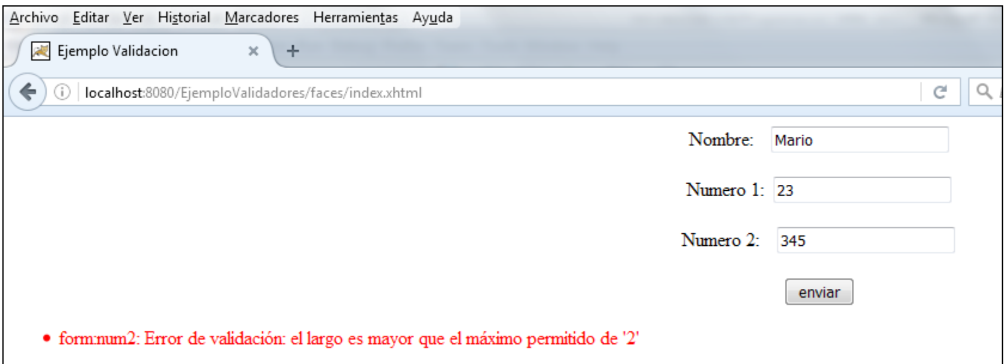


Figura XII-2. Error cuando el usuario pone más dígitos de los permitidos

Para delimitar la longitud de un campo de texto de entrada sin necesidad de validadores se puede utilizar la propiedad `maxLength`. Cuando se utiliza esta propiedad, el usuario no puede introducir más caracteres de los que se definen en la propiedad `maxLength`.

12.2.3 Validador de rango

Determina si una entrada numérica entra dentro de un rango aceptable. Los validadores estándar de JSF que sirven para validar el rango son:

Tag de JSF	Atributos	Tipo de Dato
validateDoubleRange	minimum, maximum	Para un Double
validateLongRange	minimum, maximum	Debe ser un entero

En la Figura XII-3 presentamos un ejemplo con tres errores. En el primer campo se proporcionó un número mayor que 100. El segundo campo sólo admite números enteros y en este ejemplo contiene un número real, y el tercer campo tiene un valor fuera del rango especificado.

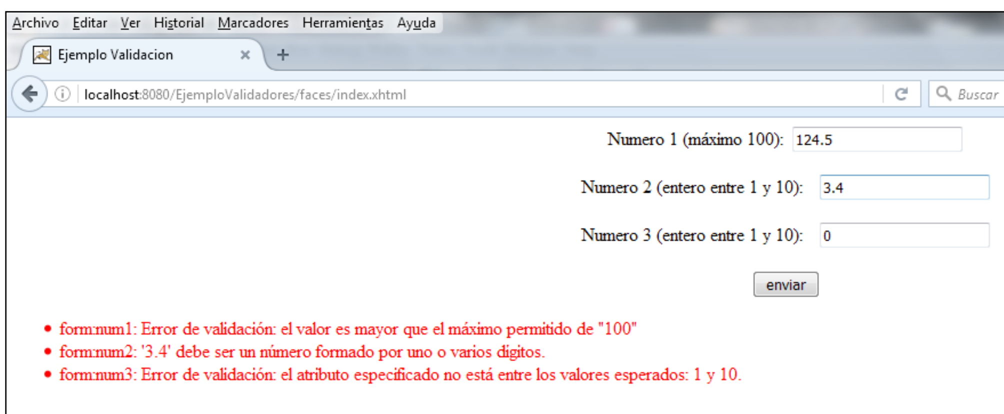


Figura XII-3. Errores en el rango y en el tipo de dato

Para la ventana de la Figura XII-3 se codificó:

```
<h:body>
      <h:form id="form">
        <center>
            <span style="margin-right: 15px;"> Numero 1 (máximo
                                               100):</span>
            <h:inputText id="num1"  value="#{inicio.num1}">
               <f:validateDoubleRange maximum="100"/>
            </h:inputText><br/><br/>

            <span style="margin-right: 15px;"> Numero 2(entero
                                       entre 1 y 10):</span>
            <h:inputText id="num2"  value="#{inicio.num2}">
              <f:validateLongRange minimum="1" maximum="10"/>
            </h:inputText><br/><br/>
```

```
            <span style="margin-right: 15px;"> Numero 2(entero
                                              entre 1 y 10):</span>
            <h:inputText id="num3"  value="#{inicio.num2}">
              <f:validateLongRange minimum="1" maximum="10"/>
            </h:inputText><br/><br/>

            <h:commandButton value="enviar"
             action="#{inicio.sumar}"/>

        </center>
      </h:form>

    </h:body>
```

12.2.4 Métodos validadores

A un campo de texto `inputText` se le puede agregar un componente de validación o se puede modificar un método validador para requisitos particulares. Para que se ejecute la validación el campo `required` debe estar en "`true`".

Para invocar al método validador de un campo de texto, es necesario incluir la propiedad *validator*, en esta propiedad se indica el método de un *managed Bean* que se invocará para hacer la validación y el nombre del atributo que se va a validar, por ejemplo:

```
<h:inputText id="emailInput"
       validator="#{registroBean.validarEmail}"
       value="#{registroBean.email}/>
```

método validador que se invoca

atributo a validar

Expresión para validar un e-mail:

\w+([-+.]\w+)

Esta expresión regular indica que una dirección de e-mail es válida si el lado izquierdo de la arroba @ contiene uno o más caracteres de palabra (una letra, un carácter alfanumérico o guión bajo), seguido opcionalmente de un guión, signo + o punto y más caracteres de palabra. Además, el lado derecho de @ debe contener uno o más grupos de caracteres de palabra separados por guión o punto. Por ejemplo, las siguientes direcciones de mail son válidas.

pedro-paramo@mi-email
pedro-direccion.personal@email.com
pedro.paramo@mi-email.com

```
<h:inputText value="e-mail" required="true"
             value="#{Registro.mail}"
            validator="Registro.validarEmail"/><br/><br/><br/>
```

12.3 Librerías adicionales a JSF para mejorar la presentación de las páginas Web

Existen librerías que mejoran la presentación del *front-end*. *PrimeFaces* y *RichFaces* son dos de las más utilizadas. Con estas librerías, los elementos de la Interfaz de Usuario (botones, menus, tablas, etc.) son más llamativos que los de JSF. Hay tutoriales interactivos en internet que enseñan a utilizar cada uno de estos elementos de la interfaz de usuario. Para PrimeFaces recomendamos:

http://www.primefaces.org/showcase/

Y para RichFaces recomendamos:

http://showcase.richfaces.org/

http://livedemo.exadel.com/richfaces-demo/

Cabe señalar que *RichFaces* está descontinuado, es decir, ya no habrá nuevas versiones, por lo que se sugiere *PrimeFaces* como primera opción.

Bibliografía

Basham B., Sierra K., Bates B. , *Head First Servlets and JSP*, O Reilly & Associates, 2nd Ed, USA, 2008.

Deitel P., Deitel H., *Internet & World Wide Web*, How to program. Pearson-Prentice-Hall, USA, 2008.

Deitel P., Deitel H., *Cómo programar en Java*, Pearson Educación, 7a ed., México, 2008.

Falkner J., Jones K., *Servlets and JavaServer Pages*, Addison-Wesley, USA, 2004.García-Molina H., Ullman J., Widom J., *Database Systems. The Complete Book*, Pearson-Prentice Hall, USA, 2009.

Leonard A., *Mastering JavaServer Faces 2.2*, Master the art of implementing user interfaces with JSF 2.2, Packt Publishing, U.K., 2014.

Murach J, Steelman A. *Murach´s JavaServlets and JSP*, Ed. Murach's, 2nd edition, USA, 2008.

Glosario de términos

Aplicación Web.- Es un conjunto de páginas que funcionan en internet; éstas páginas son las que el usuario ve a través de un navegador de internet (Internet Explorer de Microsoft, Chrome, Mozilla Firefox, etc.).

Computadora Cliente.- Computadora que se conecta al servidor vía internet.
Computadora Servidor.- Es es una computadora que se encarga de que las aplicaciones Web sean accesibles a través de internet.

Controlador.- Es el software que procesa las peticiones del usuario. Decide qué modulo tendrá el control para que ejecute la siguiente tarea

Framework.- Se traduce al español como *marco de trabajo* y es un esqueleto para el desarrollo de una aplicación. Los *frameworks* definen la estructura de la aplicación, es decir, la manera en la que se organizan los archivos e inclusive, los nombres de algunos de los archivos y las convenciones de programación.

HTTP.- *HiperText Transfer Protocol*, Protocolo de Transferencia de HiperTexto. Es un protocolo mediante el cual el software servidor se comunica con el navegador instalado en la *computadora cliente*.

JSP.- *JavaServer Page* es una página HTML a la que se le incrusta código Java.

Modelo.- Contiene el núcleo de la funcionalidad, es decir, ejecuta la "lógica del negocio". Se le llama *lógica del negocio* a la forma en la que se procesa la información para generar los resultados esperados. El *modelo* se conecta a la base de datos para guardar y recuperar información

Página Web dinámica.- Es una página Web en la cual el servidor procesa la información proporcionada por el usuario y muestra los resultados de este procesamiento.

Página Web estática.- Es una página Web informativa, en la que el usuario no puede hacer modificaciónes.

Servlet.- Es una clase Java (hija de la clase HttpServlet) que corre en el servidor.

Software Servidor.- Es el software que controla la ejecución de aplicaciones Web, y corre en la *Computadora Servidor.*

Vista.- Son los módulos SW involucrados en la interfaz con el usuario.

Introducción a la Programación Web con Java: JSP y Servlets, JavaServer Faces se terminó de imprimir en la Ciudad de México en septiembre del 2017. La producción editorial e impresión estuvo a cargo de Literatura y Alternativas en Servicios Editoriales S.C. Avenida Universidad 1815-c, Depto. 205, Colonia Oxtopulco, C. P. 04318, Delegación Coyoacán, Ciudad de México. RFC: LAS1008162Z1. En su composición se usaron tipos Minion Pro y Avenir. Se tiraron 100 ejemplares sobre papel.

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA
Unidad Cuajimalpa