

APUNTES MASTER JAVA



Indice de Contenidos

I. Prologo	Pag.- 5
II. Instalar MyEclipse como plugin	Pag.- 6
III. Introducción	Pag.- 10
a. JavaBean	Pag.- 10
b. MVC – Modelo Vista Controlador	Pag.- 11
c. Debug	Pag.- 12
IV. Entorno Grafico en Java	Pag.- 13
a. Contenedores	Pag.- 13
i. Paneles con Barras de Desplazamiento	Pag.- 14
b. Componentes	Pag.- 15
i. Barra de Menus	Pag.- 15
ii. Menus Contextuales (Botón Derecho)	Pag.- 17
iii. Barras de Herramientas (Botones)	Pag.- 18
iv. Radio Button / CheckBox	Pag.- 19
v. Listas Desplegables	Pag.- 20
vi. JTable	
vii. Elementos Comunes	
	Pag.- 21
V. Eventos	Pag.- 23
VI. Excepciones	Pag.- 25
VII. Colecciones	Pag.- 26
a. Seguridad de Tipos	Pag.- 28
VIII. Formatos	Pag.- 29
a. Formato Numérico	Pag.- 29
b. Formato Fecha	Pag.- 29
c. Formato Personalizado	
d. Validaciones	Pag.- 30
IX. Persistencia	Pag.- 33
a. Ficheros	Pag.- 34
i. Texto Plano	Pag.- 34
1. Lectura y Escritura de Archivos de Texto	Pag.- 35
ii. Properties	Pag.- 36
b. Log4J	Pag.- 37
c. Como añadir referencias externas al proyecto con Eclipse	Pag.- 38
d. Ficheros XML	Pag.- 41
i. XML y Java	Pag.- 43
X. Calidad	Pag.- 45
a. Junit	Pag.- 46
b. JavaDoc	Pag.- 53
c. Codepro	Pag.- 55
XI. Ayuda	Pag.- 56
XII. Informes (BIRT)	Pag.- 61
a. Diseño del Informe	Pag.- 62

b. Integración	Pag.- 70
XIII. Reflexion	Pag.- 72
XIV. Bases de Datos	Pag.- 75
a. JDBC	Pag.- 75
b. Transacciones	Pag.- 79
XV. SQL	Pag.- 83
a. Instalación de Oracle XE	Pag.- 87
b. Comandos DML	Pag.- 88
c. Análisis de Bases de Datos	Pag.- 92
XVI. Hibernate	Pag.- 94
a. Instalación de Hibernate con MyEclipse	Pag.- 100
XVII. J2EE	
a. Conceptos de Comunicación Web	Pag.- 106
i. Protocolo HTTP	
ii. Servidor Web	
iii. Contexto	
iv. Herramientas	
v. Despliegue de una Aplicación Web	
vi. Introducción al Servlet	
b. Instalación de Tomcat en MyEclipse	
XVIII. Servlets	Pag.- 107
a. Estructura básica de un Servlet	Pag.- 109
b. HTTPServlet	Pag.- 111
c. GenericServlet	Pag.- 112
d. Ciclo de Vida de un Servlet	Pag.- 114
e. El Interfaz HttpServletRequest	Pag.- 117
f. Cabeceras de petición del protocolo http	Pag.- 119
g. El Interfaz ServletRequest	Pag.- 121
h. El interfaz HttpServletResponse	Pag.- 123
i. Cabeceras de Respuesta http	Pag.- 125
j. Enviando información al cliente	Pag.- 126
k. El Interfaz RequestDispatcher	Pag.- 127
l. El Interfaz HttpSession	Pag.- 128
m. ServletContext	Pag.- 130
n. Filtros	Pag.- 134
o. Escuchadores	Pag.- 137
p. Instalación de Tomcat en MyEclipse	Pag.- 140
XIX. JSP	Pag.- 142
XX. JSTL	Pag.- 143
a. Etiquetas Personalizadas	Pag.- 143
b. EL (Expresion Language)	Pag.- 146
XXI. Internacionalizacion con J2EE	Pag.- 147
XXII. Diseño de la Web	Pag.- 148
a. Selectores CSS	Pag.- 148
b. JavaScript	Pag.- 150
XXIII. AJAX	Pag.- 154
a. JSON (JavaScript Object Notation)	Pag.- 156
XXIV. Struts	Pag.- 159
a. Internacionalización con Struts	Pag.- 165

b.	Etiquetas Personalizadas de Struts	Pag.- 166
c.	Instalación de Struts con MyEclipse	Pag.- 168
XXV.	Tiles	Pag.- 171
XXVI.	JSF	Pag.- 173
a.	Ciclo de Vida de JSF	Pag.- 175
b.	faces-config.xml	Pag.- 177
c.	ManagedBeans	Pag.- 179
d.	Navegación	Pag.- 182
e.	Eventos	Pag.- 184
f.	Páginas Web JSF	Pag.- 187
g.	Etiquetas estándar de JSF	Pag.- 189
h.	Conversiones y Validaciones de JSF	Pag.- 193
	i. Conversiones	Pag.- 194
	ii. Validaciones	Pag.- 197
i.	Mensajes de Error	Pag.- 199
j.	Internacionalización con JSF	Pag.- 200
k.	Implementación de proyecto JSF en MyEclipse	Pag.- 202
XXVII.	ICEfaces	Pag.- 204
a.	Creacion proyecto ICEfaces en MyEclipse	Pag.- 206
b.	DataTable en ICEfaces	Pag.- 208
c.	Contenedores de ICEfaces	Pag.- 211
	i. Contenedor panelPopup	Pag.- 212
XXVIII.	Facelets	Pag.- 213
a.	Creacion de un proyecto facelets en MyEclipse	Pag.- 215
XXIX.	Servicios	Pag.- 217
XXX.	Servicios web	Pag.- 218
a.	Creación de un Servicio Web en MyEclipse	Pag.- 222
	i. Creacion del Cliente del WS	Pag.- 228
b.	Servicios Web Complejos	Pag.- 233
c.	Servicios Web tipo REST	Pag.- 234
d.	Creación de un Servicio Web tipo REST en MyEclipse	Pag.- 235
XXXI.	Spring	Pag.- 239
a.	Ciclo de Vida	Pag.- 242
b.	Propiedades Indexadas	Pag.- 244
c.	Técnicas avanzadas de Spring	Pag.- 246
d.	Módulo de Base de Datos	Pag.- 248
e.	Hibernate con Spring	Pag.- 250
f.	Implementación de un proyecto Spring (solo) en MyEclipse	Pag.- 251
g.	Creacion de Proyecto Hibernate con Spring en MyEclipse	Pag.- 253
h.	Transacciones Declarativa	
XXXII.	Spring-MVC	Pag.- 320
XXXIII.	EJB	Pag.- 262
a.	Creación de un SessionBean de EJB 2.2	Pag.- 264
b.	Preparación Cliente EJB	Pag.- 267
c.	Instalación de JBoss sobre Windows XP	Pag.- 269
d.	Creacion de un proyecto EJB con MyEclipse	Pag.- 279

PRÓLOGO

En el máster de Java vamos a utilizar Win XP con el JDK 6.20, así como Eclipse versión Helios (3.6.1) para la primera parte de J2SE, además usaremos **MyEclipse** 8.6.0a y más tarde 8.6.1 (update file) como plug-in de Eclipse (Galileo 3.5.2) que lo usaremos sobretodo en la parte de J2EE, ya que a fecha de hoy (Febrero 2011) no existe MyEclipse para la versión Helios de Eclipse. Además usaremos los plugins **WindowBuilder** para el entorno gráfico, un repositorio **SVN** para obtener los proyectos que usamos en el curso como ejemplos y **CodePro** Tester, para las pruebas de código y otros análisis.

Al usar MyEclipse tal vez necesitaremos mejorar el rendimiento de Eclipse modificando el archivo `eclipse.ini`, que está en la carpeta `eclipse`. Ese archivo por defecto es así:

```
-startup  
plugins/org.eclipse.equinox.launcher_1.1.0.v20100507.jar  
--launcher.library  
plugins/org.eclipse.equinox.launcher.win32.win32.x86_1.1.0.v20100503  
-showsplash  
org.eclipse.platform  
--launcher.XXMaxPermSize  
256m  
--launcher.defaultAction  
openFile  
-vmargs  
-Xms40m  
-Xmx384m
```

Hay que cambiarlo para que sea así:

```
-startup  
plugins/org.eclipse.equinox.launcher_1.0.201.R35x_v20090715.jar  
--launcher.library  
plugins/org.eclipse.equinox.launcher.win32.win32.x86_1.0.200.v20090519  
-showsplash  
org.eclipse.platform  
--launcher.XXMaxPermSize  
openFile  
-Xverify:none  
-vmargs  
-Xms512m  
-Xmx512m  
-XX:PermSize=256m  
-XX:MaxPermSize=256m  
-XX:ReservedCodeCacheSize=64m  
-XX: UseConcMarkSweepGC  
-XX: CMSClassUnloadingEnabled  
-XX: CMSPermGenSweepingEnabled
```

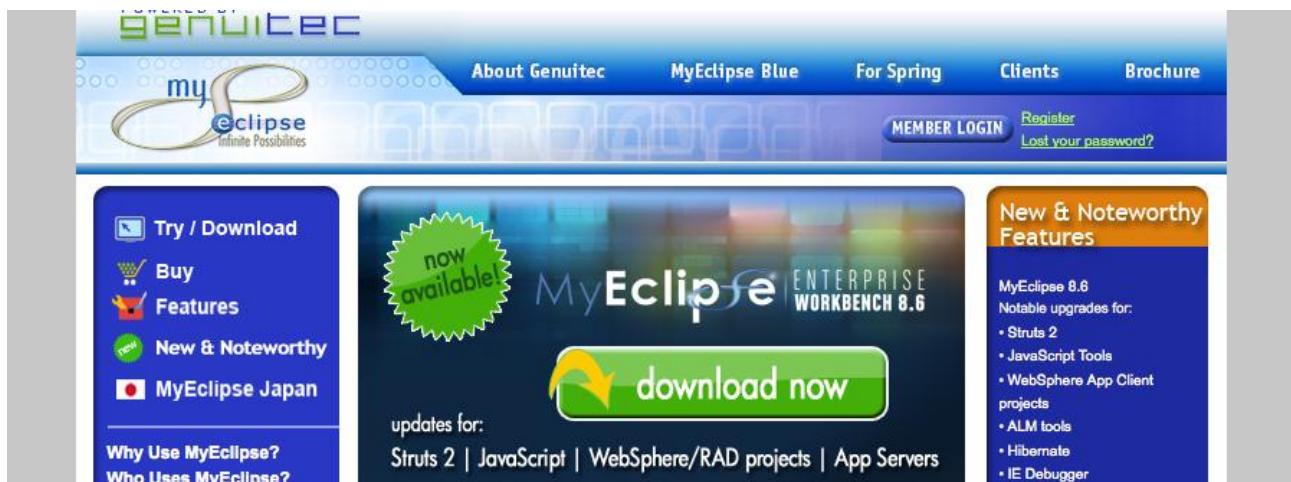
La dirección del repositorio **SVN local** en el aula es: <http://a3-1m/gestmaster>. El repositorio es un plugin aparte que se ha de instalar desde help – install new software... y es un archivo que tenemos

llamado **site-1.6.13.zip**

El entorno grafico lo hemos descargado de <http://dl.google.com/eclipse/inst/d2wbpro/latest/3.6> para Helios y de <http://dl.google.com/eclipse/inst/d2wbpro/latest/3.5> para Galileo con MyEclipse

Instalar MyEclipse como Plugin

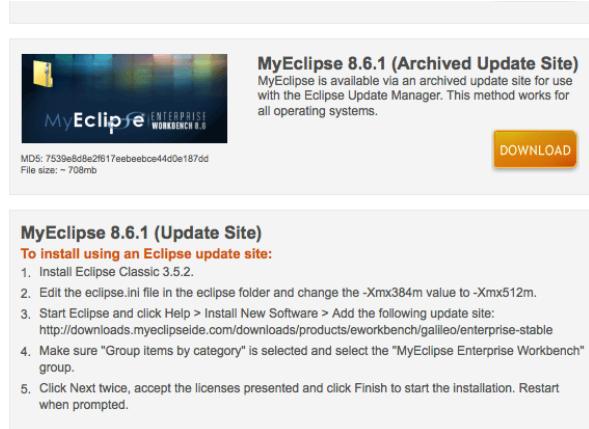
Para instalar MyEclipse como un plugin del propio Eclipse (Galileo 3.5.2) podemos hacerlo de dos maneras una de ellas es bajarnos MyEclipse desde su web. La otra opción es usar la url de descarga desde el sistema de instalacion de plugins de Eclipse. La web de MyEclipse es <http://www.myeclipseide.com/>



Hay que pulsar el botón de “download now”, de color verde y vamos a otra página que nos da una serie de opciones de descarga:



En esa página hay que marcar la casilla de aceptación del acuerdo de licencia y pulsamos en el botón :: DOWNLOAD que se corresponde con la version 8.6.1 para Eclipse 3.5.2. De ahí vamos a una nueva página que tiene varias opciones de descarga nuevamente, tendremos que usar el scroll para ir a la ultima:



La quinta opción es MyEclipse 8.6.1 (Update Site), es la adecuada para instalar MyEclipse como un plugin de Eclipse Galileo (3.5.2 32 bits) siguiendo los pasos que ellos mismos nos indican justo debajo de esa opción de descarga como muestra la imagen, es un archivo de casi 700 Mb, así que hay que tener paciencia. Como podemos ver en la imagen de arriba en la web de MyEclipse nos sugieren dos procesos de instalación, uno de ellos consiste en bajarse el archivo a nuestro disco duro y desde ahí instalarlo como actualización o bien instalar el plugin descargándolo directamente a través de Eclipse. **Las instrucciones para instalarlo desde un sitio de actualización de Eclipse son las siguientes:**

1. Instalar Eclipse Classic 3.5.2. (32 bits)

2. Editar el eclipse.ini en la carpeta de eclipse y cambiar el valor Xmx384m a Xmx512m.

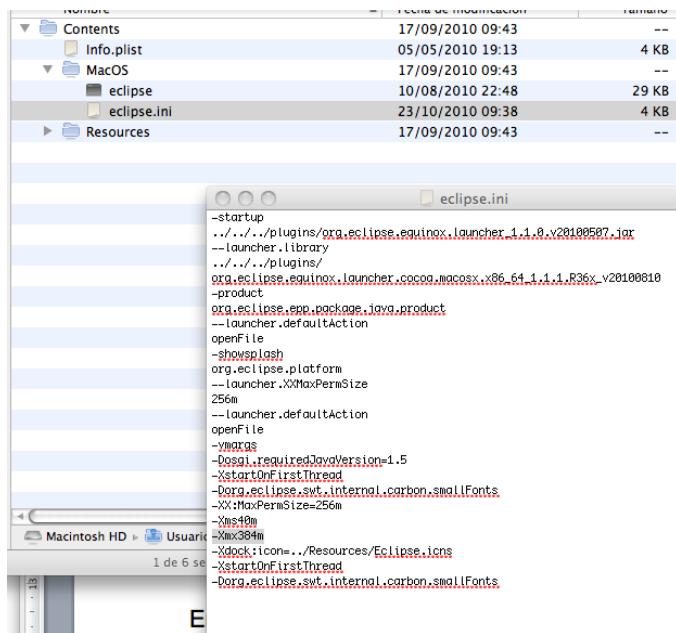
3. Inicio Eclipse y haga clic en Help > Install New Software... > Agregue el sitio de actualización siguiente:

http://downloads.myeclipseide.com/downloads/products/eworkbench/galileo/enterprise-stable

4. Asegúrese de que "Group items by Category" se ha seleccionado y seleccione la opción "MyEclipse Enterprise Workbench" grupo.

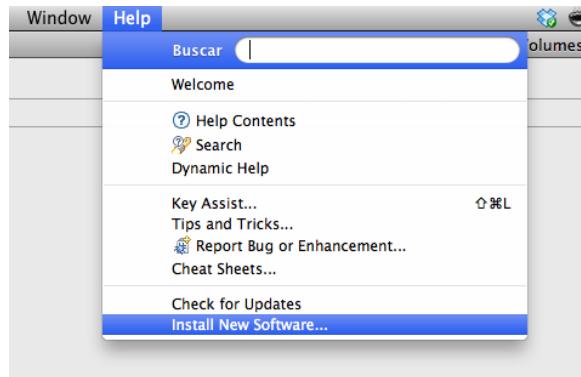
5. Haga clic en Siguiente dos veces, acepte los certificados presentados y haga clic en Finalizar para iniciar la instalación. Reiniciar cuando se le solicite.

Después de tener instalado MyEclipse, cambiamos en Eclipse el archivo eclipse.ini tal como recomiendan en la web de MyEclipse:

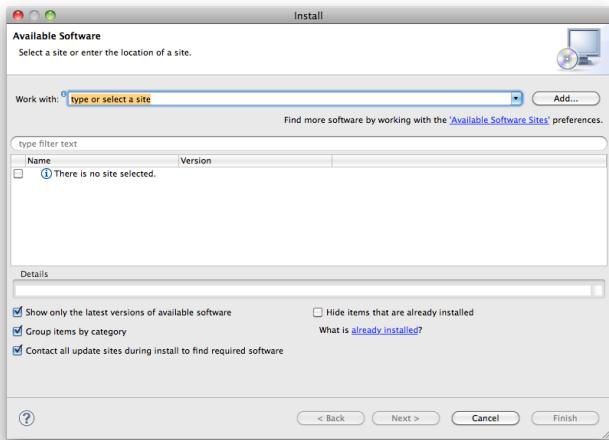


Más detalladamente, el proceso de instalación de MyEclipse es este.

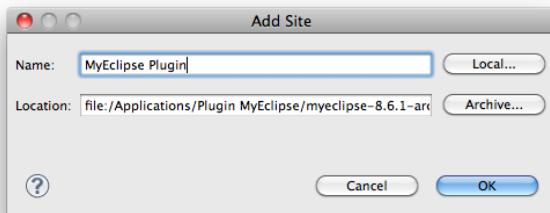
La línea que aparece resaltada en gris es la hay que cambiar, pasando de ser `-Xmx384m` a ser `-Xmx512m`. A continuación en el primer paso vamos a Help – Install New Software..., aparece la siguiente ventana



Pulsamos en `Install New Software...` y vamos a la siguiente pantalla.



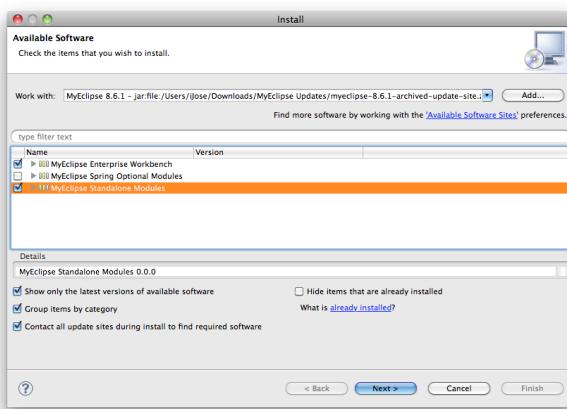
En esa ventana pulsamos el botón Add... y se abre una nueva ventana:



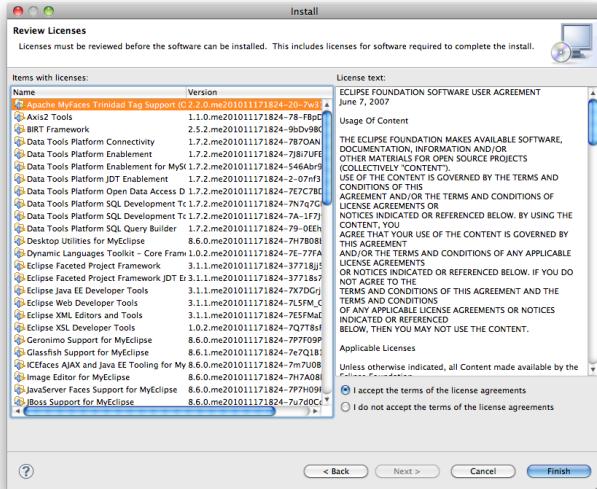
Desde esta ventana podemos instalar el plugin de MyEclipse, para ello tenemos dos opciones una de ellas instalarlo localmente, es decir, previamente hemos descargado el archivo zip, lo hemos puesto en algún sitio de nuestro disco duro y lo localizamos pulsando botón Archive..., además le damos un nombre. O bien en esa ventana escribimos un nombre para el archivo y en la caja Location escribimos la url de descarga que es:

<http://downloads.myeclipseide.com/downloads/products/eworkbench/galileo/enterprise-stable>

En cualquier caso después de poner la ruta y el nombre pulsamos en abrir, y luego en OK y obtenemos la siguiente ventana:



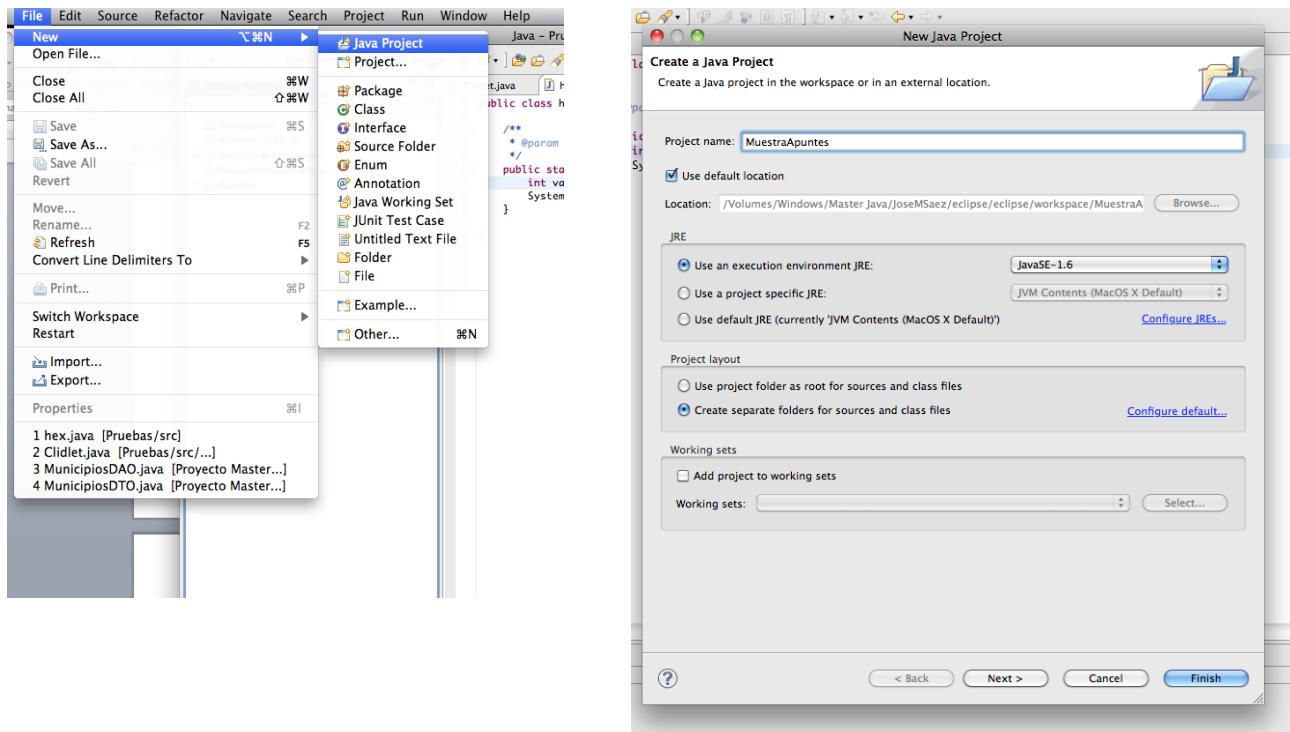
Marcamos las casillas como se ve en la imagen y le damos a next, entonces comienza el proceso de comprobaciones y aparece la siguiente ventana:



Nos pide que aceptemos la licencia con lo cual marcamos la casilla correspondiente y pulsamos finish, entonces comenzara el proceso de instalación y cuando haya terminado nos pedirá permiso para reiniciar, se lo damos y ya está.

INTRODUCCIÓN

Eclipse se divide en Perspectivas y vistas. Para empezar a trabajar con Eclipse en Java lo primero es crear un proyecto Java en Eclipse, para eso se abre File – New – Java Project. Se le da nombre y se elige el entorno de JDK, a continuación se pulsa en Next y luego Finish.



JavaBean

Es una clase con unas características determinadas. Sus propiedades son privadas, el constructor es público sin argumentos y lleva métodos accesores (setters y getters), como mínimo. También se llaman POJO (Plain Old Java Object). Son clases de transporte de datos (DTO) que se usan en la clase de modelo para contener y transportar los datos con los que trabajamos.

A lo largo del master iremos viendo su amplia e importantísima aplicación y otras variedades de javabeans.

MVC – Modelo Vista Controlador

En todos los ejemplos y en el proyecto que vamos a ir creando durante el máster vamos a usar el patrón MVC.

Se trata de una forma de construir las aplicaciones en tres capas, como su nombre indica.

Modelo: Es la parte de la aplicación que se encarga del transporte y la obtención de datos.

Vista: Es la interfaz con el usuario

Controlador: Es la parte del programa donde se define el flujo de la aplicación.

Las tres capas a su vez se dividen en paquetes, para la nomenclatura de los paquetes se usara el formato de dominio inverso según el nombre del cliente, por ejemplo com.atrium.vista.

La capa de Modelo, como hemos dicho, incluye la conexión a bases de datos. Para ello usa dos partes diferenciadas:

DTO: Data Transport Object. Es un JavaBean sin métodos de acción.

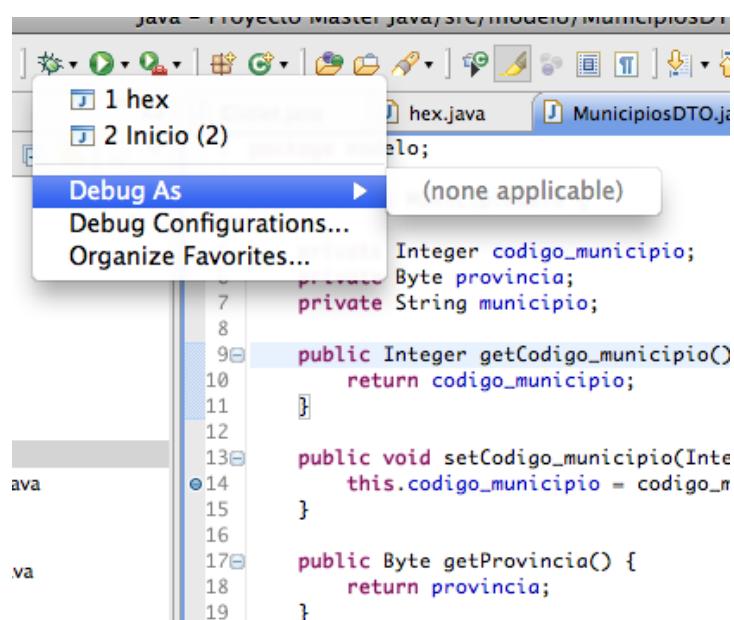
DAO: Data Access Object. Conexión a la base de datos.

En la capa Modelo se incluye también una capa llamada **FACADE** (fachada), es una capa intermedia entre el DAO y el DTO donde se da forma a los datos, por ejemplo el cifrado/descifrado se haría aquí, etc.

Por cada tabla de datos de nuestro interés en la base de datos debe haber un DAO, un FACADE y un DTO.

Debug

Eclipse lleva integrada la posibilidad de depurar el código. Pueden añadirse puntos de interrupción haciendo doble click en el margen izquierdo en el punto donde queramos que se detenga la ejecución de código para ir viendo cómo se va ejecutando. Aparecerá un pequeño círculo azul donde hayamos puesto el punto de interrupción.



ENTORNO GRÁFICO EN JAVA

Hay dos entornos gráficos en Java; **AWT** y **Swing (JFC)**. Principalmente usaremos Swing, ya que además de ser clases extendidas de AWT, tiene más componentes y mejor desempeño.

En Swing encontraremos Contenedores, Componentes y Eventos.

Contenedores

Hay dos tipos de contenedores, los de alto nivel y los de bajo nivel.

Los contenedores de alto nivel son los que son independientes, no pueden ir dentro de otros contenedores pero sí admiten otros contenedores de bajo nivel. Son por ejemplo el JFrame, JWindow.

Los contenedores de bajo nivel pueden ir dentro de otros contenedores y dentro se ponen otros componentes. Un ejemplo de panel de bajo nivel es JPanel, JToolBar, etc.

JFrame es el contenedor principal de la aplicación, se crean como cualquier otra clase:

```
JFrame ventana = new JFrame();
```

Se puede elegir entre varios aspectos para los programas, por defecto el JDK 1.6 usa Ocean, pero se puede cambiar a otros, recientemente se puede elegir también Nimbus. La apariencia de las ventanas se llama Look&Feel.

Los JFrame no especifican el tamaño y la posición.

El método `setBounds(int x, int y, int ancho, int alto);` da la posición (x é y) y el tamaño en pixeles. La posición es la de la esquina superior izquierda de la ventana con respecto a la esquina superior izquierda de la pantalla.

Por defecto, JFrame no es visible, para ello se usa `setVisible(boolean);` Este **debe ser el último de los comandos** que se escriba en el constructor o donde creamos el JFrame.

La propiedad `setResizable(boolean)`, es la que permite o no, que se pueda cambiar el tamaño de la ventana en tiempo de ejecución.

Hay que darle un comportamiento también al botón de cierre de la ventana . Por defecto no hace nada. Los JDK antes del 6 obligan a implementar el comportamiento de este botón usando eventos. El JDK 6 permite cambiar propiedades con el método `setDefaultCloseOperation(constante int);`

El entorno Java permite distribuir el espacio de un contendor de forma automática o manual.

Java utiliza **gestores de posicionamiento** que pueden colocar de alguna forma determinada los componentes que se añaden a un contenedor. Hay 9 gestores de posicionamiento diferentes. Por ejemplo:

BorderLayout – Para ser usado con contenedores.

Los gestores de posicionamiento se pueden eliminar con `setLayout(null);` Cuando eliminamos el layout de una ventana tendremos que definir la posición de los componentes con `setBounds(x,y,ancho,alto);`

`setTitle()` pone título a la ventana en la barra de título. El título también se puede pasar como parámetro en el constructor de la clase JFrame.

JPanel es un contenedor de bajo nivel, esta vacío y es invisible por defecto. Su gestor de posicionamiento por defecto es FlowLayout y coloca en línea los elementos que se le añaden. Lo normal es quitar el posicionamiento en este contenedor.

Para añadir los componentes tendremos que hacer `ventana.Add(etiqueta);`

PANELES CON BARRAS DE DESPLAZAMIENTO

JScrollPane es un contenedor de bajo nivel que sirve para mostrar scrolls en el elemento contenido, por ejemplo un `textArea`.

El contenido pueden ser muchas cosas, por ejemplo `JList`, una imagen, `JTable`, etc. Su sintaxis seria:

```
JScrollPane xxx = new JScrollPane();
xxx.setBounds(...);
xxx.setViewPortView(componente);
```

```
// Para ajustar las barras de desplazamiento en horizontal, vertical o ambas se usa:
xxx.setHorizontalScrollBarPolicy(INT_STATIC);
xxx.setVerticalScrollBarPolicy(INT_STATIC);
```

Donde `INT_STATIC` sería una de estas opciones:

```
HORIZONTALSCROLLBAR. // o bien
VERTICALSCROLLBAR.AS_NEEDED
VERTICALSCROLLBAR.NEVER
VERTICALSCROLLBAR.ALWAYS
```

Las barras de scroll admiten muchas modificaciones sobre todo respecto a su estética, todas las zonas de scroll admiten cambios.

Para añadir movimiento con la rueda del ratón se usa `setWheelScrollingEnabled(boolean);`

Componentes

Los componentes se añaden a los contenedores de bajo nivel. Los componentes más clásicos son las etiquetas de texto JLabel, Los campos de texto JTextField, y los botones JButton.

Un código de Ejemplo que crearía una ventana con un contenedor y algunos componentes seria este:

```
public class JFrame Ventana = new JFrame("Titulo del Programa") {  
  
    public Ventana() {  
        Ventana.setBounds(x,y,ancho,alto);  
        Ventana.setResizable(false);  
        Ventana.setDefaultCloseOperation(this.EXIT_ON_CLOSE);  
        Ventana.setLayout(null);  
        Ventana.setVisible(true);  
  
        JPanel contenedor = new JPanel();  
        JLabel etiqueta = new JLabel();  
        JTextField texto = new JTextField();  
        JButton boton = new JButton();  
  
        contenedor.setLayout(null);  
        etiqueta.setBounds(,,,);  
        texto.setBounds(,,,);  
        boton.setBounds(,,,);  
  
        add(contenedor);  
        contenedor.add(etiqueta);  
        contenedor.add(texto);  
        contenedor.add(boton);  
    }  
}
```

BARRA DE MENÚS

No tiene métodos de posición ni de tamaño ya que siempre va arriba y ocupando todo el ancho de la ventana de la aplicación.

Se coloca dentro del contenedor de la ventana, su nombre es **JMenuBar**. Se instancia como una clase normal `JMenuBar miMenu = new JMenuBar();` es como un contenedor que admite componentes de tipo JMenu y JMenuItem.

JMenu van dentro del contenedor de menu (JMenuBar) y es el menú desplegable. Se declara así:
`JMenu menu1 = new JMenu("Texto del nombre del menu");`

Se generan tantos como cabeceras de menú queramos en la barra de menús de nuestra aplicación. No arrojan eventos. Admiten Texto y/o iconos.

JMenuItem es el contenido del JMenu. Sí lanza eventos de tipo ActionEvent. Se instancia así `JMenuItem nombreItem = new JMenuItem("Texto a mostrar");` Se crean tantos como elementos vaya a tener ese menú.

```
JFrame setJMenuBar(miMenu); // El JMenuBar se añade a la ventana de la aplicación  
miMenu.add(menu1); // así se añaden los JMenus a la barra de menús.  
menu1.add(nombreItem); // así se añaden los elementos al menú.
```

A los JMenuItem hay que añadirles el evento que van a lanzar así: `JMenuItem.addActionListener(this);` El *this* puede ser otra cosa.

Los elementos del menú pueden llevar a su vez otros submenús, para ello lo que hacemos es añadir un JMenu a el JMenu que lo contiene, y a continuación a ese submenú le añadimos sus elementos como JMenuItem.

Se pueden añadir **líneas de separación** en los menús con `JMenu.addSeparator()`

Se pueden incluir **atajos de teclado** o aceleradores tanto en JMenus como en JMenuItems. Para incluir estos atajos de teclado hay dos maneras.

Primera forma:

```
Obj.setMnemonic('char' ó KeyEvent.CONST) El KeyEvent.CONST es algo como  
vk.LETRA.
```

Este es el modo en el que una letra del menuItem queda subrayada y es accesible con el alt + tecla. Pero no es buena práctica si el menu es extenso porque nos quedaremos sin caracteres.

Segunda forma:

Desde JDK 1.4

```
obj.setAccelerator(keyStroke  
.getKeyStroke(KeyEvent.CONST, InputEvent.CONST);  
  
obj.setAccelerator(keyStroke  
.getKeyStroke(KeyEvent.VK_A, InputEvent.ALT_MASK);
```

Donde el KeyEvent.CONST es la tecla pulsada e InputEvent.CONST es la tecla que se combina con la letra, como Alt, Ctrl o Shift, etc.

Después en el ActionPerformed para identificar quien ha sido pulsado:

```
Public void actionPerformed(ActionEvent evento){  
    String opcion = ((JMenuItem)evento.getSource()).getName();  
  
    If(opcion.equals("----")) {  
  
    }  
}
```

Después, como hacíamos con los botones necesitaremos tantos ifs como opciones tengamos para

tratarlo y operar en consecuencia.

Cargar el look&feel

Ira situado en el Inicio para que haga la comprobación de tema antes de crearla.

```
for (UIManager.LookAndFeelInfo laf: UIManager.getInstalledLookAndFeels()) {
    if ("Nimbus".equals(laf.getName())) {
        try {
            UIManager.setLookAndFeel(laf.getName());
        } catch (Exception e) {
        }
    }
}
```

Los iconos deben estar ubicados en algun paquete de la aplicación y con las dimensiones deseadas, porque Java ni crea ni redimensiona, solo muestra. Como ya sabemos tendremos que jugar con la parte fija y dinamica de la ruta de los iconos para que funcione en cualquier equipo.

Para flexibilizar su uso es recomendable crear una Clase BuscarIcono y resolver en ella la ruta necesaria, ya que si en cualquier momento se cambia la ruta por exigencias del cliente, en vez de cambiar cada referencia a cada icono, solo se tendría que cambiar una vez en la Clase creada. Es recomendable que al usarse mucho esta clase, creemos constantes de clase para resolverlo ([Public static final](#)).

MENUS CONTEXTUALES (BOTÓN DERECHO)

Son objetos de la clase **JPopupMenu**.

```
JpopupMenu miMenuContextual = new JpopupMenu();
```

Los elementos que se añadan a este menú ya deberían existir en los menús de la barra de menús, y se añaden así:

```
miMenuContextual.add(obj JmenuElement);
```

Utiliza eventos de ratón para reconocer el botón derecho pulsado.

El listener hay que ponérselo al contenedor en el que queramos que se active.

El evento sería así:

```
Public void MouseClicked (MouseEvent evento)
    If (evento.getClicked == 3)
        xx.show(objContenedor, posicion X, posicion Y); // JPanel,
e.getX, e.getY
```

BARRAS DE HERRAMIENTAS (BOTONES)

Es la hilera de botoncitos que suele ir debajo de la barra de menús, o también pueden ser barras flotantes.

La clase se llama **JToolBar**. Admite iconos, cajas de texto, combos, etc. Es un tipo de contenedor de bajo nivel.

```
JToolBox misHerramientas = new JtoolBox();
```

Hay que darle posición y tamaño con `setBounds(x, y, ancho, alto);`

Para darle o quitarle la propiedad de flotabilidad se usa `obj.setFloatable(boolean);`

`Obj.add(nombre);` añade los componentes que le queramos agregar, en el orden a ser agregados. El layout que usa por defecto suele ser el adecuado para este contenedor.

RADIO BUTTON / CHECK BOX

Solo tienen dos estados posibles, seleccionado o no seleccionado.

Permiten texto y / o icono.

Ambos pueden ser excluyentes (solo uno puede estar seleccionado) o no (múltiples selecciones).

Para hacerlos excluyentes se usa la clase **ButtonGroup**. La sintaxis para hacer que un grupo de estos componentes sean excluyentes esta:

```
JChekBox chkUno = new JCheckBox();
JChekBox chkDos = new JCheckBox();
ButtonGroup grupoComponentes = new ButtonGroup();
grupoComponentes.add(chkUno);
grupoComponentes.add(chkDos);
```

Los botones de tipo **JToggleButton** tambien se comportan del mismo modo.

Se les puede añadir eventos de tipo ActionListener o ChangedListener. Para capturar cuando se ha seleccionado o no se usa ChangedListener, y para capturar cuando ha sido pulsado solamente se usa ActionListener.

LISTAS DESPLEGABLES

JComboBox (lista desplegable de selección única)

JList (listado que permite selección múltiple)

Para que un Jlist admita selección multiple se usa
`JList.setSelectionMode(listSelectionModel.CONSTANT);`

Las constantes para el tipo de selección de JList son: `ListSelectionModel.`

`SINGLE_SELECTION` → selección única

`SINGLE_INTERVAL_SELECTION` → múltiples elementos pero seguidos

`MULTIPLE_INTERVAL_SELECTION` → selección múltiple

Sintaxis para creación de un ComboBox:

```
String[] tablaDatos = new String[int]();  
JComboBox miCombo = new JComboBox(tablaDatos);
```

El método **JComboBox.add()** para añadir elementos al combo **NO FUNCIONA**.

Para saber el elemento que ha sido seleccionado se puede elegir el método

`getSelectedIndex()` que nos dará la posición (int) o -1 en caso de no existir, o el método `getSelectedItem()` que nos dará el contenido de tipo Object, con lo cual habrá que castear.

Lanzan eventos ActionListener.

Para actualizar la lista de elementos del combobox hay que eliminarlo primero y entonces reinstanciarlo (new).

Para indicar como será presentada la información en un JList:

`setLayoutOrientation.JList.CONSTANT()` siendo las constantes:
`JList.VERTICAL`
`JList.HORIZONTAL.WRAP`
`JList.VERTICAL.WRAP`

Cuando se elige la opción de selección múltiple, para obtener los datos seleccionados se elige `getSelectedValues()` o `getSelectedIndexes()` que devuelven un Array de ints o de Objects que habrá que castear a String normalmente.

El método `setVisibleRowCount(int)` indica el numero de líneas de elementos visibles.

JTABLE

Es el componente más complicado. La clase con más métodos de Java

```
JTable tabla = new JTable([int],[int]);
```

Con él se puede hacer cualquier cosa que pueda ser representada en un elemento tabular. Habitualmente se muestra dentro de un **JScrollPane**, para tener barras de desplazamiento.

El JTable tiene como modelo un elemento que hereda de la Clase **AbstractTableModel**. Al ser abstracta no puede ser instanciada, sino heredada, la clase que hereda esa otra clase es **DefaultTableModel**, pero es básica y solo se usa para mostrar información. Esa clase solo se encarga de manejar la información que le llega al JTable, no de dibujar esa información en el JTable. Hay que reescribirse el modelo para que haga lo que queremos.

```
java.lang.Object
└ javax.swing.table.AbstractTableModel
    └ javax.swing.table.DefaultTableModel
```

Los métodos abstractos de AbstractTableModel son:

Object getValueAt(int fila, int Columna) – Devuelve un Object con la información una celda concreta definida por Fila y Columna.

Int getColumnCount() – devuelve las columnas que tiene que pintar

Int getRowCount() – devuelve las filas que tiene que pintar

Otros métodos no abstractos de AbstractTableModel son:

Void setValueAt(Object, int Fila, int columna) – Object es el valor que se manda la celda a, siendo fila y columna la celda en concreto donde va la información de Object.

Boolean isCellEditable(int Fila, int Columna) – Devuelve la condición de editable de una celda definida por Fila y columna

Si quisiéramos poner una **cabecera** al JTable usaremos el objeto **JTableHeader**.

A la vez el JTable tiene filas y **columnas**, cada columna es un objeto **JTableColumn**.

Se pueden establecer formas de **ordenación**, eso corre a cargo de **JTableSorted**.

Se pueden hacer **filtros** sobre la información a través de **JTableFilter**.

Se puede permitir la inserción de información en las celdas para controlar el formato de las celdas se usa **DefaultTableCellRenderer**.

Usando todos estos objetos de JTable podemos hacer cualquier cosa con este elemento.

No se pueden hacer objetos de una clase abstracta, por lo tanto esta creada para que otras hereden de ella es como una interfaz pero con código. Puede tener métodos sin código como abstractos, y

con ello sera obligatorio implementarlo.

Una vez tengamos claro el uso que le vamos a dar a nuestro JTable, tenemos que crear nuestro propio modelo:

```
public class xxx extends AbstractTableModel {  
  
    public Object getValueAt(int Fila, int columna){} //valor celda  
    public int getColumnCount(){} //numero de columnas a colocar  
    public int getRowCount(){} //numero de filas a colocar  
-----o-----  
    Public void setValueAt(object, int column, int row)  
  
}
```

Hay algunos modelos que casi siempre se sobre escriben, es el caso de **setValueAt()** con el que mandamos la información al modelo.

Otro es el método **isCellEditable(int, int)→bool**. Las celdas por defecto estarán deshabilitadas para escribir.

En nuestro proyecto del master tenemos un paquete separado para todo lo referente al JTable, en ese paquete están las clases, donde podemos ver cómo trabajar con este elemento para cambiar cosas como anchos de columna, numero de filas y columnas, propiedad modificable de las celdas, las líneas de separación entre celdas, añadir o quitar filas etc.

Ejemplo:

1. El numero de filas y columnas tiene que ser dinamico. El usuario tiene que poder manipular las columnas y filas. En este caso vamos a dejar las filas dinámicas para que el usuario pueda añadir o borrar las que quiera. Esto nos llevara a necesitar implementar un ArrayList dentro de otro ArrayList.
2. Decidir si las celdas serán editables o no. Esto se decide por columnas
3. Una fila de edición configurable
4. Definir el tamaño de la columna

```
package com.atrium.jtablepersonalizado;  
  
import java.util.ArrayList;  
  
import javax.swing.JFrame;  
import javax.swing.JScrollPane;  
import javax.swing.JTable;  
import javax.swing.UIManager;  
import javax.swing.table.AbstractTableModel;  
  
/**  
 * Modelo personalizado para los Jtable. Reglas de uso:  
 * <ul>  
 * <li>  
 * Se añade de forma automatica una primera columna y una ultima fila.</li>  
 * <li>  
 * Ni el la primera columna ni en la ultima fila se podra escribir.</li>  
 * <li>  
 * De forma dinamica añadiremos filas o las quitaremos.</li>  
 * <li>  
 * Las columnas siempre seran las mismas, una vez creado el modelo.</li>
```

```

* <li>
* La ultima fila servira para, al hacer click en ella, añadir una nueva fila.</li>
* <li>
* Mediante la eleccion en menu de despliege rapido se elegira la opcion de
* guitar o poner filas al modelo</li>
* </ul>
*
* @author Profesor
* @version 1.6 14-07-2011
*
*/
public class Modelo_JTable extends AbstractTableModel {

    // estructura de datos para el modelo
    private ArrayList<ArrayList<String>> datos_modelo = new ArrayList<ArrayList<String>>();

    // variables que nos diran el numero de filas y columnas del modelo
    private int filas;
    private int columnas;

    // propiedad que controlara si se añada una fila mas de edicion al modelo o
    // no valor por defecto no (false).
    private boolean fila_edicion = false;

    // tabla de permisos de escritura por columnas
    private boolean permiso_escrituraceldas[];

    // ----->>>>>> ZONA DE CONSTRUCTORES <<<<<<<<<-----
    /**
     * Construimos el modelo sin indicar tamaño de filas ni columnas. se debera
     * llamar a continuacion al metodo establecer tamaño para poder darle ese
     * tamaño inicial
     */
    public Modelo_JTable() {
    }

    /**
     * Construimos el modelo indicandole el numero inicial de filas y columnas
     * Adoptaria la opcion por defecto de no tener fila de edicion
     *
     * @param filas
     *          Numero de filas del modelo.
     * @param columnas
     *          Numero de columnas del modelo.
     */
    public Modelo_JTable(int filas, int columnas) {
        this.establecer_Tamaño(filas, columnas);
    }

    /**
     * Construimos el modelo indicandole el numero inicial de filas y columnas.
     * En este constructor se indicaria con un true el echo de adoptar una fila
     * de edicion
     *
     * @param filas
     *          Numero de filas del modelo.
     * @param columnas
     *          Numero de columnas del modelo.
     * @param fila_edicion
     *          Booleano que indicara si se usa la fila de edicion o no.
     */
    public Modelo_JTable(int filas, int columnas, boolean fila_edicion) {
        this.fila_edicion = fila_edicion;
        this.establecer_Tamaño(filas, columnas);
    }

    // ----->>>>>> FIN DE ZONA DE CONSTRUCTORES <<<<<<<-----

    // ----->>>>>>> METODOS PROPIOS DEL MODELO <<<<<<<-----

```

```

/**
 * Creara la cantidad de filas y columnas necesarias en cada caso. Ademas
 * comprobara si ya tuviera cualquier contenido anterior borrandolo. Esto
 * posibilitara que el usuario pueda borrar el contenido del modelo de forma
 * dinamica a partir del uso de la interface.
 *
 * @param filas
 *          Numero de filas iniciales.
 * @param columnas
 *          Numero de columnas definitivas.
 */
public void establecer_Tamaño(int filas, int columnas) {
    if (fila_edicion) {
        this.filas = ++filas;
    } else {
        this.filas = filas;
    }
    this.columnas = columnas + 1;
    if (!datos_modelo.isEmpty()) {
        datos_modelo.clear();
    }
    for (int i = 0; i < filas; i++) {
        datos_modelo.add(this.crear_FilaNueva());
    }
    this.primera_Columna();
}

/**
 * Proceso mediante el cual creamos una fila para añadirla al modelo.
 *
 * @return El objeto arraylist que es la nueva fila a añadir al modelo.
 */
private ArrayList<String> crear_FilaNueva() {
    ArrayList<String> fila_nueva = new ArrayList<String>(columnas);
    for (int col = 0; col < this.columnas; col++) {
        fila_nueva.add(" ");
    }
    return fila_nueva;
}

/**
 * Numera las filas del Jtable, excepto la ultima donde pone un asterisco.
 */
private void primera_Columna() {
    for (int i = 0; i < filas; i++) {
        ArrayList<String> datos_filas = datos_modelo.get(i);
        datos_filas.set(0, String.valueOf(i + 1));
        if (fila_edicion && i == filas - 1) {
            datos_filas.set(0, "*");
        }
    }
}

/**
 * Recibe los parametros para poder establecer la logica de decision de que
 * columna es modificable y cual no. La primera columna siempre sera no
 * editable por parte del usuario.
 *
 * @param tabla_permisos
 *          Tabla de booleanos para indicar las columnas editables.
 */
public void setPermisos(boolean tabla_permisos[]) {
    int posiciones = tabla_permisos.length + 1;
    permiso_escrituraceldas = new boolean[posiciones];
    permiso_escrituraceldas[0] = false;
    for (int i = 1; i < posiciones; i++) {
        permiso_escrituraceldas[i] = tabla_permisos[i - 1];
    }
}

```



```

@Override
public void setValueAt(Object valor, int fila, int columna) {
    if (columna > 0) {
        datos_modelo.get(fila).set(columna, (String) valor);
    }
}

// ----->>>>>> FIN ZONA DE SOBRESCRITURA DE METODOS DE CLASE <<<<<<

// ----->>> ZONA DE SOBRESCRITURA DE METODOS ABSTRACTOS DE CLASE <<<<
/** 
 * Devuelve el numero de columnas del modelo
 *
 * @return int Numero entero que son las columnas.
 * @see javax.swing.table.TableModel#getColumnCount()
 */
@Override
public int getColumnCount() {
    return columnas;
}

/** 
 * Devuelve el numero de filas del modelo
 *
 * @return int Numero entero que son las filas.
 * @see javax.swing.table.TableModel#getRowCount()
 */
@Override
public int getRowCount() {
    return filas;
}

/** 
 * Metodo invocado por el objeto Jtable cada vez que necesita saber el dato
 * de una celda en concreto.
 *
 * @param fila
 *          Número de la fila del dato a obtener
 * @param columna
 *          Número de la columna del dato a obtener
 *
 * @return Object Valor que el Jtable representa dentro de la tabla.
 * @see javax.swing.table.TableModel#getValueAt(int, int)
 */
@Override
public Object getValueAt(int fila, int columna) {
    ArrayList<String> fila_datos = datos_modelo.get(fila);
    String dato = fila_datos.get(columna);
    return dato;
}

// --->> FIN ZONA DE SOBRESCRITURA DE METODOS ABSTRACTOS DE CLASE <<---

// ----->>>>>>>><< ZONA DE METODOS ACCESORES <<<<

public boolean isFila_edicion() {
    return fila_edicion;
}

public void setFila_edicion(boolean filaEdicion) {
    fila_edicion = filaEdicion;
}

public boolean[] getPermiso_escrituraceldas() {
    return permiso_escrituraceldas;
}
}

```

Para establecer si son editables, necesitare una propiedad de clase array para guardar los valores de

true/false boolean tabla_permisos. Necesitaremos un metodo setPermisos, al que le pasaremos el array para establecer los permisos. Tabla_permisos[] = {true, false, false, true}.

Nos quedara implementar que se pueda crear una fila o columna. En el metodo quitar_Filas() debemos implementar que se puedan borrar una o varias filas. Cuales son las seleccionadas nos lo dara la vista de una propiedad selected capturando el evento o con un menu contextual. En este metodo tendre que recorrer al reves para que si quito filas sean del final.

```
public void quitar_Fila(int[] fila_a_eliminar) {
    int tamaño = fila_a_eliminar.length;
    for (int i = tamaño; i > 0; i--) {
        datos_modelo.remove(fila_a_eliminar[i - 1]);
    }
    filas = filas - fila_a_eliminar.length;
    this.primera_Columna();
}
```

En el metodo añadir_Filas()

```
public void añadir_Fila(){
    datos_modelo.add(crear_FilaNueva());
    this.filas++;
    this.primera_Columna();
}
```

Vista_JTPersonalizado

Ahora tendremos que crear una clase para implementar la vista del JTable. Para ello la vista tiene que conocer el JTable (prop JTable en Vista_JTPersonalizado con sus accesores).

Hay 3 actores: El JTable (nos lo da el JDK), Modelo_JTable(mio) y Vista_JTPersonalizado(mio). Al JTable le paso el Modelo_JTable y a este Vista_JTPersonalizado.

Para poder meter las cabeceras a las columnas, creamos el metodo crear_Cabeceras, tendremos que coger las columnas. Para ello le pedimos al modelo que nos diga el numero de columnas

Public void establecer_Cabeceras(String cabeceras[])

La filosofia general es crear metodos que reciben arrays para meter parámetros en serie y después instanciar arrays con los valores y pasarlos como argumentos en las llamadas a los metodos.

Para establecer los anchos creare el metodo y le pasare un array de Strings para decirle el ancho en px de las columnas

Public void establecer_Anchos(int [] anchos)

En el establecemos que no se puedan reordenar las columnas de cabecera.

Centramos el texto de las cabeceras

Le colocamos el ancho preferido (que debe coincidir con el ancho minimo por cuestion interna de swing)

Anulamos la posibilidad de cambiar el ancho

Creamos un metodo establecer_Movilidad de Columnas(boolean movilidad) para que podamos decidir si permitirlo o no. Tambien necesitaremos la prop correspondiente mover_columnas.

establecer_Movilidad de Columnas(boolean movilidad)

Creamos otro metodo para permitir que se modifiquen los ancho o no. Creamos la prop bool [] permisos_ModificarAnchos.

permisos_ModificarAnchos(boolean permisos_ModificarAnchos);

Creamos un metodo para crear los permisos por defecto y otro para establecer los permisos que queramos.

Establecer_PermisosAnchos()

Establecer_PermisosPorDefecto()

Creamos un metodo para establecer el rayado de las filas. Para pasarle el tipo de fila se requieren numeros por lo que nos creamos constantes de clase, para no tener que recordarlo.

Establecer_Rayado(int tipo_lineas)

Menu contextual (en Vista_JTPersonalizado)

Creamos un menu para crear filas o quitarlas.

Crear_MenuRapido()

Instanciamos los JMenuItem, los añadimos al menu y por ultimo añadimos los escuchadores al JTable, que es el que tiene que recibir el mensaje. En este caso utilizaremos ActionCommand("xxxxxx"), con el que no necesitamos establecer setName. En el Action performed compruebo que entra con un evento.equals("xxxxxx") y si coincide con poner, añadimos una fila con nuestro metodo `añadir_Fila()` e idem con `quitar_Fila()`, pero tengamos en cuenta que necesitaremos castear a nuestro Modelo_JTable para poder sus metodos porque es el quien los tiene. Despues de hacer esto, tenemos que revalidar y repintar. Tendremos que instanciar en el main el setMenuRapido a trae.

Para ocultar filas añadimos otra opcion al menu rapido, otro escuhador y otro if en el actionPerformed(). La mecanica sera identificar la fila seleccionada y despues hacer su alto = 0. Asi la info se mantiene pero no se ve.

Probablemente nos interesa un modelo que admita cualquier numero de filas y columnas. Además nos puede interesar que la primera columna sea una que vaya numerando las filas. También hemos de idear un sistema que permita decidir que celdas pueden ser modificadas y cuáles no, esto puede hacerse por filas o por columnas. También podemos permitir que el usuario pueda añadir tantas filas

como desee, con lo cual podemos poner una última fila cuya primera fila sea un botón que sirva para añadir filas y abrir una ventana de configuración.

La manipulación de las celdas se hace a partir de una clase que hereda de DefaultTableCellRenderer (que es un JTextField). Habrá una clase por cada tipo (fechas, números enteros, números con decimales). Tendremos que interceptar el setValue para poner el formato y mandarlo a la tabla. Como donde se mostrará la info será en un JTextField, llamaremos a setText para pasárselle el valor.

```
Public class Celda_Fecha extends DefaultTableCellRenderer{
{
    En el constructor pasamos el formato que querremos ("yyyy-mm-dd")
}

Public void setValue(object valor)
{
    If(valor != null){

        Try{
            setText(formato.format(valor));
        }catch{
            setText("");
        }
    }
}
}
```

Nos creamos tantas constantes de clase como formatos necesitemos. Estas servirán para pasárselas al constructor y elegir la fecha.

Una vez que ya tenemos las clases de los formatos, en la vista (Vista_JTPersonalizado) nos creamos un método establecer_Formato(DefaultTableRender formats[])

Editor

Un componente me da la visualización y otro la edición (para introducir información). También crearemos una clase por cada editor. Editor_moneda, etc. Aquí nuestra clase hereda de DefaultCellEditor. Aquí usaremos JFormattedTextField.

getTableCellEditorComponent

Este método nos da el componente al que llama JTable para editar. Nosotros lo interceptamos y lo formateamos.

Cuando el usuario termina de editar y pulsa intro tenemos que pasar esa información al componente que lo muestra. Para eso utilizamos el método getCellEditorValue().

También tendremos que sobrescribir stopCellEditing(). En este método damos por finalizada la edición de la celda. Necesitamos llamar a la implementación de la clase padre.

En la vista tendremos que tener un método establecerEditoresCeldas(DefaultCellEditor editores[]).

ELEMENTOS COMUNES

Muchos componentes tienen la capacidad de incorporar un **icono**. Para ello tienen la propiedad `setIcon()`; Los iconos se almacenarán en un paquete propio de recursos externos. En la ruta a ese paquete habrá dos partes, una dinámica según el equipo en el q se encuentre y otra fija en el paquete q lo hayamos metido.

```
.setIcon(newImageIcon(getClass().getResource("com/atrium/image/xx.jpg")));  
 .setIcon(newImageIcon(getClass().getResourceAsString(idem)));
```

`newImageIcon()`; es una instanciación anónima.

Hay un metodo en la clase Class, que permite averiguar la parte dinámica de la ruta pasandole la parte fija. Class proporciona objetos para obtener información.

El **color** se puede obtener de la clase Color. Se utilizan varios metodos para cambiar el fondo y el color frontal (del texto):

```
setBackground(Color.YELLOW);  
.setForeground();
```

Hay constantes de clase en color con el nombre de una docena de colores basicos. Sin embargo para obtener cualquier otro color más exacto habrá que instanciarse un objeto Color y especificar el RGB

```
Color color = new Color(0-255, x, x);
```

Fuente: se utiliza `setFont()`; para establecer la fuente, que debe estar previamente instalada en el equipo.

```
.setFont(new Font("nom fuente", Font.BOLD+Font.ITALIC,el cuerpo de la  
letra en int));
```

Nombre fuente(String)

Estilo de la fuente(CONSTANTES DE CLASE): negrita, italica.

Cuerpo fuente(int): el tamaño normal es 10-12

Propiedades comunes

Anular un control: `setDisable(true);`

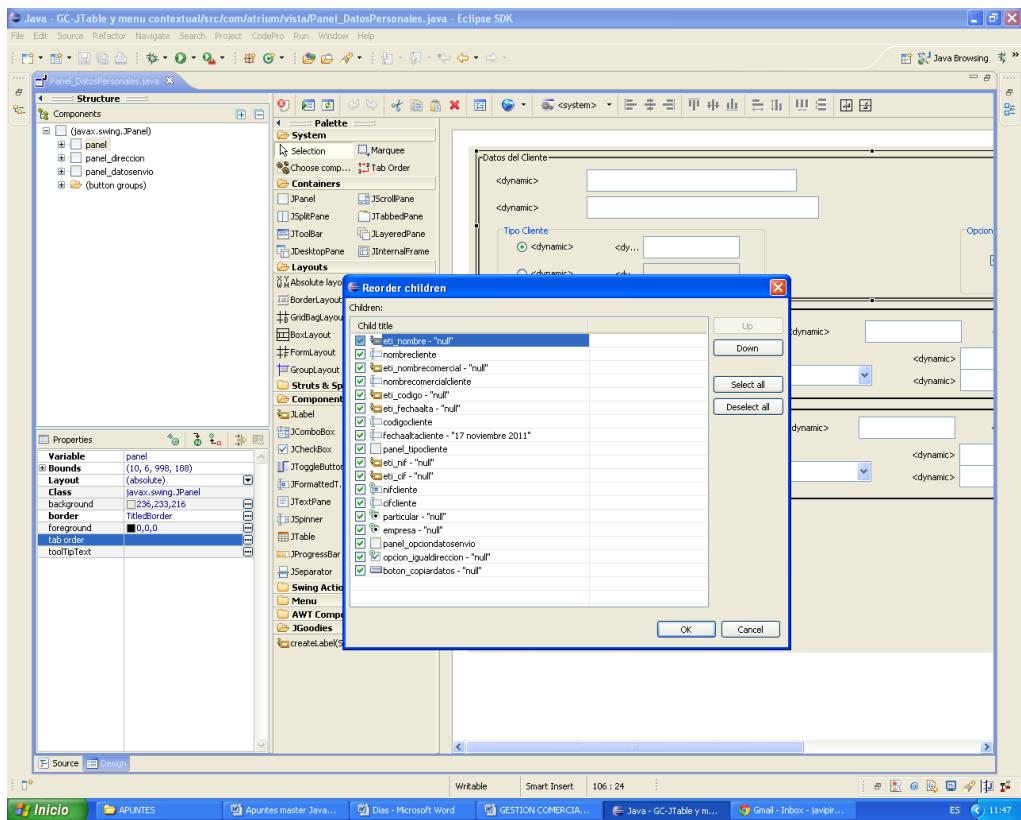
Hacer un elemento opaco: `setOpaque(false);`

Dar nombre: `setName();`

FocusCycle

Puedo establecer el orden de foco. Las etiquetas no suelen llevarlo. Podemos tambien elegir las teclas que hacen que se cambie el foco de un componente a otro. Por ejemplo que al pulsar intro ademas de validar pase el foco al siguiente componente.

Vamos a windowBuilder y en las propiedades de los contenedores, seleccionamos taborder, en el asistente quitamos o ponemos checks y cambiamos el orden en base a nuestras necesidades.



En cuanto al código que se genera

```
panel.setFocusTraversalPolicy(new FocusTraversalOnArray(new Component[]{nombreciente, nombrecomercialcliente, nifcliente }));
```

Lo englobamos en una función y añadimos la gestión de teclas que resuelven el foco

```
Public establecer_GestionFoco () {

//ESTABLECEMOS EL ORDEN DE FOCO
panel.setFocusTraversalPolicy(new FocusTraversalOnArray(new Component[]{nombreciente, nombrecomercialcliente, nifcliente }));

//ESTABLECE TECLAS NUEVAS PARA EL CICLO DE FOCO
Set<AWTKeyStroke> teclas_avance= getFocusTraversalKeys(
KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS);
    Set newForward = new HashSet<AWTKeyStroke>(teclas_avance);
    newForwardKeys.add(KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0));
    setFocusTraversalKeys(KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS,
newForwardKeys);
}
```

EVENTOS

En un entorno gráfico de ventanas, es necesario el uso de eventos para que las acciones del usuario sobre los componentes ejecuten alguna acción.

Los componentes son los emisores, lanzan un evento que es capturado por un receptor también llamado listener o escuchador. A los componentes que queramos que lancen un determinado evento hay que añadirles el evento. Por ejemplo, `boton.addActionListener(Object)`.

La relación entre emisores y escuchadores puede ser de un emisor a muchos escuchadores o también de muchos emisores a un solo escuchador.

Los eventos más habituales son los eventos de acción “ActionEvent”, los crea el usuario al, por ejemplo, elegir una opción de un menú, o al pulsar un botón.

Los listeners o escuchadores **son en realidad interfaces**. Con lo cual sus métodos han de ser implementados por la clase y es en esa implementación de los métodos abstractos de la interface (listeners) donde se escribe el código de la acción que se desarrollará cuando se produce el evento. Cada componente puede lanzar una serie de eventos propia del componente.

Un ejemplo seria este:

```
Public class xxx implements ActionListener {  
  
    JButton B1 = new JButton("Boton Pruebas");  
    B1.addActionListener(this);  
  
    Public void actionPerformed(ActionEvent evento) {  
        JButton boton = (JButton)evento.getSource();  
        If (evento.getText() = "Boton Pruebas") {  
            // codigo de la accion a realizar  
        }  
    }  
}
```

`evento.getSource()`; da un objeto de tipo Object y desde esta clase no tenemos forma de saber que botón ha sido pulsado. Por ello se tiene que hacer una conversión `JButton boton = (JButton)evento.getSource()`;

`GetText()`; nos sirve para saber el texto del botón, pero esto no es totalmente eficiente, ya que si idiomatizas la aplicación el texto cambiara en función de idioma. Para ello podemos usar `getName()`; que nos da un name que le hemos tenido que asignar antes con `setName()`;

```

Public class xxx implements ActionListener {

    JButton B1 = new JButton("Boton Pruebas");
    B1.addActionListener(this);
    B1.setName("Nombrado por mi");

    Public void actionPerformed(ActionEvent evento) {
        JButton boton = (JButton)evento.getSource();
        If (evento.getName() = "Nombrado por mi") {
            // codigo de la accion a realizar
        }
    }
}

```

El operador `instanceof` nos da un bool de si es una instancia de la clase q le digamos
 Hay muchos tipos de evento, por ejemplo:

ActionEvent → ActionListener →actionPerformed()

FocusEvent → FocusListener →focusGained() y focusLost()

MouseWheelEvent → MouseWheelListener → mouseWheelMoved()

MouseMotionEvent → MoseMotionListener →mouseDragged() y mouseMoved()

KeyEvent → KeyListener →keyPressed(), keyReleased() y keyTyped()

MouseEvent → MouseListener →mouseClicked(), mousePressed(), mouseReleased(), mouseEntered(), mouseExited().

ItemStateChangedEvent → ItemListener →IStateChanged()

windowEvent → WindowListener → windowActivated(), windowDeactivated(), windowClosed(), windowClosed(), windowClosing(), windowIconified(), windowDeiconified(), windowOpened

windowFocusEvent → WindowFocusListener → windowFocusGained() y windowFocusLost()

windowStateEvent → WindowStateListener → windowStateChange()

FOCO: En ppio todos los componentes tienen foco, pero se puede modificar el ciclo de foco, por ejemplo para quitar las labels.

MOUSE: MouseEvent tiene métodos de coordenadas y también para saber el botón del ratón que se ha pulsado, el método `MouseEvent.getButton()` devuelve un int, 1, 2 ó 3, para referirse al botón izquierdo (1), central (2) o derecho (3).

`MouseEvent.getX()` y `MouseEvent.getY()` devuelven las coordenadas del botón.

EXCEPCIONES

Las excepciones no siempre son errores. Existen dos tipos de excepciones a nivel de tratamiento de las mismas:

Activas: son de obligada captura

Pasivas: Si no se capturan termina el programa. Están más relacionadas con errores.

Las excepciones las lanzan todos los métodos y se capturan con el procedimientos **Try / Catch**.

Las estructuras **Try / Catch** son anidables. Puede haber más de un catch de diferentes tipos pero deben colocarse por orden de menor (más específico) a mayor (más general). Si tenemos dos metodos en un try/catch que generan el mismo tipo de excepcion podemos tratarlo con un catch, pero no sabremos que metodo lanza la excepcion.

Como resolver una excepcion es algo no reglado, aunque normalmente lo que necesitaremos sera, informar al usuario de lo que ha pasado y registrarla. La mejor opcion para tratar excepciones es centralizar en una clase todos los tratamientos, de esta forma es mucho mas funcional.

La clausula **finally** es opcional, pero si la hay, su código se ejecuta siempre.

Delegación: El tratamiento de una excepcion se puede delegar. Utilizando la la palabra reservada **throws** en la firma seguido del tipo de excepcion, indicamos que ese metodo genera excepciones

Para hacer que un método lance una excepción se usa la palabra reservada **Throws** seguida del nombre de la clase de la excepción. Si se usa **Throws** en la declaración de un método, deben usarse estructuras **Try / Catch** en su llamada.

Para **crear una clase de tipo Exception** se debe extender la clase **Exception** y su nombre debería terminar también con la palabra **Exception**. El método que pueda lanzar (**Throws**) esa excepción debe llevar en algún punto (en un if o similar) la palabra **Throw** (sin s) para indicar que si se produce tal cosa, se lance la excepción. La sintaxis es así:

`Throw new miException();`

Su usamos esa sintaxis en un catch estaríamos intercambiando la excepción que produce el catch por la que le indicamos con el Throw.

Es buena idea crear una clase en nuestras aplicaciones que se encargue del manejo de las excepciones y así no repetir código en el programa.

COLECCIONES

Existen más de 30 clases que son colecciones. Se llama **Collection Framework**.

Su ventaja radica en que el numero de elementos es dinamico, es decir, que puede variar. Son objetos y se tratan como tal.

Hay una colección que se llama **vector** muy usada hasta hace poco pero actualmente se desaconseja su uso. No pertenece al collection framework y es la más antigua.

Las colecciones tienen un comportamiento común definido por las interfaces List, Tree, Map y Queue.

Set/List: los elementos se numeran desde el cero y siempre nos moveremos por la posición del elemento en la colección. Si se elimina un elemento, se reenumeran las posiciones. Muy parecido a una tabla.

```
List lista = new ArrayList(); Empieza con un numero minimo  
List lista = new ArrayList(int); Elementos con lo q se inicio  
List lista = new ArrayList(int, int); num elementos, factor de  
crecimiento
```

Como List es una interfaz y no tiene constructor hay usar el de alguna clase que lo implemente como por ejemplo ArrayList.

Map: Es un conjunto de dos colecciones relacionadas de forma Clave – Valor. En ellos nos movemos accediendo por el nombre de la clave. Por ejemplo, si quiero hacer una colección de facturas, las guardare por su numero de factura, sin importar su posición. Para meter un valor tendre que meter 2 objetos (clave-valor) y para sacarlo necesitare la clave.

```
Map mapa = new HashMap();
```

Tree: Es una estructura de arbol.

Queue: Es como una pila (stack) de tipo FIFO (JDK 1.5 o superior)

Las colecciones **solo almacenan objetos**, NO primitivos. Si quisiéramos almacenar objetos ha de ser a través de su wrapper, por ejemplo int – Integer..., etc.

Cuando se declara una colección se puede hacer sin argumentos, con 1 argumento o con 2 argumentos.

Si la colección no lleva argumentos se creará con un número determinado por defecto de argumentos.

Si lleva **un argumento** se tratará del numero de elementos para la creación de la colección. Eso significa que cuando la colección alcance ese límite se añadirán más elementos que por defecto

serán el 75% del número indicado en ese parámetro.

El **segundo parámetro** es el “**factor de crecimiento**” indicaría cuantos elementos hay que añadir exactamente cada vez que se alcance el límite establecido en el primer parámetro.

Para **agregar elementos** se usa el método `add();` y se añade el elemento en el primer sitio libre que encuentre. Si no hay sitios libres añadirá elementos según el “factor de crecimiento”.

Para **colocar elementos** se usa el metodo `set(posición, objeto);` que devuelve un objeto. Tenemos que especificar en que posición queremos que se situe. Esa posición debe existir y, si esta ocupada, sustituirá al elemento que la ocupa.

Para **acceder a un elemento** se usa `get(int);` donde int es la posición. Devuelve el contenido de tipo Object. Así que hay que castear al tipo que nos interesa.

Para **eliminar** un elemento se usa `remove(int);` donde int es la posición. Devuelve el contenido de tipo Object, con lo cual hay que castear. Cuando se elimina un elemento se elimina también su posición, con lo cual la posición de los que van detrás cambia.

Para **eliminar** el contenido de un elemento sin eliminar la posición utilizaremos `set(pos, null);`

Para limpiar la tabla `Clear();`

Para saber si esta vacía o no `isEmpty();` (bool)

Para que nos cree una tabla con el contenido de la colección `toArraay();`

Los **Maps**, como no trabajan por posición, tiene alguna diferencia. Por ejemplo, para añadir se usa `put(clave, valor);` donde clave y valor son ambos Object. Para obtener un valor se usa `get(clave);` se obtiene un object con lo cual hay que castear. Si esa clave no existiera devolvería null.

Otros métodos son `remove(clave);`

`Clear();`

`Size()`

`isEempty();`

Como las colecciones pueden tener elementos vacíos, se usan **iteradores** para recorrer la colección solo por los elementos NO vacíos.

El objeto `Iterator();` tiene dos métodos:

`hashNext();`: Recorre la colección completa, devuelve true mientras haya elementos o false cuando ya no hay más elementos.

`next();`: Pasa al siguiente elemento de la colección y devuelve su contenido como Object, con lo cual hay que castear.

Un ejemplo de uso seria:

```
Iterator ite = new Lista.Iterator();
    While (ite.hasNext())
        (casteo)ite.next();
```

En el caso de los **Maps**. Los **iteradores** necesitan saber sobre qué colección se va a iterar (clave o valor), para eso se usa **Set map.keySet** para las claves. Se usa de esta forma.

```
Set listaClaves = Mapa.keySet();
Iterator ite = listaClaves.iterator();

En una linea
Iterator ite = Mapa.keySet().iterator();

Whlie (iteHashNext()){
    listaClaves.get(ite.next());
}
```

O con for ...

```
For(String clave: ite){
    listaClaves.get(clave);
}
```

Set es una subclase de List que representa las claves del mapa.

A partir del **JDK 5** se usan los bucles **FOR mejorados** para iterar sobre las colecciones, hacen los mismo que los Iterator, recorren los elementos NO vacíos, y además los castean, etc. Un ejemplo seria:

```
for (tipoSalida nombre : colección sobre la que iterar)
for (String texto:Lista);
```

Seguridad de Tipos

A partir del JDK 5 puede usarse seguridad de tipos en las colecciones, esto quiere decir que podemos obligar a que en una determinada colección, solo se usen elementos de un tipo determinado. Se usa poniendo entre ángulos (< y >) el tipo de la colección. Por Ejemplo:

```
List<String> miListaDeStrings = new ArrayList<String>();
```

Si algún elemento no es de ese tipo lanza una excepción en tiempo de compilación.

En el caso de un Mapa se usaría así:

```
Map<tipo, tipo> miMapa
```

```
HashMap<String, String> mapa = new HashMap<String, String>();
```

y si los valores fueran colecciones seria así

```
Map<tipo, ArrayList<tipo>>
```

FORMATOS

Es necesario asegurarse de que la información introducida por los usuarios en nuestros formularios es correcta.

Java tiene unos mecanismos para controlar esto. Se trata del componente **JFormattedTextField**.

Permite controlar números, fechas, y otros formatos personalizados como el NIF, códigos postales, números de teléfono, etc.

La sintaxis es:

```
JFormattedTextField xx = new JFormattedTextField(object);
```

Donde object es el objeto que se va a comprobar (un NumberFormat, SimpleDateFormat). Los JFormattedTextField hay que pasarle siempre un objeto con el metodo .setValue(obj);

Formato Numérico

Para formato de números se usa la clase **NumberFormat**.

Es una clase sin constructor con miembros static. Sus métodos son:

```
getInstance(); Devuelve un formato numérico de acuerdo a la cultura local (locale).
getCurrencyInstance(); Formato de moneda según la cultura.
getPercentInstance(); Formato de porcentaje.
```

Se puede establecer el tamaño de la parte entera y el tamaño de la parte decimal a traves de cuatro métodos:

```
Obj.setMaximumDigit(int);
Obj.setMinimumDigit(int);
Obj.setMaximumFractionDigit(int);
Obj.setMinimumFractionDigit(int);
```

Para que el formato numérico funcione **debe llevar un valor inicial** (`campo.setValue(new Integer(int));`) que se corresponda con el formato establecido.

```
.parse("") -> Lumber
.format(double o long) -> String
```

Formato Fecha

En Java las fechas y las horas son objetos **Date**.

Es un número de tipo long en milisegundos (Unix Time). Por lo tanto a las fechas hay que darles formato de tipo fecha y/o hora.

```
Date fecha = new Date();
```

Contiene el valor en milisegundos del momento en que se instancia. Tiene muchos métodos deprecated.

La clase **Calendar** se creó para manipular fechas según la cultura.

Es una clase abstracta que debe ser heredada y sus métodos implementados. Solo la tiene implementada la clase **GregorianCalendar**.

```
GregorianCalendar fecha = new GregorianCalendar();
```

.getTimeInMillis(); Devuelve long y da la hora en milisegundos.

.add(MAG, Val); Añade o resta según que simbolo se utilice, la magnitud indicada por una constante de la clase calendar. Es void, lo modifica pero no devuelve nada.

```
.getMonth();
.getDay();
.setMonth(int);
.setTime(date);
.setTimeInMillis(long);
```

Tiene las clases necesarias para funcionar con fechas. **DateFormat**, **SimpleDateFormat**, tienen métodos de conversión String-Fecha, etc.

```
SimpleDateFormat fecha = new SimpleDateFormat("String");
```

Donde “String” tiene el formato que le queremos dar a la fecha siguiendo los siguientes códigos:

TABLA DE FORMATO DE FECHAS Y HORAS	
FECHAS	HORAS
G Era (A.C / D.C)	a a.m / p.m
y Año (yyyy) dígitos	H Hora del día (0 – 23)
M Mes (dígitos) MM o MMM o MMMM(nombre)	k Hora del 1 al 24 (no usar el reloj puede llegar a las 24:59)
w Semana del año (número 1-52)	K Hora de 0 a 11 (formato de 12 horas)
W Semana del mes (número 1-5)	h Hora de 1 a 12 (12 horas)
D Día del año (1 – 365) DDD	m Minutos
d Día del mes (1 – 31)	s Segundos
F Número del día de la semana	S milisegundos
E Nombre del día de la semana	z Desplazamiento horario con nuestra zona horaria
	Z La hora más la diferencia de horas

SimpleDateFormat tiene los métodos de conversión:

`parse(String)` → Recibe String y devuelve Date
`format(Date)` → Recibe Date y devuelve String

Para sumar o restar dos fechas, se opera con ella (son long) y luego se convierte al formato que convenga.

Formato Personalizado

Se crea a partir de objetos de la clase **MaskFormatter**. Deben ajustarse a la máscara que le demos.

```
MaskFormatter mascara = new MaskFormatter("String");
```

Donde “String” indica el número de caracteres y el tipo, según esta tabla:

= Número
U = Letra mayúscula
L = Letra minúscula
A = Cualquier carácter
? = Cualquier carácter
* = Cualquier carácter (incluidos los especiales, fn etc...)
H = Numeros hexadecimales
Este formato lanza la excepción **ParseException**.

El problema del MaskFormatter es q cuando el usuario no llena el campo con todos los caracteres, no es valido.

Este formato solo funciona si le decimos que no valen los caracteres especiales, así:

```
mascara.setAllowInvalid(false);
```

El método **setPlaceHolder** añade una ayuda visual extra al usuario, mostrando en el JFormattedTextField unos caracteres que insinuan la cantidad de caracteres admitida en ese campo.

```
setPlaceHolder("String");// Se le pasa el string que queremos mostrar en el text field  
setPalceHolderCharacter('char');// Se le pasa el carácter que queremos que aparezca como ayuda para el usuario en cada posición.
```

Para controlar la cantidad máxima de caracteres admitida en un textBox hay que hacerlo por programación en el “modelo” del TextBox.

Todos los componentes de texto tienen un “modelo” de la clase **Document**. Cada vez q se introduce un texto DocumentFilter recibe un objeto y si todo es correcto, lo manda al modelo. Para ello necesitaremos dos metodos.

`campo.getDocument();` nos da ese modelo, hay que añadirle un evento “**abstractDocument**”.

El código es así:

```
AbstractDocument modelo = (AbstractDocument)campo.getDocument();
modelo.setDocumentFilter(new claseManejadora(posiciones));// Donde
claseManejadora es la clase que maneja y filtra los caracteres
```

Esa clase manejadora debería ser algo así:

```
public class claseManejadora extends DocumentFilter
public void insertString(FilterByPass textoAnterior, Int posicion,
String textoNuevo, AttributeSet xx) Throws BadLocationException
```

InsertString es el método que se llama cuando el usuario escribe o copia.

Los cuatro parámetros que recibe son:

- 1.- El texto que había antes en el modelo.
- 2.- La posición dentro del texto anterior donde irá el texto nuevo
- 3.- El texto nuevo
- 4.- Los atributos del texto. Color, tipo de letra, tamaño, etc.

En este método se escriben las condiciones por las que se acepta o no el código. Si el texto es correcto entonces se llama al método `insertString` del padre (`super.insertString();`) con sus cuatro parámetros.

Si el texto no es correcto se avisa al usuario de alguna manera, por ejemplo con un beep (`toolkit.beep();`), o con un mensaje, etc.

DocumentFilter tiene otro método que se llama **replace**. Lo usamos solo si quisieramos reemplazar algo.

```
replace(FilterByPass, Int posicion, int tamañoAReemplazar, String texto,
Attributes xx);
```

Dentro de este método comprobamos las cosas y luego se llama al método `replace` del padre pasándole todos los argumentos: `super.replace(-,-,-,-);`

```
public class claseManejadora extends DocumentFilter{

    private --- numCaracter;
    public void insertString(FilterByPass textoAnterior, Int
    posicion, String textoNuevo, AttributeSet xx) Throws
    BadLocationException{

        if(-----){
            super.insertString();
        }

        }else{
        }
    }

    Public void replace(FilterByPass, Int posicion, int
    tamañoAReemplazar, String texto, Attributes xx){
```

```

        if(-----){
            super.replaceString();
        }

    }else{
}

}

```

Ahora tenemos que crear un clase para enlazar el documento de texto con el document. Solo tendra un metodo.

```

Public Class tamañoCampo{
    Public Static void metodo(JTextComponent xxx, int _ _ _){
        ad = (AbstractDocument).componente.getDocument();
        ad.setDocumentFilter(new xxx(_ _ _))
    }
}

```

Preparada la clase, hacemos la llamada en nuestro programa

```
tamañoCampo.metodo(obj, 15);
```

Validaciones

Pattern (patron), tiene el metodo compile();
 Matcher (comprobador)

```
pattern.compile("expresion");
```

Expresiones:

Simple: [XmBb9]

Rangos: [a-z A-Z] [0-9] [^a-zA-Z] (^=no permitido y afecta a a-z)

Union: [a-c[c-p]]

Intersección: [a-c &&[c-g]] (solo vale c)

Substracción: [a-c &&[^c-g]] (no vale c)

Símbolos: [-:/]{1} permitido 1 vez unos de los 3 símbolos

Expresiones abreviadas:

\w = caracteres (letras y números)

\W = no caracteres (ni letras ni números)

\d = dígitos

\s = espacio

\S = no espacio

Operadores:

?= uno o ninguno

*= cero o infinito

+= uno o infinito

{numero de veces}

{num min, (espacio en blanco)} solo se dice el nmero minimo de veces

{min num veces, max num veces}

PERSISTENCIA

Es la capacidad de que las cosas permanezcan en memoria incluso después de terminado el programa, pero a su vez también se refiere a poder recuperar la información del mismo (leer) para que pueda ser nuevamente utilizada.

En Java esto se hace a través de la **serialización, los motores de persistencia y las bases de Datos**. La información puede guardarse de forma binaria, como texto, o como XML. A través de Ficheros de Texto o Bases de Datos.

Serializar es grabar el estado de un objeto en un fichero. El estado es el contenido de las propiedades de clase.

Ficheros

Al hablar de ficheros nos referimos a ficheros de texto que pueden tener diferentes formatos según nos convenga. Algunos de los mas usados son los textos planos,

TEXTO PLANO

Se usa la clase **File** para trabajar con archivos de texto plano.

```
File xx = new File("ruta o nombre fichero");
```

Puede apuntar a un directorio o a un archivo, exista o no. En caso de no existir no dara errores, los creara.

La clase File tiene algunos métodos informativos para saber las propiedades del fichero. Estos son:

canRead(), canWrite(), canExecute(). Devuelven Booleano.

Los métodos para saber si el File es un directorio o un archive son:

isDirectory() y isFile(), tambien devuelven Booleano. Si el archivo o directorio no existe devuelve false.

El metodo **isHidden()** nos devuelve un booleano en función de si el archivo o directorio esta oculto o no.

Los siguientes dos métodos necesitan que la ruta del file exista para funcionar.

El método **File[].list()** me devuelve un array de objetos File con directorios y archivos.

El método **File[].listFile()** devuelve un array de objetos File con archivos.

El método **File[].listFile(Filefilter)** devuelve un array de objetos File con archivos, según un filtro.

Si el método **file()** lo combinamos con el método **.listRoots()** obtenemos un array de las unidades de disco.

Sí apuntamos a un directorio y no existe y lo queremos crear tenemos los métodos **mkdir()** y **mkdirs()**. mkdir solo crea la última carpeta y mkdirs crea toda la trayectoria de directorios. Devuelven boolean si se ha realizado la operación con éxito o no. Estos funcionan dependiendo de que el sistema operativo nos haya dado permiso para crear directorios.

Para borrar archivos y directorios se usa el metodo **.delete()**, devuelve true o false dependiendo de si ha podido hacerlo o no. En el caso de un directorio delete solo funciona si este está vacío. El OS es el que nos da permiso o no para borrar. Hay una variante de borrado solamente para archivos que es **deleteOnExit()**, lo que hace es borrar el archivo cuando el programa termina.

Para saber si un archivo o directorio existe tenemos el metodo **exists()**, devuelve booleano.

Tambien existe un método static que sirve para crear ficheros temporales, se llama **createTempFile("nombre", "extension", ["ruta"]);** tres argumentos, el tercero optativo, si no aparece lo crea en la carpeta que el operativo tenga para archivos temporales.

Para crear un fichero vacío se usa **createNewFile()**, no es static. Devuelve booleano según se haya podido crear o no. El nombre y la ruta se establecen en el constructor lógicamente.

.getTotalSpace() = El espacio total de la unidad en bites.

.getFreeSpace() = El espacio libre que tiene la unidad a la que estamos apuntando en bites.

.getUsableSpace() = El espacio disponible en la unidad a la que apuntamos en bites.

.length() = Devuelve el tamaño del archivo en bites.

.lastModified() = Devuelve la fecha (long = unix time) de la ultima modificación, un objeto Date.

.setWritable(), .setReadable(), .setExecutable(), .setReadOnly(), .setLastModified(). Devuelven Boolean.

Un fichero File puede apuntar a ficheros de otra máquina, la ruta no sería un String sino un objeto **URL**.

LECTURA Y ESCRITURA DE ARCHIVOS DE TEXTO

Leer:

Existen las clases **FileReader** y **FileWriter**. El archivo debe existir para poder leerlo o escribirlo.

```
FileReader xx = new FileReader("nombre o ruta");
FileWriter xx = new FileWriter("nombre o ruta");
```

El argumento puede ser un string, pero no es recomendable, sera mejor utilizar un objeto File. Si el objeto no existe, lanza excepciones IOException. Para que esto no suceda utilizamos un objeto FileChooser.

```
File fichero = obj.getSelectedFile();
FileReader lector = new FileReader(fichero);
```

```

BuefferedReader lector2 = new BufferedReader(lector);

String textoLeido = "";
StringBuffer textoTotal = new StringBuffer();

While(textoLeido != null){
    textoLeido = lector.ReadLine();
    textoTotal = textoTotal.append(textoLeido + "\n");
}
Texto = textoTotal.toString();
Lector.close();

```

El unico formato que tiene un archivo de texto plano es el intro. La clase capaz de leer hasta el intro es BuffereReader. Este nos proporciona trozos de texto, pero para unirlos todos en un objeto de texto con el total y a la vez evitar el alto coste de crear y destruir objetos String; utilizamos la clase StringBuffer, que si permite modificar Strings. Con un bucle vamos leyendo las partes y pegandolas una detrás de otra y almacenandolas en el objeto String buffer hasta que no hay mas texto. Entonces sale del bucle y convertimos el objeto StringBuffer en un String con .toString(); Al terminar de utilizar el archivo tendremos que cerrar el flujo para liberarlo y que otro usuario lo pueda usar.

Escribir:

Existen los conceptos **BOF** y **EOF**, principio del archivo y fin del archivo. Son como punteros.

El metodo **write()** introduce el texto en el punto donde se encuentre el cursor (BOF). Se usa con **FileWriter** y sobreescribe a partir de ese punto.

Existe el metodo **append()**, lo que hace es añadir al final, es decir a partir de donde apunte **EOF**. Existe desde el JDK 5. se usa con **FileWriter**.

Estos metodos de escritura no escriben en el momento en que se escriben sino cuando el buffer se ha llenado. Para poder escribir cuando le decimos hay que usar **flush()**, esto vacia el buffer. El metodo **close()**, cierra el archivo y lo desbloquea pero no vacia el buffer, con lo cual **antes de cerrar hay que hacer un flush()**.

Para escribir cadenas de caracteres debe usarse la clase **BufferedWriter**.

Cuando hayamos terminado con el fichero SIEMPRE hay que cerrarlo.

```

FileWriter flujo = new FileWriter(File);
flujo.write(objTextArea.getText());
flujo.flush();
flujo.closed();

```

PROPERTIES

Java tiene su propio formato de archivos de texto, los ficheros de tipo properties. Estos siempre llevan la extensión .properties. Siempre se guardan en las carpetas de las clases, es decir en los paquetes donde se guardan las clases.

El formato de los properties sigue un patron de tipo clave – valor.

Un ejemplo de linea seria: **clave = valor**

Se recomienda que las claves tengan una cierta jerarquía y orden. No debe haber claves repetidas.

Se usan mucho para configuración de programas y para internacionalización.

El uso en internacionalización requiere los archivos de recursos idiomáticos en formato properties.

Primero se crea el archivo en español con sus claves y valores, después se copia ese archivo en el archivo de otra idioma en el que solo se cambian los valores al idioma al que se va a traducir.

Una buena forma crear las claves es esta:

NombrePantalla + tipo componente + nombre componente

Cuando lo usemos para internacionalizar, el nombre del archivo properties debería terminar con el código ISO del idioma antes de la extensión. Por ejemplo, para español seria nombreArchivo_ES.properties, para inglés sería nombreArchivo_EN.properties, etc.

Es buena idea preparar siempre nuestras aplicaciones usando la internacionalización aunque no vayan a traducirse.

Estos archivos properties suelen ser de solo lectura.

La clase **ResourceBundle** es la encargada de **leer** estos archivos a través de un método static llamado **getBundle()**

```
 ResourceBundle rb = ResourceBundle.getBundle("ruta y  
nombre.properties");
```

Para aplicar el texto a un componente hay que leerlo del archivo y luego aplicarlo así:

```
JLabel.setText(rb.getString("clave"));
```

Eso mostrara el texto en el idioma elegido en el objeto ResourceBundle (rb).

Hablaremos más sobre internacionalización a lo largo del master.

Log4J

Properties con Ficheros de Texto para creacion de Logs. Recoge texto y lo trata. Establece unos niveles y en función de los receptores recogera unos mensajes u otros.

Existe una herramienta de código abierto para tratar con archivos de texto llamada, **Log4J**.

Puede encontrarse, al igual que muchos otros proyectos de código abierto en la web de Apache, www.apache.org

Es muy habitual registrar en logs (ficheros de bitácora) la actividad de nuestra aplicación. Para esta actividad la herramienta Log4J es muy útil.

Hay que: Descargar, añadir al proyecto la dependencia y configurar el descriptor.

Log4J tiene dos clases:

Logger → Crean los logs. Emite los mensajes. Establecen un filtro de nivel.

Appender → En función del nivel establecido por el Logger, el appender lo trata o no.

Filtrar por nivel es muy útil para no tener en cuenta los mensajes de bajo nivel una vez que la aplicación ya está en producción.

Los niveles, de menor a mayor, de los mensajes son:

Debug

Info

Trace

Warn

Error

Fatal

Además existen distintos tipos de appender según el uso que vayamos a dar a los mensajes lanzados por el Logger: Los nombres los elegimos nosotros como queramos, por ejemplo:

Consola

Ficheros

Correo Electrónico

Base de Datos

Los mensajes tienen distinto formato según nos interese.

Todo esto lo hace Log4J sin necesidad de programar, simplemente hay que configurarlo a través de un archivo tipo properties.

El mecanismo interno que usa Log4J es el mecanismo de **Reflection**. Reflection es un mecanismo

de java que permite manipular objetos sin conocerlos. Es un mecanismo de servidor. Lo veremos más adelante en el master.

Log4J se puede usar en cualquier parte del código de esta forma:

Si se va a mandar mensajes desde una clase, se tiene que registrar la clase:

(Usando el Logger de apache.org, no el de Java.Util)

```
Logger log = Logger.getLogger(NombreClase.class);
```

De esa forma hemos **registrado** la clase. Para enviarle los mensajes se hace así:

```
nombreMetodo() {  
    log.Debug("String - mensaje a enviar");  
    log.Info  
    log.Trace  
    etc  
}
```

Pueden usarse los booleanos **IsDebugEnabled o IsInfoEnabled**, etc en un if para prevenir el uso de filtros de mensajes que no están configurados.

Para configurar el Log4J se usan archivos de tipo properties, como ya dijimos. Log4J tiene el archivo Log4J.properties y para que lo encuentre directamente hay que ponerlo en la raíz del proyecto, no dentro de los paquetes, sino fuera.

En el archivo properties primero hay que decirle cuantos appenders va a tener, usa este formato para la clave:

log4j.properties

Log4J.rootCategory – Aquí va el nivel del appender, es decir Debug, Trace, etc. Separado con comas van los appenders por tipo, es decir consola, Fichero, etc.

Por ejemplo: `Log4J.rootCategory = Debug, Consola, Fichero`

Este ejemplo crea dos appenders (Consola y Fichero) que admiten todos los niveles ya que Debug es el nivel más bajo.

Después se define el appender:

```
Log4J.Appender.Consola = org.apache.Log4J.ConsoleAppender
```

Este appender mostraría los mensajes por consola.

A continuación se le da formato al texto:

```
Log4J.appenders.consola.layout = org.apache.log4j.PatternLayout
```

Con esto le damos el formato al mensaje según estos patrones:

```
Log4j.appenders.consola.layout.conversionpattern = %m
```

La **m** saca el texto del mensaje, con lo cual es obligatoria y siempre seguida del símbolo **%**

%m – saca el texto del mensaje

%n – retorno de carro, va siempre al final, cuando sea necesario escribir un retorno de carro

%p – nivel del mensaje con el que se ha lanzado el mensaje
%d – para indicar fecha y hora: **%d{dd/mm/yyyy}** como indicar el formato de Date
%x – La clase que ha lanzado el mensaje
%f – El metodo que ha lanzado el mensaje
%l – La linea del codigo que ha lanzado el mensaje
%-5 – Cantidad de caracteres que debe ocupar. Combinado con la **%n** cortaria lo que sobra.

Si se pone un numero entre el signo % y la letra estamos indicando el tamaño fijo de caracteres que va a tener el texto. Sirve para tabular el log de forma que tenga un formato uniforme. Si el numero lleva un signo mas (+) por delante se colocarian espacios en blanco por detrás y si el signo es menos (-) los coloca por delante, en el caso de haberle dado más caracteres de los que necesita.

Para los **ficheros** se hace asi:

```
Log4J.appenders.Fichero = org.apache.Log4J.FileAppender  
Log4J.appenders.Fichero = org.apache.Log4J.RollingFileAppender  
Log4J.appenders.Fichero = org.apache.Log4J.DailyRollingFileappender
```

FileAppender obliga a un mantenimiento porque el fichero tiene un limite de capacidad. Si no le pongo tamaño, será el maximo permitido por el operativo.

RolligFileAppender, permite definir copias de seguridad al llegar al maximo de la capacidad del fichero.

DailyRollingFileAppender genera un fichero de log diario. Añade la fecha al nombre del fichero. Hay que darle el formato de la fecha, siendo buena idea yy-mm-dd para el formato, ya que así se ordenan alfabeticamente.

Para añadir la fecha al archivo (RolligFileAppender o DailyRollingFileAppender) se usaria esto:

```
Log4J.appenders.Fichero.DatePattern = '.' yyyy-mm-dd
```

En cualquiera de estas tres opciones hay que darle la ruta absoluta y el nombre del fichero de esta forma:

```
Log4J.appenders.fichero.File = "ruta y nombre de archivo"
```

Logicamente hay que asegurarse de que se tienen permisos de escritura y de que esa ruta es valida. Lo mejor es usar una variable de entorno, asi:

```
 ${variable de entorno}xxxxx.log
```

Es decir: `Log4J.appenders.fichero.File =${variable de entorno}xxxxx.log`

InmediateFlush y Append son dos propiedades que se usan para escribir el fichero, pueden usarse con cualquiera de las tres opciones vistas anteriormente. El flush envia el flujo y append añade el texto al final. Se usan con un booleano como valor.

```
Log4J.appenders.fichero.InmiedateFlush = true o false
```

```
Log4J.appendender.fichero.Append = true o false
```

Inmediate flush con true envia el mensaje cada vez que llega y con false lo envia cuando se llena el buffer.

Append con true añade la informacion al final de fichero y con false lo sobreescribe.

Dependiendo del tipo de appendder que vayamos a usar puede ser necesario darle mas opciones como cuentas de correo electronico, etc.

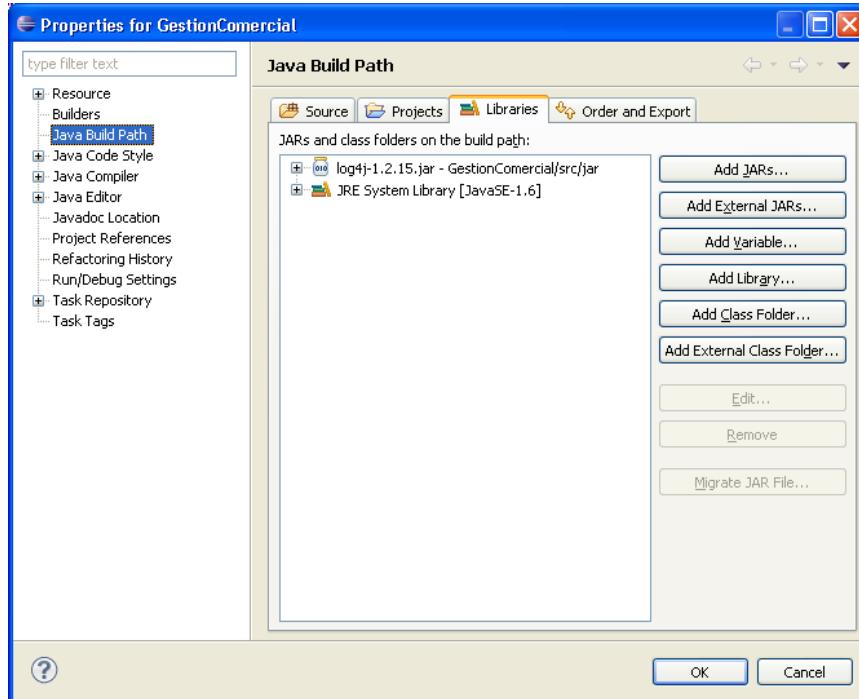
Como añadir referencias externas al proyecto con Eclipse

Para añadir el Log4J primero hay que bajarlo de la web de Apache. Después se añade un paquete llamado por ejemplo Jar y se arrastra a el.

Después con el boton derecho sobre el nombre del proyecto, propiedades, se abre la ventana de Propiedades, se elige la opcion de Java Build Path, pestaña Libraries, se pulsa Add JARs..., se elige el proyecto en el que habias colocado previamente el paquete jar y se pulsa OK.

De esta forma se creara un “paquete” Referenced Libraries que muestra la librerias externas importadas.

Esto vale para añadir cualquier tipo de referencia externa, incluidos los drivers para la base de datos.



Ficheros XML

Un fichero XML es texto plano. Se compone de etiquetas encerradas en ángulos <>. Las etiquetas son creadas por nosotros siguiendo unas reglas. El fichero puede tener el nombre y la extensión que queramos, aunque como extensión suele usarse .xml.

Los ficheros XML siempre comienzan por un encabezado de dos líneas que indican las características generales del fichero. Este encabezado de dos líneas se llama **prólogo**.

```
<?XML version="1.0" Encoding="xxxxxx"?> // El encoding es el conjunto de caracteres
<!DOCTYPE nombreDocumentoXML (System=/Public=) ruta+nombreDTD>
```

Version indica la versión de XML, que de momento es la única versión.

Encoding indica que codificación estamos utilizando. UTF-8 (Europa occidental), o iso-8859-1(para el Catellano)

Las etiquetas XML pueden llevar atributos.

Los archivos XML deben seguir un patrón que se define en otro documento llamado **DTD o XSD**, y el archivo debe ajustarse a este DTD.

La segunda línea del prólogo define la validación del documento XML y se encarga de indicar qué DTD va a usarse. Si para la validación usamos XSD no se indica en la segunda línea del prologo. Si el archivo DTD está en nuestra maquina se usa **System** para indicarlo, si está en otra máquina se usa **Public**. A continuación va la ruta al archivo DTD. Si se usa un DTD externo, a continuación de la Palabra Public se usa un texto con dos partes la primera es trivial e indica quien ha hecho ese fichero(persona o empresa) y la segunda es una URL que indica donde esta el fichero. No tiene que especificar una ruta física, pero si una cadena única para que no pueda ser validado por otro fichero

En Java el archivo XML y el DTD deben estar dentro del proyecto.

Después del prólogo va el cuerpo del documento XML.

El cuerpo lleva una estructura con etiquetas de apertura y cierra. De hecho el cuerpo lleva una etiqueta de apertura y otra de cierre que es la etiqueta raíz, y dentro de esas dos van las demás etiquetas. Todas las etiquetas deben cerrarse <loquesea></loquesea>.

Cuando las etiquetas llevan atributos estos deben ir en la etiqueta de apertura, y pueden ser tantos como necesitemos. El formato de los atributos es *nombre="valor"* y se separan por un espacio, los valores deben ir siempre entrecomillados con comillas dobles o simples.

Se puede indicar si un atributo es obligatorio u optativo. Obligatorios son lo que hay que poner al crear la etiqueta. También se puede indicar si un atributo tiene un valor por defecto o no, y se pueden limitar los valores.

El cuerpo de una etiqueta es lo que lleva entre su etiqueta de apertura y de cierre. El cuerpo puede estar vacío. O puede indicarse si el cuerpo es optativo u obligatorio. El cuerpo de una etiqueta puede incluir otras etiquetas.

Las etiquetas se cierran en el orden inverso en el que se abren. Por lo tanto la primera que se abre es la última que se cierra y viceversa. Es decir, que van anidadas.

Configuración.XML

```
<?xml version="1" encoding="iso-8859-1" ?>
<!DOCTYPE opciones.configuracion system="/configuración.DTD">
<opciones_configuracion version=" " >
<fecha_actualizacion>_____</ fecha_actualizacion >
<autor>_____</autor>
<propiedades>
    <opciones_usuarios>
        <idioma preferido="__" defecto="__" />
    </opciones_usuarios>
    <permisos>
        <permitir_idioma permiso="si" />
        <modo_ejecucion modalidad="p" />
    </permisos>
    <rutas>
        <ruta_bitacora>_____</ruta_bitacora>
        <ruta_ayuda>_____</ruta_ayuda>
    </rutas>
</propiedades>
</opciones_configuracion>
```

El archivo DTD debe llevar la extensión .DTD. Al igual que el archivo XML, también tiene un **prólogo de una línea**, la primera.

Tiene dos etiquetas únicamente; **ELEMENT** y **ATTLIST**.

La etiqueta **ELEMENT** indica cada etiqueta que se va a definir en el documento XML:

```
<!ELEMENT nombreEtiquetaRaizDelDocumentoXML (cardinalidad
contenidoEtiquetaRaiz) >
```

Una de las cosas que se indican dentro de ELEMENT es la cardinalidad, es decir el número de veces que se va a repetir un elemento, existen tres opciones de cardinalidad:

- Una vez ()
- Una o infinitas (*)
- Ninguna o infinitas (?)

La opción por defecto es “una vez” si no ponemos nada detrás del elemento será la opción elegida, para indicar la opción de “una o infinitas” se usa **asterisco *** y para la opción “ninguna o infinitas” se usa la **interrogación ?** las **opciones se escriben después del nombre sin espacio entre ellos**.

Para indicar que se puede incluir cualquier texto se usa #PCDATA para un ELEMENT sin cuerpo se usa EMPTY

Para indicar los atributos se usa la etiqueta ATTLIST, a continuación de la palabra ATTLIST va el nombre de la etiqueta, luego el nombre del atributo, a continuación separado por espacio, se usa CDATA (para indicar que se puede escribir cualquier cosa), para acotar la información se pone entre paréntesis las opciones literales separadas por comas. Si es un valor obligatorio se pone #REQUIRED, si no se pone nada es que el atributo es optativo también se puede especificar con #IMPLIED.

Un ejemplo sería

Configuración.DTD

```
<?XML -----?>
<!ELEMENT opciones_configuracion (fecha_actualizacion, autor,
propiedades)>
<!ELEMENT fecha_actualizacion (#PCDATA) >
<!ELEMENT autor (#PCDATA) >
<!ELEMENT propiedades (opciones_usuario, permisos, rutas) >
<!ELEMENT opciones_usuario (idioma_preferido) >
<!ELEMENT idioma_preferido EMPTY>
<!ATTLIST opciones_configuracion
    version CDATA #IMPLIED "valor x defecto" >
    (---|---|---) #REQUIRED >
```

Otro ejemplo:

Archivo XML:

```
<?XML version=1.0 Encoding="ISO-8859-1"?>
<!DOCTYPE consultaClientes (System/Public) ruta+nombreDTD>
<consultaClientes>
    <cliente>
        <nombre>sdfsfsfs</nombre>
        <direccion>
            <poblacion>cccccc</poblacion>
            <provincia>xxxxxxxx</provincia>
        </direccion>
    </cliente>
    <cliente></cliente>
    <cliente></cliente>
    <cliente></cliente>
</consultaClientes>
```

Archivo DTD:

```
<?XML version=1.0 Encoding="ISO-8859-1"?>
<!ELEMENT consultaClientes (? Cliente) >
<!ELEMENT cliente (nombre, *direccion) >
<!ELEMENT nombre (#PCDATA) >
<!ELEMENT direccion (poblacion, provincia) >
<!ELEMENT poblacion (#PCDATA) >
<!ELEMENT provincia (#PCDATA) >
<!ATTLIST.....>
```

Validación con XSD

Con el XSD ganamos ya que permite una ordenación de las etiquetas y nos permite definir tipos.

XML Y JAVA

En J2EE se usan muy habitualmente en configuración.

Para tratar archivos XML en Java se usan las tecnologías **SAX ó DOM**, son dos APIs de Java desarrolladas por Apache, aunque Java también tiene sus propias APIs de SAX y DOM.

SAX (simple api xml) es la forma más simple y rápida de tratar archivos XML en Java.
DOM (document object model) se usa en escenarios web.

SAX es un tratamiento secuencial del fichero, lee las etiquetas una a una y no puede volver a una etiqueta ya leída, por lo que si necesito conservar alguna info la tendré que guardar. Por cada etiqueta leída se genera un evento. Es un tratamiento orientado a eventos muy rápido y ligero.

DOM lee todo el fichero XML y genera en memoria un árbol de objetos, donde cada objeto representa una etiqueta del fichero XML. Al tener todo en memoria nos podemos mover por él como queramos. Esto tiene alto coste de proceso y memoria. Por lo tanto habitualmente se usará SAX.

Tratamiento de ficheros con SAX:

Para tratar un fichero XML con SAX hay que leerlo, para ello necesitamos la clase **XMLReader** que se crea a partir de una **factoría**:

```
XMLReader lector = XMLReaderFactory.CreateXMLReader(); // Flujo de lectura
eventosMenuXML escuchadorEventos = new eventosMenuXML(); // Eventos
lector.setContentHandler(escuchador_Eventos); // listener
FileReader fichero = new FileReader("Ruta y Nombre"); // Fichero (comprueba existencia)
InputSource flujoEntrada = new InputSource("Ruta y Nombre"); // Fichero
lector.parse(flujoEntrada); // lee archivo y valida
```

A veces en vez de usar un string o objeto file, se puede utilizar un objeto URL que sirve para una ruta externa o interna. Y de esa forma simple podemos acceder a recursos externos o internos.

Las **factorías** se usan en algunas ocasiones cuando los objetos son difíciles o especiales de construir. Se construyen con métodos estáticos.

Para implementar los **eventos** en una clase se puede implementar una interfaz (DefaultHandler) o extender otra clase (DefaultHandler) para leer los atributos, lanzará excepciones de tipo SAXException:

```
public class eventosMenuXML extends DefaultHandler {
public void xxxx throws SAXException
    // Escuchador se lanza cuando comienza a leer (poco uso)
    startDocument(){};
```

```

    // Escuchador se lanza cuando ha terminado de leer (poco uso)
endDocument() {};
    // cuando se comienza a leer una etiqueta, obtenemos los atributos
startElement(URI, nombreLocal, nombreEtiqueta, ATTRIBUTES) {}
    // cuando se termina de leer una etiqueta
endElement(URI, nombreLocal, nombreEtiqueta) {}
// para leer el cuerpo de la etiquetas, llega el documento XML completo
String texto = characters(char[], int inicio, int fin);
}

```

Se llama a startElement y endElement tantas veces como etiquetas haya. Si se valida con DTD los tres primeros parametros de startElement y endElement son lo mismo.

ATTRIBUTES es un mapa de clave/valor.

En estos metodos se debe filtrar la implementacion por nombre de etiqueta, usando if.

Hay otros metodos menos usados como:

ignorableWhiteSpace() elimina los intos o espacios que no van dentro del cuerpo de ninguna etiqueta.

resolveEntity()

skipEntity()

DefaultHandler tambien incorpora tratamiento de errores, y este tratamiento incluye tres metodos:

warning(SAXException) - avisos

error(SAXException) – una etiqueta esta mal y no se procesa esa etiqueta

fatalError(SAXException) – algo va mal en el fichero y el fichero no se procesa

Tratamiento de ficheros con DOM:

(buscar metodo modificar_Configuracion)

Para modificar un XML con DOM tengo q hacer 3 cosas

1. leer el fichero para almacenarlo en memoria
2. buscar lo que tengo que modificar y modificarlo
3. escribir el fichero

CALIDAD

Es importante seguir las recomendaciones de nombrado para clases, metodos, etc.

Comentar el programa es cada vez mas exigente, llegándose incluso a pactar la cantidad de lineas de comentarios respecto a lineas de codigo.

Los comentarios pueden establecerse de varias formas pero los comentarios que cuentan en calidad son los de multiples lineas, es decir `/** */` este tipo de comentario es el exigido por **javadoc** para construir la documentacion en HTML.

Dependiendo de la posición de los comentarios, estos apareceran en un sitio u otro en la documentacion generada por javadoc.

Si van por delante de la clase, esos comentarios perteneceran a la clase.

Si van por delante de una variable de clase, sera el comentario que acompaña a esa variable de clase.

Si van por delante del metodo sera la documentacion asociada al metodo.

Este tipo de comentario admite etiquetas HTML.

JavaDoc tiene palabras reservadas que siempre comienzan por `@` para describir algunos conceptos sobre lo que estamos comentando.

Por ejemplo, en los metodos, se usa `@param` para describir un parametro del metodo, si el metodo devuelve algo se usa `@return` para describirlo. A nivel de Clase usariamos `@version` para la version, `@author` para el nombre del autor o `@since` para la fecha.

Existen herramientas para facilitar esta labor de los comentarios, como plugins para eclipse como **CodePro**.

JUnit

Además de los comentarios, en calidad se tienen en cuenta las **pruebas unitarias** (metodologías ágiles). Es decir, debe haber clases de prueba por cada clase del programa para comprobar si la clase funciona bien o no. La clase de prueba lleva los mismos métodos de prueba que métodos haya en la clase original. Con lo cual se duplica el proyecto solo para probar que funciona.

Para poder probar un método, debe cumplirse una condición que es, que devuelva un valor. Los métodos que devuelven void no se pueden testear con lo cual puede modificarse el código para que devuelva algo que se pueda testear.

Esto obliga a replantearse la forma de pensar cuando se va a desarrollar un método, buscando la forma de **probarlo antes de escribirlo**. TDD Test Driven Development, Desarrollo Dirigido a las Pruebas.

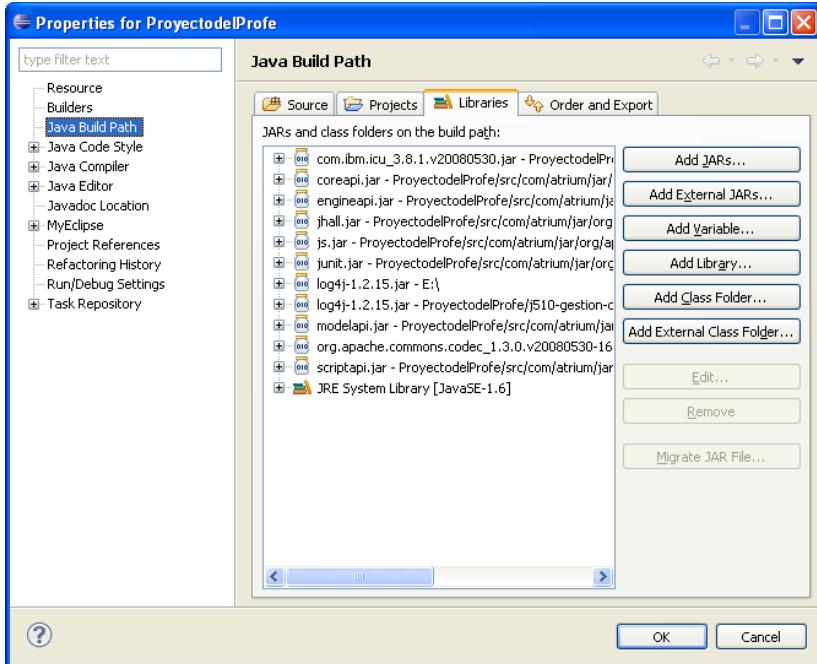
Existe un marco de trabajo para pruebas en Java que se llama **JUnit**, es una dependencia externa pero va incluido en Eclipse. Requiere organizarse el proyecto.

Para integrarlo, tal como se explica más arriba en el apartado “[Como añadir referencias externas al proyecto con Eclipse](#)”, pero lo vuelvo a explicar aquí.

En Eclipse hay que descargar el JAR (junit.jar) de la web (<http://www.junit.org/>) y copiarlo en algún paquete de nuestro proyecto. A continuación pulsando con el botón derecho del ratón sobre el nombre del proyecto se va a propiedades.

A continuación se abre otra ventana donde se elige la opción “Java Build Path” en el lado izquierdo

y la pestaña “Libraries” en el lado derecho.



Se pulsa el botón Add JARs... y se busca el archivo junit.jar en el paquete donde anteriormente lo hemos copiado. Cuando lo tengamos seleccionado pulsamos OK y así se copiara en el paquete de la aplicación “Referenced Libraries”.

Una vez hecho esto, lo lógico es crearse un paquete por cada paquete que se va a probar, con tantas clase como clases tenga el paquete de software. La nomenclatura debería llevar la palabra “test” por delante del nombre original.

También puede importarse un paquete llamado dbUnit que sirve para hacer pruebas contra bases de datos, es gratis y esta accesible en apache.

Eclipse trae dos versiones de **JUnit**, la 3.8 y la 4.2.

La 3.8 usa POJOS (plain old java objects), es un objeto de java sin anotaciones.
La version 4.2 sí trabaja con anotaciones.

Cómo escribir una clase de prueba:

```
Public class xxxx extends TestCase {  
  
    Public void testxxxxxxxx () {  
  
        ObjetoDeLaClaseATestear instanciado;  
        *Assert ObjetoDeLaclaseATestear.metodoATestear();  
    }  
  
}
```

* Los métodos assert siempre devuelven un booleano. Hay varios tipos de métodos assert, estos son

dos de los muchos posibles:

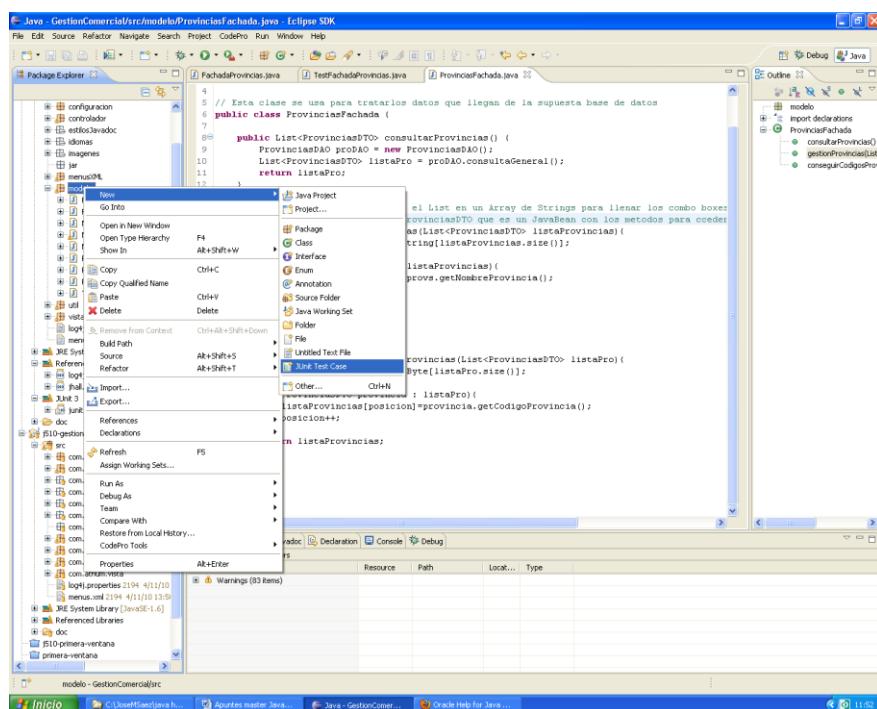
```
assertEquals(valor esperado, obj.metodo());  
assertNotNull(object); // comprueba si un objeto no esta vacio
```

Cualquier Assert puede llevar un **primer parámetro opcional**, el cual es un String con el texto que se muestra en caso de error. Así por ejemplo:

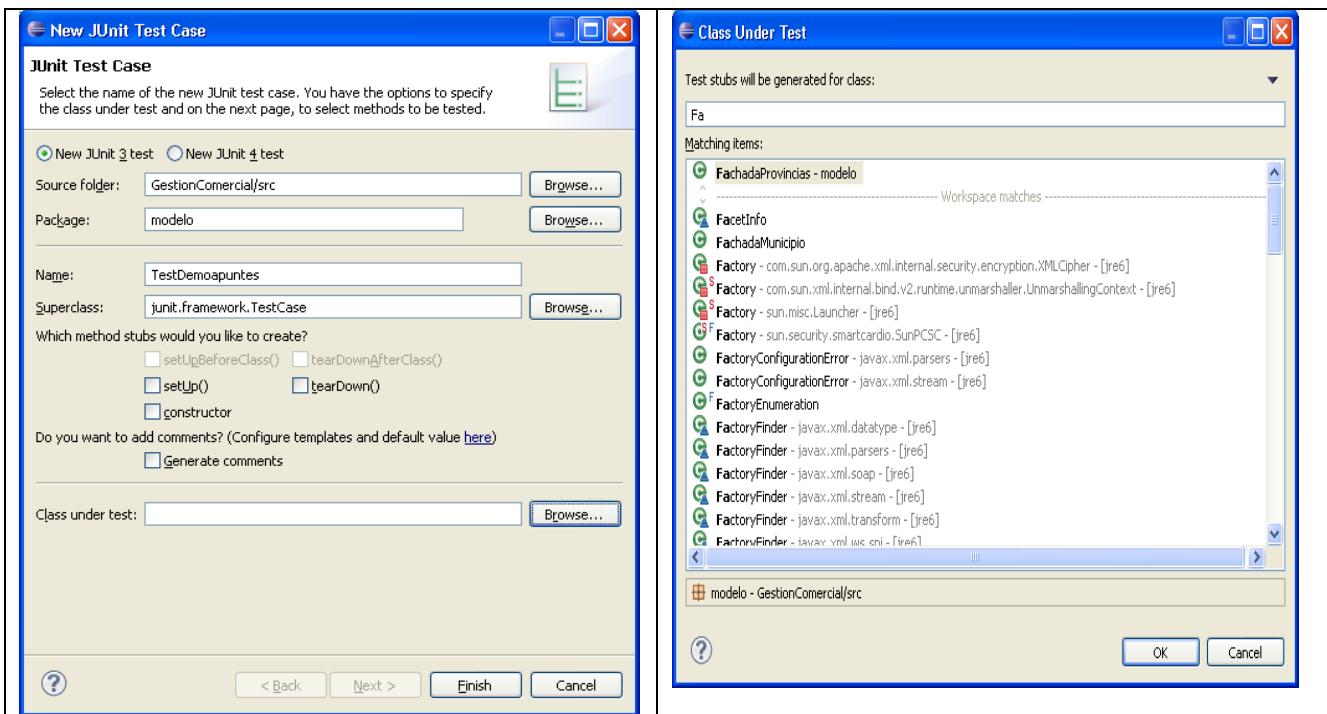
```
AssertEqueals("Error", int valor Esperado, obj.metodo());
```

JUnit proporciona suites para organizar las pruebas por bloques. Esto se hace a través de la clase **TestSuite** y es simplemente una agrupación de pruebas. Eclipse también lo hace de forma casi automática.

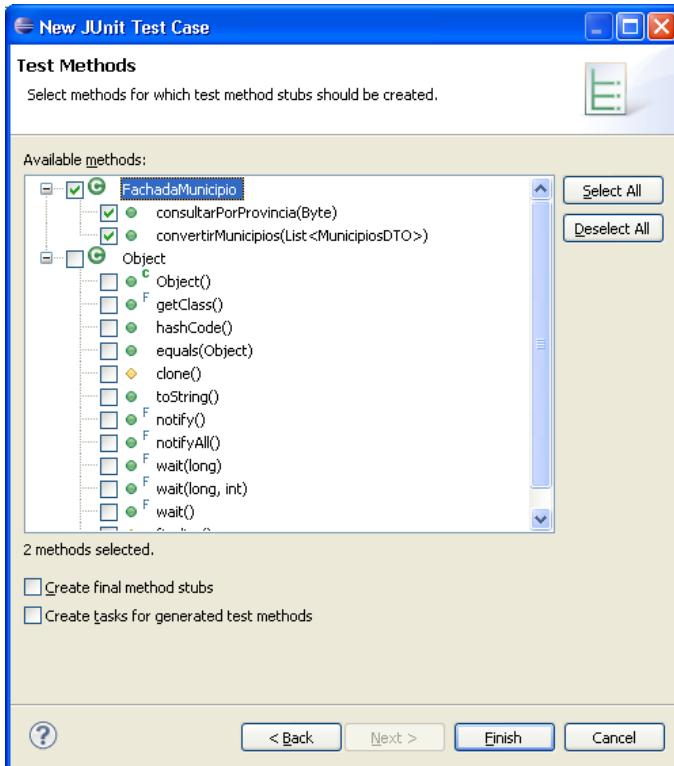
Para implementar una clase para probar un conjunto clases de pruebas en Eclipse se hace añadiendo una clase **JUnit Test Case** en el paquete donde está la clase que se quiere probar. **New – Junit Test Case**



Al hacer eso aparece una ventana de dialogo en la que elegimos “New JUnit 3 Case”, le damos un nombre que debería comenzar por la palabra “test” y además hemos de elegir la clase que vamos a probar en la parte de debajo de esa ventana, en el botón **Browse**, que al pulsarlo se abre otra ventana. Donde al ir escribiendo el nombre de la clase nos aparecen las coincidencias en el panel de abajo.



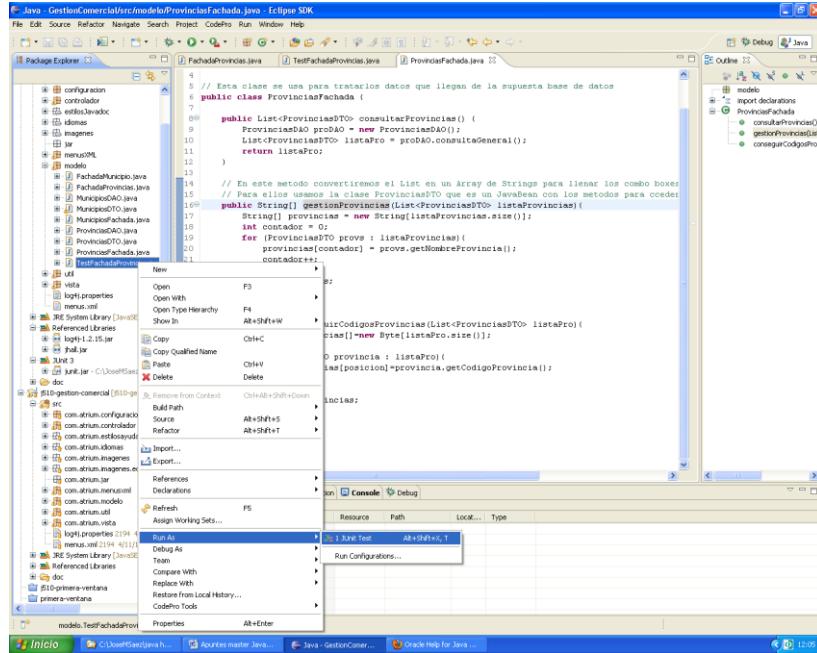
Elegimos la clase y pulsamos ok, y a continuación pulsamos Next > en la ventana anterior, con lo que la ventana ahora mostrará una lista de métodos que podemos probar y seleccionamos los que nos interese probar.



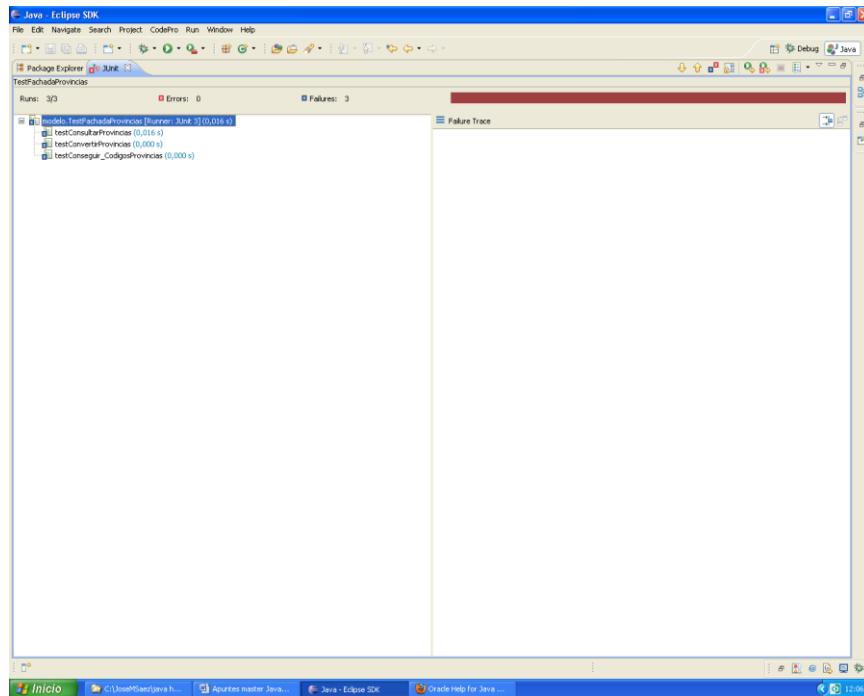
A continuación pulsamos Finish y ya se genera la clase. Sobre la que escribimos el código para las pruebas según la funcionalidad de los métodos y usando los asserts que necesitamos.

Los métodos generados en eclipse en la clase de pruebas llevan un método `fail("Not yet implemented")`; para que no se nos olvide implementarlos. Eso hay que quitarlo cuando implementemos el código para esas pruebas.

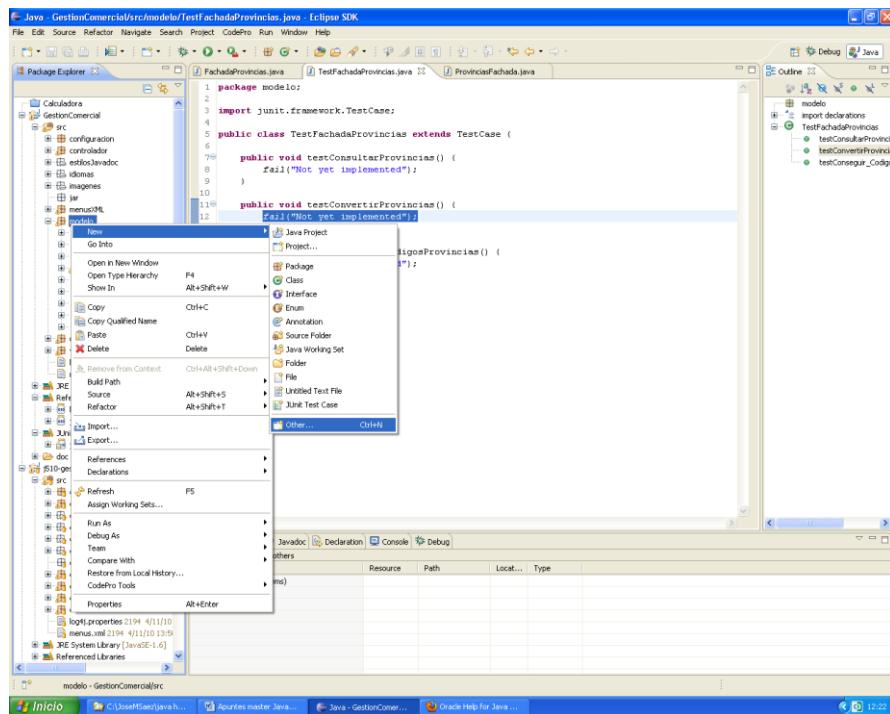
Para correr la prueba se da botón derecho sobre la clase de prueba y se elige *Run as... Junit Test*. O bien el botón de Run.



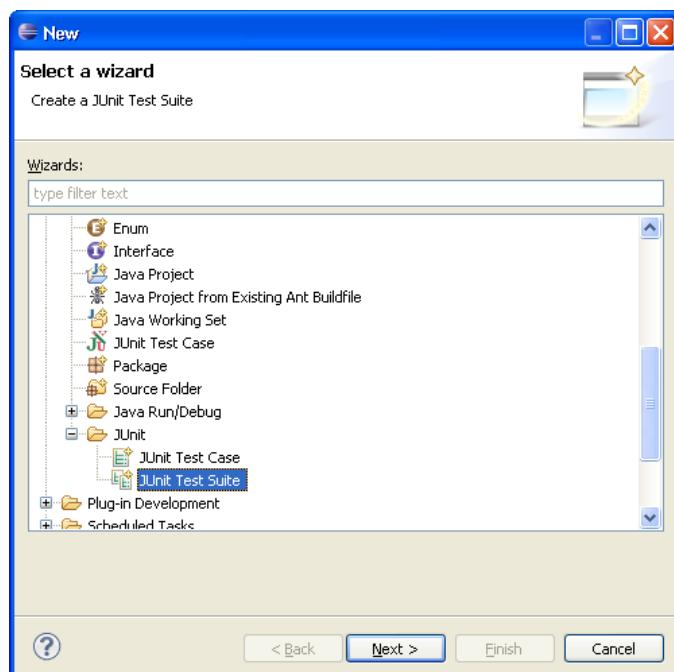
Esto abre una pestaña en el lado izquierdo llamada JUnit, ahí se muestra el resultado de las pruebas. Se usa la ventana JUnit de las pestañas de la parte de la consola en Eclipse.



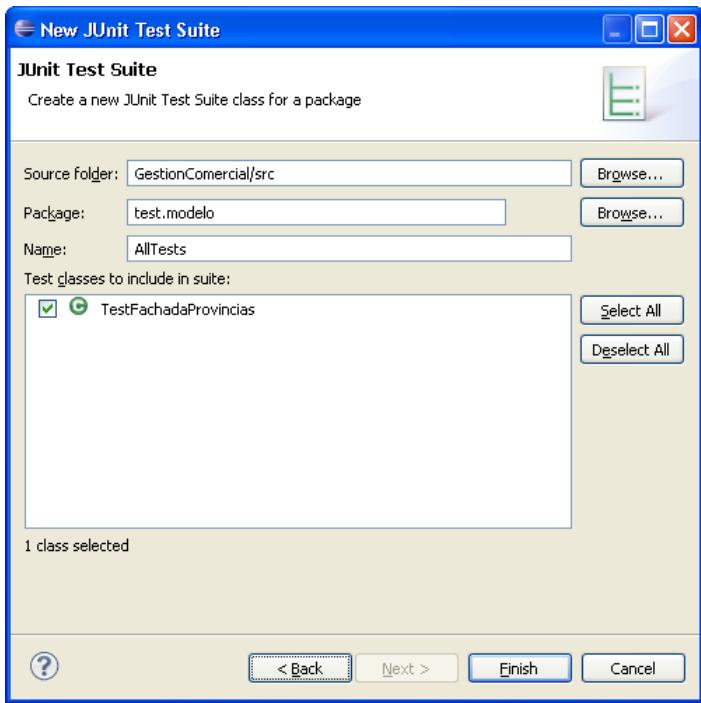
Para que eclipse genere la test suite automáticamente, y simplificar el proceso de pruebas, se hace pulsando con botón derecho en el paquete de clases de prueba para el que queremos generar la test suite, y vamos a New – Other.



Esto abre una nueva ventana, en la que abrimos Java, luego JUnit y elegimos JUnit Test Suite



Pulsamos en next, elegimos las clases que queremos incluir en la suite



Pulsamos Finish y se genera la clase, llamada por defecto AllTests.java, donde se agrupan los métodos de todas las pruebas.

Podemos hacer todo el conjunto de pruebas a la vez si probamos la clase de la suite de la misma forma que probamos otra clase de pruebas unitarias individual normal.

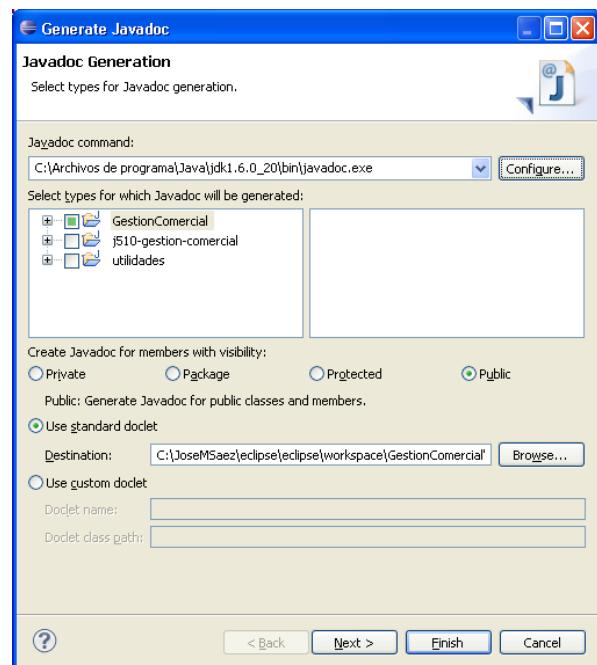
JavaDoc

En Eclipse para generar el Javadoc se va a **Project – Generate Javadoc**. En Javadoc command se escribe la ruta al javadoc en el JDK, por defecto esta en archivos de Programas – Java y apareceran tantos como versiones del jdk haya.

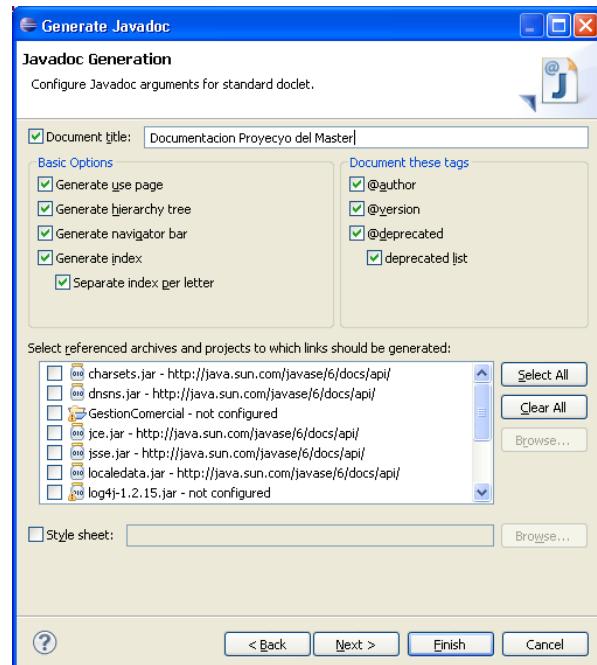
El Destination se rellena solo según lo elegido en Javadoc command y la selección de proyecto hecha.

Pulsar en Next >:

Se puede elegir una hoja de estilo para la creacion de la web de Javadoc.



Se pulsa Finish y se crea una carpeta doc en el proyecto, dentro del cual estará la web que se ha generado.



CodePro

CodePro analytics es un plugin para Eclipse gratuito de Google <http://code.google.com/intl/es/javadevtools/> ó

<http://code.google.com/intl/es/javadevtools/codepro/doc/index.html>, tiene una enorme variedad de herramientas de ayuda para probar o mejorar la calidad del código. Incluso repara el JavaDoc generando los comentarios en las clases o métodos donde no los hay.

Así mismo es una herramienta muy usada para medir distintos parámetros de la aplicación, como el porcentaje de líneas de código frente a líneas de comentarios, número de caracteres, métodos, líneas de código por método, etc, etc. Muy usado por analistas o jefes de proyecto.

Una vez instalado, se integra en los menús contextuales de Eclipse y se añade un menú a la barra de menús. Dá estadísticas y la métrica. Se puede comprobar la calidad del código con **auditCode**.

Para instalarlo en Eclipse hay que descargarlo usando el enlace según la versión que tengamos de Eclipse. El enlace para Helios es <http://dl.google.com/eclipse/inst/codepro/latest/3.6> y para Galileo es <http://dl.google.com/eclipse/inst/codepro/latest/3.5>

En Eclipse se va a Help – Install new software – Add En Location pegamos el enlace copiado de la web de google y OK. Se eligen las dos primeras opciones Codecoverage y CodePro y se pulsa Next y Next, se acepta la licencia y comienza el proceso de descarga e instalación. Es posible que nos de un warning sobre instalación de software sin firmar, lo aceptamos. Al final se reinicia Eclipse.

AYUDA

JavaHelp esta disponible en la comunidad www.java.net

Para las ayudas del programa, Java tiene una ayuda completa donde únicamente hay que añadir el contenido. Se puede acceder a ella a través de cualquier componente del programa, un elemento de menú, un botón etc.

Aunque este sistema pertenece a Sun (Oracle) hay que descargarlo de la web de Sun u Oracle para poderlo instalar en nuestra aplicación de Eclipse.

<https://javahelp.dev.java.net/>

Se configura mediante ficheros XML. Las páginas de ayuda que se muestran son HTML que son recursos externos que se hacen aparte. Eso permite ser usado tanto en un entorno cliente-servidor como en una aplicación web.

La ayuda es un paquete adicional llamado “**Java Help**”.

Los paquetes XML de configuración de la ayuda son varios con distintas extensiones según lo que van a configurar.

extension **.hs**

extension **.jhm**

extension **.xml**

Archivos .hs

La etiqueta raíz es **<helpset version=xx>** donde xx es la versión del Java Help que vamos a utilizar, actualmente la 2.0.

Dentro del helpset definiremos vistas con la etiqueta **<view>**, estas vistas son las pestañas que aparecen en la parte izquierda de la ventana de ayuda. La ordenación más habitual es alfabéticamente o por temática.

Dentro del view van:

El nombre de la vista (**<name>**)

El texto de la etiqueta (**<label>**)

El **<type>** que es la forma como vamos a tratar esa información. Hay varias clases de java para tratarlo y varias alternativas para el type:

- La más habitual es **javax.help**
- **TOC.View** (Table Of Contents) que tiene forma de árbol
- **IndexView** que muestra un índice de los temas en el orden en que nosotros lo pongamos.
- **DefaultSearchEngine**, que es la ventana de búsqueda que encontrara ese texto en el contenido de la ayuda.
- **FavoriteView**, es la pestaña de Favoritos.

Además, en la vista (**<view>**) hay que indicar la información que se va a mostrar, eso se define

aparte y son los ficheros que contienen esa información, y se indica en otro fichero XML. La etiqueta usada es <**data**>

Habrá un fichero por cada vista que creemos y otro donde se almacena la información.

Un ejemplo podría ser este:

```
<helpset version=2.0>
  <view>
    <name>nombreDeLaVista</name>
    <label>textoEnPestaña</label>
    <type>javax.help</type>
    <data>ficheroXML</data>
  </view>
</helpset>
```

Pueden usarse iconos en varios sitios de la ayuda. Estos iconos son recursos externos tienen su propio fichero de mapeado de recursos. Con lo cual se referenciará por un nombre distinto al de su propio nombre. Ese fichero se indica con la etiqueta <**maps**> que lleva dos etiquetas hijas <**homeID**> y <**mapref**> con el atributo location. La facilidad que proporciona el mapeado es, por ejemplo, poder cambiar el fichero del contenido con solo variar ese valor en el fichero de mapeo.

```
<maps>
  <homeID>nombreDelRecuso</homeID>
  <mapref location="nombreFichero.jhm" />
</maps>
```

La ayuda se puede colocar como una ventana aparte o embebida dentro del programa.

Si es una ventana aparte se le debe indicar tamaño, posición, título, barra de botones a esa ventana. Eso se indica también en un archivo XML con la etiqueta <**presentation**> y el atributo default que puede ser true o false. Así:

```
<presentation default="true" displayviews="true"
displayviewimages="true">
  <name>MainWin</name>
  <!-- Dimensiones iniciales -->
  <size width="640" height="480" />
  <!-- Posición inicial -->
  <location x="200" y="200" />
  <!-- Título de la ventana -->
  <title>Título de la ventana de ayuda</title>
  <!-- Definimos la barra de herramientas de la ventana -->
  <toolbar>
    <!-- Permitimos ir a la página anterior -->
    <helpaction image="BackwardIco">
      javax.help.BackAction
    </helpaction>
    <!-- Permitimos ir a la página siguiente -->
    <helpaction image="ForwardIco">
      javax.help.ForwardAction
    </helpaction>
    <!-- Permitimos imprimir el contenido -->
```

```

<helpaction image="PrintIco">
    javax.help.PrintAction
</helpaction>
<!-- Permitimos configurar la impresión --&gt;
&lt;helpaction image="PrintSetupIco"&gt;
    javax.help.PrintSetupAction
&lt;/helpaction&gt;
&lt;/toolbar&gt;
&lt;/presentation&gt;
</pre>

```

La barra de botones **<toolbar>** tiene varias clases según la acción y el tipo de botón:

BackAction → la ayuda anterior – backward.ico

ForwardAction → La ayuda siguiente – forward.ico

PrintAction → imprimir la ayuda – print.ico

PrintSetupAction → llamar a la ventana de configuracion de la impresora para configurarla – printsetup.ico

Archivos .jhm

Jhm = java help mapping

La etiqueta principal es **<map>** con el atributo **version**.

Después lleva la etiqueta **<mapID>**; tantas como ficheros haya y los atributos **target** y **url**.

```

<map version="2.0">
    <mapID target="clientes" url="clientes.html" />
    <mapID target="dp" url="DP.html" />
    <mapID target="de" url="DE.html" />
    <mapID target="db" url="DB.html" />
    <mapID target="ob" url="OB.html" />
</map>

```

Target es el nombre del archivo u url es la pagina web que contiene las explicaciones de la ayuda pertinente.

Archivos con extensión .xml

Las vistas de tipo **TOC.view** (definidas en el archivo descriptor con extensión .hs - arbol) llevan un atributo versión y se construyen así:

```

<toc>
    <tocitem version="xxx" text="Clientes" target="clientes">
        <tocitem text="Datos Personales" target="de"/>
        <tocitem text="Datos Economicos" target="dp"/>
        <tocitem text="Datos Bancarios" target="db"/>
        <tocitem text="Observaciones" target="ob"/>
    </tocitem>
</toc>

```

Pueden llevar opcionalmente iconos o imágenes asociadas. Con lo cual cada etiqueta toc puede llevar tres atributos; **text**, **icon** y **target**. A estos recursos siempre se les llama por su nombre mapeado.

Target es la página de ayuda que va a mostrar es decir el archivo .jhm llamado por su mapeado.

Es recomendable hacer capturas de pantalla e incluirlas en las páginas de ayuda HTML.

Las vistas **IndexView** se construyen de forma parecida a las TOC.view. La etiqueta es <**index**>.

```
<index>
    <indexitem version="xxx" text="Clientes" target="clientes">
        <indexitem text="Datos Personales" target="de"/>
        <indexitem text="Datos Economicos" target="dp"/>
        <indexitem text="Datos Bancarios" target="db"/>
        <indexitem text="Observaciones" target="ob"/>
    </indexitem>
</index>
```

Hasta aquí hemos visto la parte de configuración o definición. Para que se pueda mostrar la ayuda hay dos clases. La clase **HelpSet** y la clase **HelpBroker**.

HelpSet recupera la información de otros ficheros.

HelpBroker muestra la información.

Por lo tanto cuando nos creamos el objeto HelpSet (lanza excepciones activas) pasamos en el constructor la información que se necesita. Esta información es la ruta y el nombre del fichero principal de configuración. Para indicárselo hay que tener en cuenta que la ayuda puede no estar en el mismo ordenador donde se instala la aplicación, sino que se carga desde otro sitio. Esto se hace así para facilitar la actualización de la ayuda sin tener que actualizar los programas cliente. Por lo tanto esto se hace a través de un objeto de tipo URL, tanto si es una ruta externa como interna.

Así mismo habrá que crear un objeto ActionListener y asociarlo (añadirlo) al componente que vaya a lanzar la ayuda.

```
URL ruta = new URL("rutaURL");
Helpset objetoAyuda = new HelpSet(null, ruta); // classLoader, URL ruta
ObjetoHelpBroker ventana = objetoAyuda.createHelpBroker();
ActionListener escuchadorAyuda = new
CSH.DisplayHelpFromSource(ObjetoHelpBroker);
componenteLanzadorDeAyuda.addActionListener(escuchadorAyuda);
```

Si queremos sacar la ayuda con la tecla F1 hay que llamar al método **enableHelpKey(,,)**; que pertenece a **HelpBroker** y al que se le pasan tres argumentos:

1.- **Component** → Es el componente sobre el cual queremos activar, que debería ser el

contentPane.

2.- **String** → El nombre de la pagina web sin extensión que se va a mostrar

3.- **HelpSet** → El Helpset que queramos mostrar

```
enableHelpKey(obj, " ", "paginaWeb", objetoAyuda);  
  
ObjetoHelpBroker.enableHelpKey(this.getContentPane(), "paginaWeb",  
objetoAyuda);
```

Si queremos hacer una ayuda sensible al contexto tendremos que usar hlp.SetCurrentId(), y tendremos que usarlo muchas veces a lo largo del código.

Si queremos que se muestre en la ventana de ayuda en el panel derecho la ayuda correspondiente a lo que está haciendo (ayuda sensible al contexto), se usa el método **setViewID(target)** este método se tiene que incluir en el código de la aplicación. El parámetro target ha de ser el nombre que le hayamos mapeado a ese target.

En Eclipse para la creación de la ayuda nos creamos un paquete para la ayuda exclusivamente y en ese paquete añadimos ficheros (New – File) con extensión .hs. para los archivos XML. Hay un ejemplo en el proyecto j510-gestion-comercial del master.

Con el editor de textos creamos los archivos XML, podemos encontrar el prólogo o cabecera en el archivo comprimido de Java Help. También podríamos encontrar algún IDE o programa de generación de archivos XML que nos pudiera ayudar en esta labor, como el **MyEclipse**.

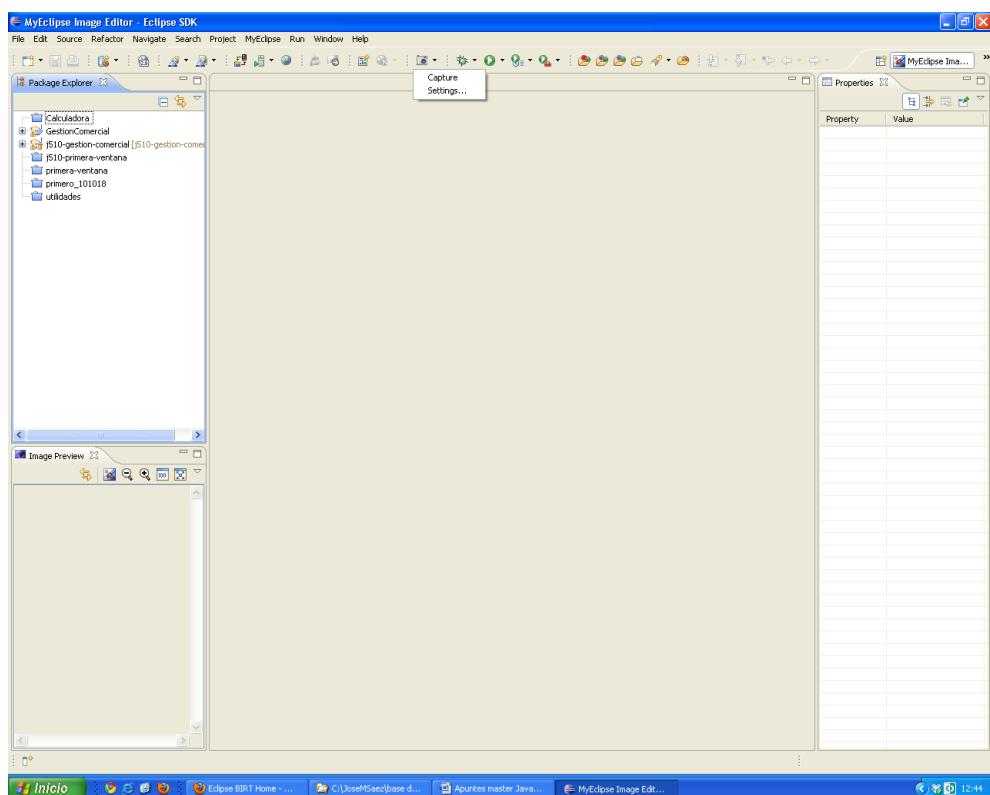
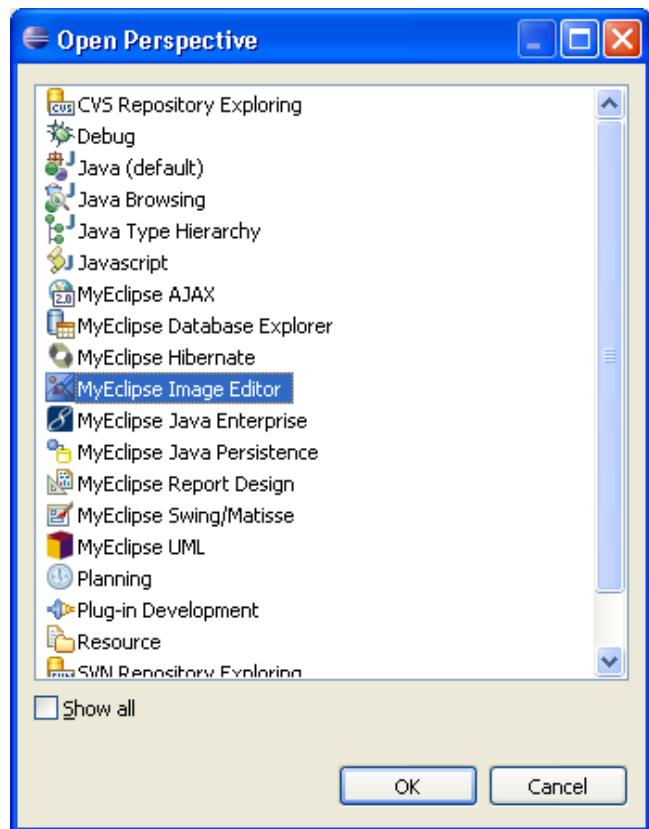
MyEclipse es un conjunto de plugins que cubre casi todas las necesidades que se pueden tener en el entorno de desarrollo. Es un producto comercial que no es gratis.

El **MyEclipse** debe ser de una versión compatible con nuestro eclipse. Al momento de escribir estos apuntes la última versión de MyEclipse (8.6.1) es compatible con la versión de Eclipse 3.5.2 (Galileo), así que, como veníamos usando la versión 3.6.0(Helio), tenemos que instalarnos otra versión de Eclipse más antigua. Debemos usar el mismo workspace que usamos con la versión Helio de Eclipse. Además tendremos que cargar el entorno gráfico de google, el SVN, etc.

Una vez tenemos el eclipse compatible con MyEclipse lo instalamos como un plugin desde donde lo tengamos alojado.

Para crear las páginas web de la ayuda MyEclipse tiene unos plugins muy útiles por ejemplo para capturar pantallas como imágenes que pueden ser incluidas en esas páginas web.

Para usarlo hay que pulsar el botón de perspectivas, después en Other y se abre una ventana con las perspectivas que ofrece MyEclipse, la que se llama “MyEclipse Image Editor” hace las capturas de pantalla, usando un ícono de una cámara de fotos que aparecerá en la barra de herramientas. Mirad las dos imágenes a continuación.



Así mismo MyEclipse tiene editores HTML para hacer las páginas web, podemos usar estos editores o bien usar los que creamos convenientes.

INFORMES (BIRT)

La realización de informes en Java puede ser desarrollada por nosotros mismos o usando alguna solución ya creada. Existe una solución llamada **BIRT** (Business Intelligence Report Tools).

Los programas profesionales incluyen en gran medida la elaboración de informes y su salida por impresora.

BIRT es un paquete ya integrado en Eclipse y permite una enorme variedad de opciones y posibilidades de impresión, tanto para J2SE como para J2EE. Es gratuito y libre, propiedad de IBM.

BIRT tiene dos fases una es el **diseño** del propio informe y otra es la fase de **integración**

El diseño ha de ser elaborado con antelación e incluye la forma de adquisición de los datos que serán mostrados y el formato. Se genera en un **archivo con extensión .rptdesign** y es un archivo XML.

El diseño visual de este informe puede ser un plugin instalado en nuestro eclipse u otro eclipse aparte. O bien usar el que viene integrado en **MyEclipse**.

La fase de integración es más compleja por que el informe nace de la lectura del fichero rptdesign. Con lo cual hemos de tener todas las clases Java en nuestro proyecto que se encargan de generar el informe.

El BIRT puede ser un proyecto elaborado de forma independiente que puede ser accedido por distintas aplicaciones diferentes.

El Plugin (proyecto) BIRT puede descargarse de la web de eclipse del link de proyectos (<http://download.eclipse.org/birt/downloads/>). En el sitio encontraremos no solo el archivo de descarga sino todo tipo de documentación.

Cuadro Nuevo perfil de fuente de datos JDBC

Clase controlador: oracle.jdbc.driver.OracleDriver(v10.12)

url del controlador:jdbc:oracle:thin:@a3-1m:1521:XE

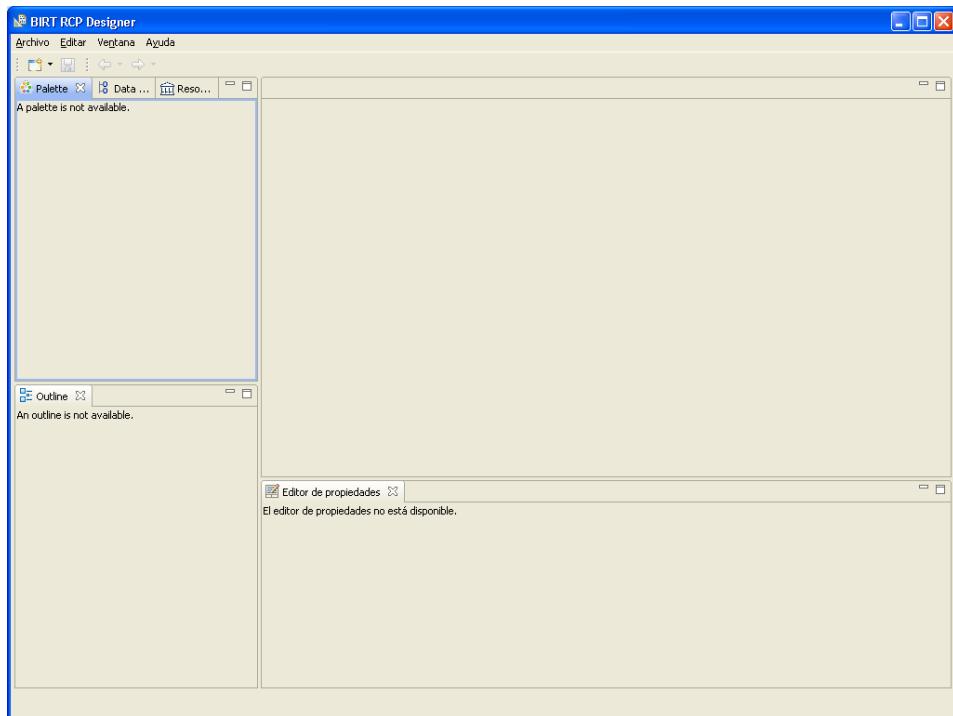
user:master

pass: master

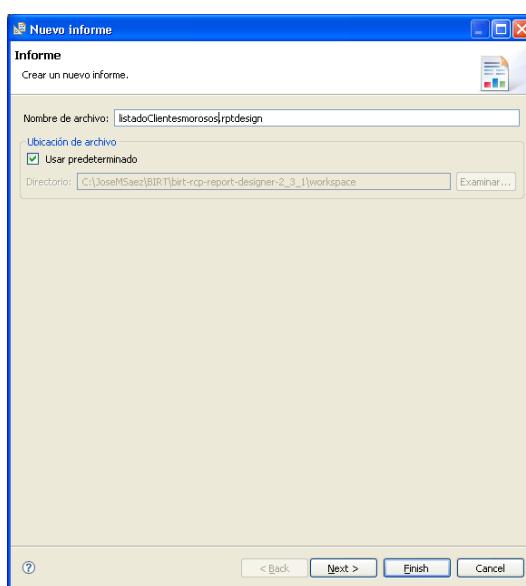
tras elegir la fuente de datos hay que elegir que conjunto de datos queremos

Diseño del Informe

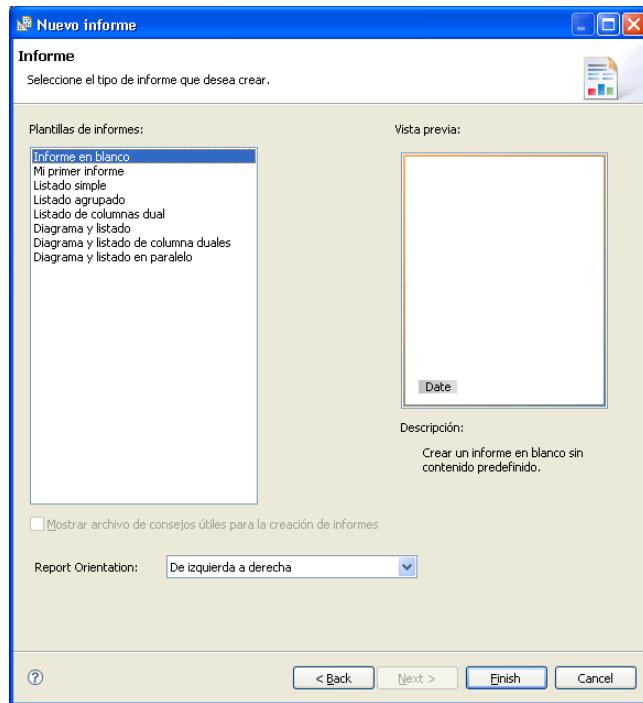
Para desarrollar el diseño en J2SE hay que hacerlo instalando en eclipse el birt correspondiente, nosotros lo hacemos descomprimiendo el archivo **birt-rcp-report-designer-2_3_1.zip** en algún sitio que conozcamos y encontraremos un archivo llamado **BIRT.exe** que es un eclipse con el **BIRT**, tenemos una ayuda para traducirlo al español llamado **NLpack1-birt-rcp-report-designer-2_3_1.zip** que la extraemos en la carpeta plugins dentro de la carpeta donde hayamos descomprimido el BIRT.



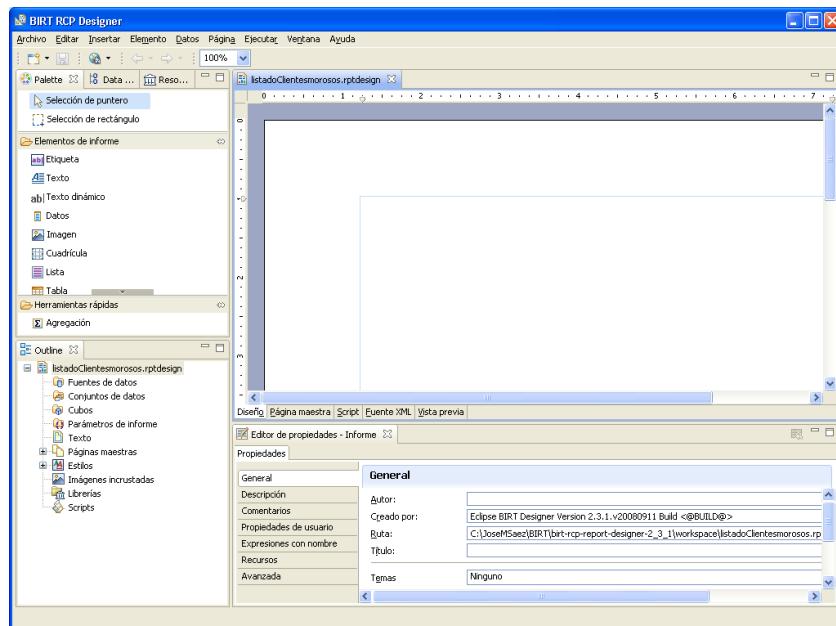
En este BIRT creamos el diseño, desde Archivo – Nuevo – Nuevo Informe, le damos un nombre



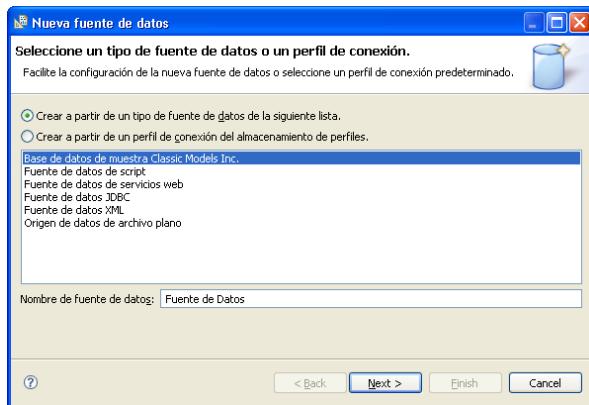
Pulsamos Next y nos da opciones de tipos de informes nosotros elegimos informe en blanco



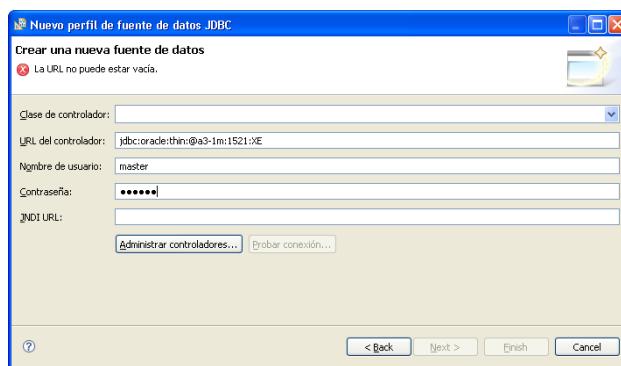
Y pulsamos Finish



Primero elegimos la fuente de datos, con botón derecho sobre Fuente de Datos, Nueva fuente de Datos



Aquí elegimos la fuente de datos que vamos a necesitar, si no tenemos ninguna base de datos preparada en este momento elegimos la que trae BIRT de prueba. En el aula del curso usaremos JDBC (base de datos). Le damos un nombre y pulsamos Next.



Rellenamos los datos de la conexión que serán siempre los mismos, para nuestro ejemplo esos datos son:

Clase de controlador:

URL del controlador: jdbc:oracle:thin:@a3-1m:1521:XE

Nombre de Usuario: master

Contraseña: master

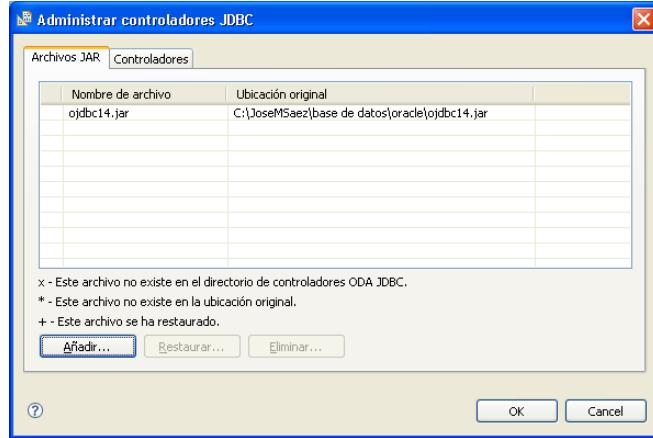
JNDI URL:

Para MySQL usaríamos esta URL: jdbc:mysql://localhost:3306/nombreDB

El formato para MySQL es este:

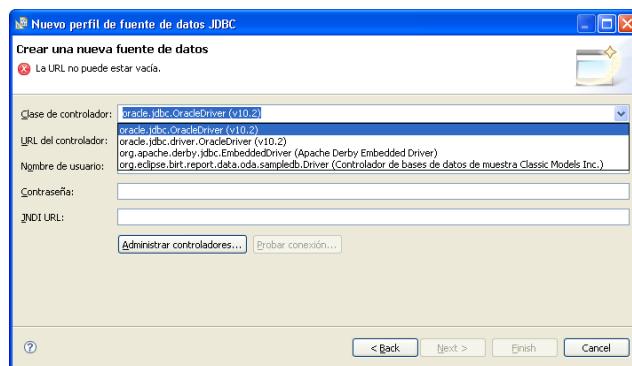
```
jdbc:mysql://host:puerto/database
jdbc:mysql://host:puerto/database?user=usuario&password=clave
```

Pulsamos en Administrar controladores...

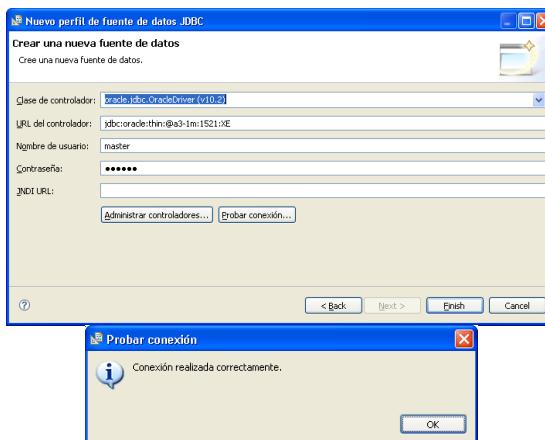


Le damos a Añadir... y buscamos el driver en nuestro ordenador. Pulsamos la pestaña controladores y elegimos oracle.jdbc.OracleDriver(10.2) y pulsamos OK.

De la pestaña de Clase de Controlador elegimos ese driver.



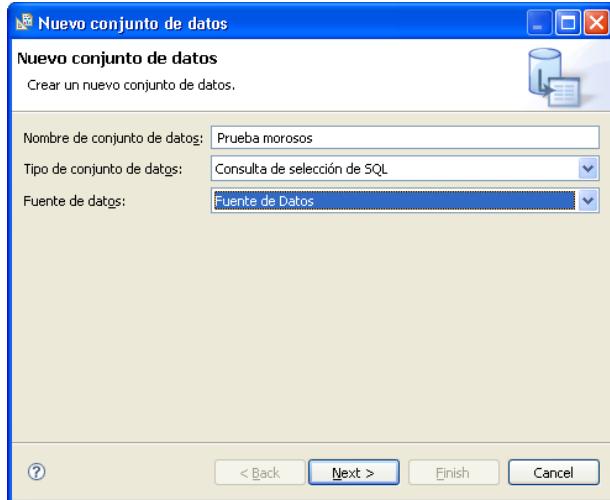
Y rellenamos los demás datos como se ha explicado arriba, entonces podemos pulsar en probar conexión



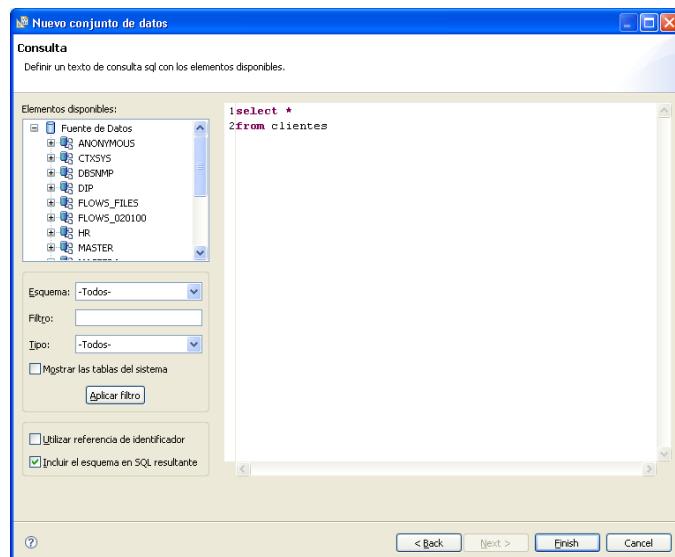
Pulsamos Finish.

Con esto hemos definido la fuente de datos. Pero no, de donde vamos a coger los datos. Para ello, entramos en Conjunto de datos (botón derecho – nuevo conjunto de datos). Y rellenamos el

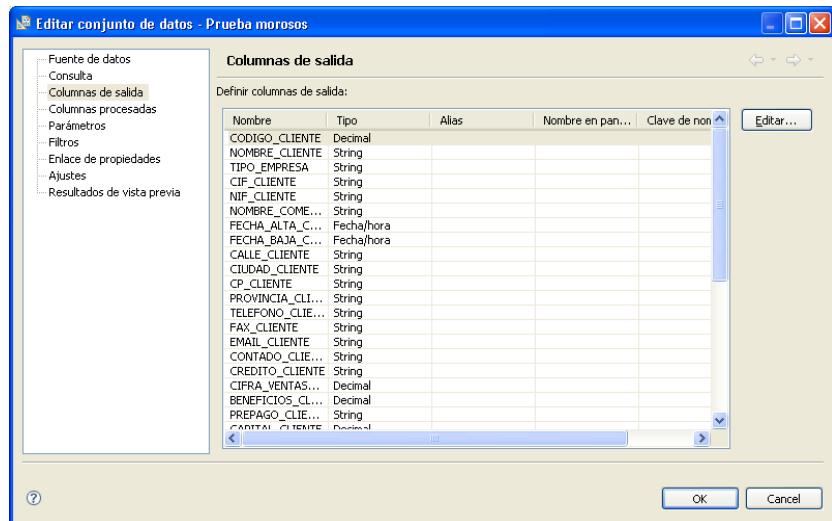
formulario



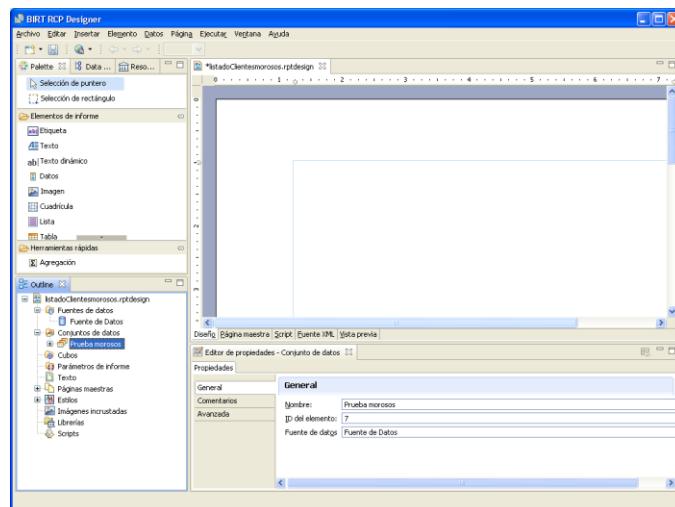
Pulsamos next y aparece una ventana donde se escriben las consultas SQL.



Se escribe la consulta y se pulsa finish. Entonces aparece la lista de columnas de donde elegimos lo que vamos a usar.



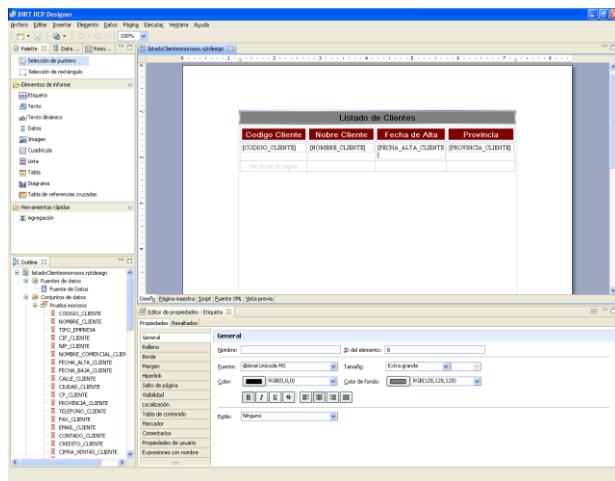
Para nuestro ejercicio pulsamos OK. Y ya podemos comenzar a diseñar el informe:



Arrastrando Elementos de Informe a la hoja en blanco. Y adecuándolo a nuestras necesidades. Desde la pestaña Editor de propiedades (doble click para agrandar) podemos cambiar tipos de letra, color de fondo, etc.

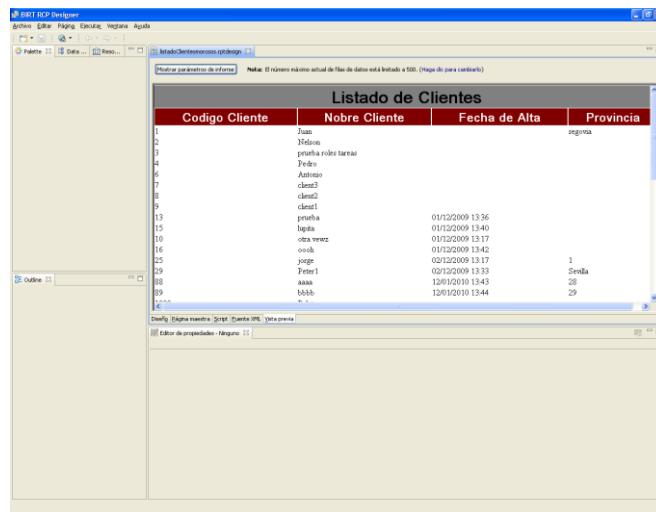
Es un editor muy completo y se puede hacer un diseño muy adecuado a nuestras necesidades, pueden incluirse tablas y los datos se añaden desde la fuente de datos elegida.

Como hemos elegido una base de datos podemos añadir una tabla a nuestras necesidades para el informe y podemos ir añadiendo los campos a la tabla desde la zona derecha donde están los campos de la tabla de la base de datos.



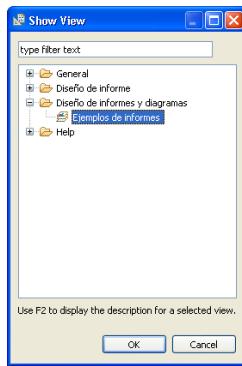
El editor es muy adaptable y se trata de familiarizarnos con su uso y sus opciones, existen manuales y toda la información necesaria en la web del fabricante y en la de Eclipse.

La pestaña vista previa nos muestra los cambios que vamos haciendo a nuestro informe

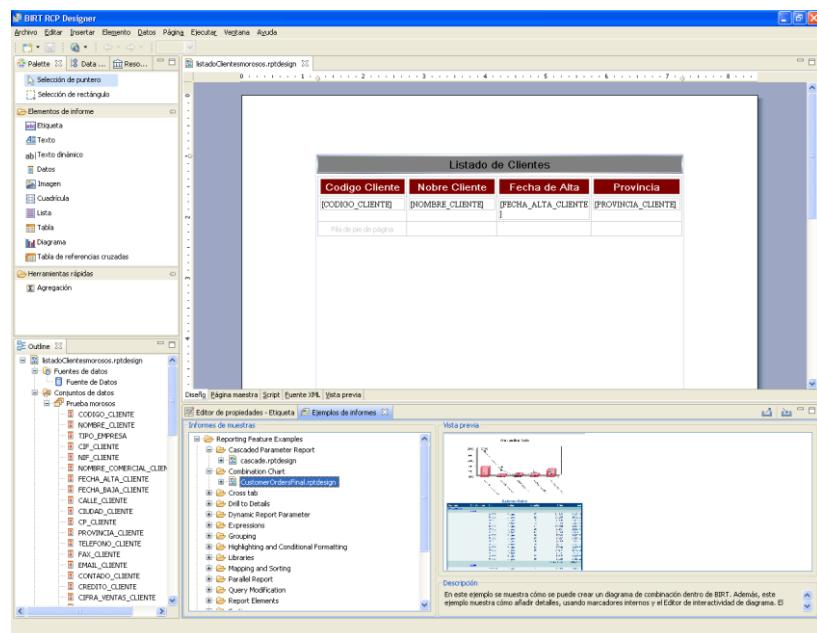


Una vez hayamos ajustado el informe como mas nos interesa podemos guardar el informe y cerrar la aplicación.

Tenemos una serie de ejemplos en Ventana – Mostrar Vista – Other – Diseño de Informes y diagramas – Ejemplos de informes.



Esto nos añade una pestaña al BIRT en la parte de abajo junto al Editor de Propiedades donde podemos ver los ejemplos de informes.



Los datos que se muestran en el informe pueden ser calculados en tiempo de ejecución. La cantidad de opciones disponible para el diseño de informes es interminable y es aconsejable dedicarle tiempo a familiarizarse con esta aplicación.

Según vamos haciendo el diseño se van escribiendo los archivos XML con extensión rptdesign y los encontraremos en el workspace del BIRT que lo genera automáticamente dentro de la carpeta donde lo hemos descomprimido.

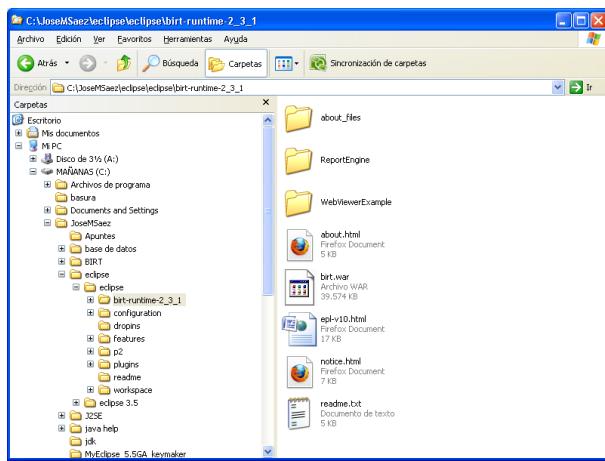
Para integrarlo en nuestro proyecto creamos un paquete en nuestro proyecto y lo copiamos en ese paquete el archivo rptdesign que encontraremos en el workspace de BIRT.

Tenemos disponible el proyecto j510-gestion-comercial hecho durante el master para ver clases y archivos de muestra.

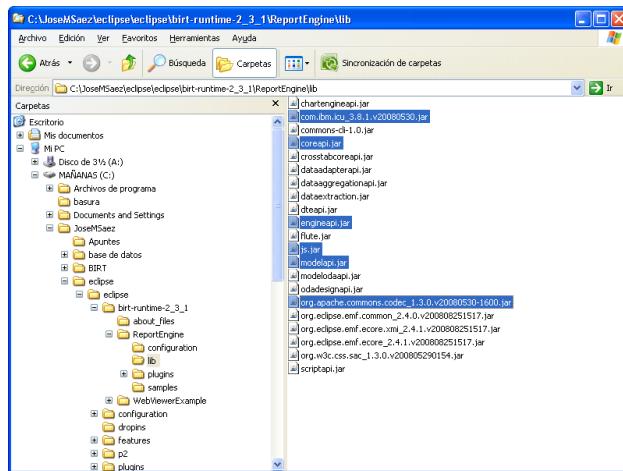
Integración

Consiste en leer la información contenida en el fichero rptdesign e integrarla con las clases. Para que esto funcione necesitamos tener lo que se llama el "motor", o conjunto de clases que sirven para funcionar con él. El motor hay que instalarlo.

El motor es un fichero que se llama birt-runtime-2_3_1.zip en nuestro caso. Lo descomprimimos en una carpeta, por ejemplo donde tengamos nuestro Eclipse.



Nos centramos en la carpeta llamada ReportEngine – lib. Dentro de esa carpeta lib hay una serie de ficheros jar que son los que podemos necesitar en nuestro proyecto según lo que vayamos a necesitar en nuestro informe.



Para saber lo que hace cada uno de estos archivos jar hay que consultar la documentación. Los seis señalados en la imagen de arriba son los que necesitamos en nuestro proyecto. Pero podemos incluir todos y así evitaremos problemas.

Los ficheros jar que necesitemos los añadiremos a nuestro proyecto. Con lo cual los copiamos y pegamos en algún paquete y después los añadimos como hicimos con el log4J, etc. Como una librería.

Una vez añadidas estas referencias podemos crear nuestra clase de integración para crear el informe en los formatos que deseemos, ya sea HTML, PDF, etc, a partir de nuestro proyecto.

En la clase necesitamos cuatro propiedades que son String y son el directorio del motor, el directorio del log de BIRT, el informe de entrada y el informe de salida.

El **directorío del motor** debe incluir la ruta hasta el directorio ReportEngine donde tengamos los archivos jar de los que hemos hablado antes, por ejemplo *C:\JoseMSaez\eclipse\eclipse\birt-runtime-2_3_1\ReportEngine*.

El **directorío del log** lo creamos nosotros en el mismo directorio donde este el directorio ReportEnginne, por ejemplo y se lo pasamos como ruta, por ejemplo esto *C:\JoseMSaez\eclipse\eclipse\birt-runtime-2_3_1\log*

El **informe de salida** es el directorio donde vamos a poner el informe una vez generado y su nombre, por lo tanto es un directorio que debe existir.

El **informe de entrada** es la localización en nuestro proyecto del archivo rptdesign.

Si trabajamos con bases de datos en nuestro informe, el driver de la conexión debe ponerse en un sitio concreto que es el directorio drivers dentro del directorio plugins y la carpeta del driver que estamos usando, por ejemplo esto:

C:\JoseMSaez\eclipse\eclipse\birt-runtime-2_3_1\ReportEngine\plugins\org.eclipse.birt.report.data.oda.jdbc_2.3.1.v20080827\drivers

Con lo cual hay que ponerlo ahí antes de nada. El que usamos en nuestro ejemplo esta en *C:\JoseMSaez\base de datos\oracle* y se llama **ojdbc14.jar**

En nuestro proyecto de Gestión Comercial tenemos una clase que podemos consultar en cualquier momento para ver los detalles de la construcción de esta clase. Se llama **Ejecutar_birt**.

La clase puede incluir un método **main** para poder, si queremos, ejecutarlo por separado para probarlo.

Debe llevar un método por cada formato de salida, HTML, PDF, etc.

Cuando se ejecute esta clase aparecerá un archivo en el formato elegido dentro del directorio de salida que le hemos indicado.

Reflexión

El API de Reflection es un API de bajo nivel mediante el cual podremos trabajar de forma abstracta con los objetos, con lo cual no necesitamos conocer el objeto o el código.

De esta forma podemos gestionar los objetos de forma totalmente independiente ahorrando muchas líneas de código. Por ejemplo, cuando modificamos objetos en un entorno gráfico con listeners, el color, la fuente, etc.

Normalmente usamos métodos accesores en JavaBeans, pero se puede resolver con reflexión, evitándonos una enorme cantidad de líneas de código en nuestros programas.

La API de Reflection siempre trabaja sobre los **objetos** de tipo Class (no archivos .class), a través del **método getClass()** o de la **propiedad class**.

La clase Class pertenece a Object, con lo cual todos los objetos tienen Class. Para trabajar con ellos puede hacerse directamente con cualquier clase o declarándolos. Por ejemplo:

```
Ventana.getClass();  
O bien:  
Class Obj.C = Ventana.getClass();  
Obj.C.getMethod();
```

Normalmente trabajaremos con los métodos:

`getClass().getMethods();` → Nos devuelve un array de objetos **Method**, que representan los métodos que posee el objeto. Con este método podemos conocer todas las características de un método, si es público o no, qué devuelve, si tiene anotaciones, etc.

`getClass().getMethod(nombreMetodo, objetosClass[]);` → Este sirve para ejecutar métodos de ese objeto. Tiene dos parámetros. El primero es el nombre del método, y el segundo es un array de objetos Class que representarían los argumentos que podría recibir ese método. Si no recibiera ningún método sería un array vacío.

Existe un truco para saber cómo se llama el método que queremos ejecutar, el cual es la base de la tecnología de reflexión. Se trata de usar la norma de métodos JavaBean, donde los accesores usan get o set más el nombre de la propiedad con la inicial en mayúscula. Las propiedades pueden ser conocidas o pueden obtenerse a partir de `getProperties()`. Con lo cual **el uso de JavaBeans es fundamental en Reflexión** para conocer los nombres. Todos los componentes de Swing son JavaBean.

Un ejemplo de uso de reflexión sería que, partiendo de la base de que tenemos una clase JavaBean con un parámetro privado de tipo Boolean a enabled, crearíamos una clase para la reflexión, con un método static que acepta tres parámetros:

```

public class usoPropiedadesConReflexion {

    Public static void establecerValor(String
        "nombrePropiedadAModificar",
        Object valorPropiedad,
        ArrayList<JComponent> listaComponentesALosQueAplicarCambios) {

        for(JComponent comp : listaComponentesALosQueAplicarCambios) {
            Class objCompo = comp.getClass();
            Class[] parametros = new Class[1]; // los setters tienen 1
            parametro                                // para controlar el tipo
            if (valorPropiedad instanceof boolean) {
                parametros[0] = boolean.class;
            }
            // Creariamos tantos if como posibles tipos se pudieran dar

            // con esto obtenemos el nombre completo del metodo setter
            Method metodo      =      objCompo.getMethod("set" + +
            nombrePropiedadAModificar.substring(0,1).toUpperCase() + +
            nombrePropiedadAModificar.substring(1),
            listaComponentesALosQueAplicarCambios);

            metodo.invoke(objCompo, valorPropiedad);
        }
    }
}

```

Para poder generar el nombre del método accesor (setter en este caso) usamos:

```

objCompo.getMethod("set"
nombrePropiedadAModificar.substring(0,1).toUpperCase()
nombrePropiedadAModificar.substring(1)

```

Donde primero va el literal “set” o “get” seguido de la primera letra del método en mayúscula más el resto del nombre en minúscula.

El método invoke es el que ejecuta el método en el objeto Class que se le indica. Recibe dos parámetros. El primero es el objeto sobre el que se va a ejecutar el método y el segundo es el valor de la propiedad.

Para cargar los escuchadores a los componentes, lo que sabemos es que los nombres de los métodos siempre acaba en **listener** y podemos usar una clase así:

```

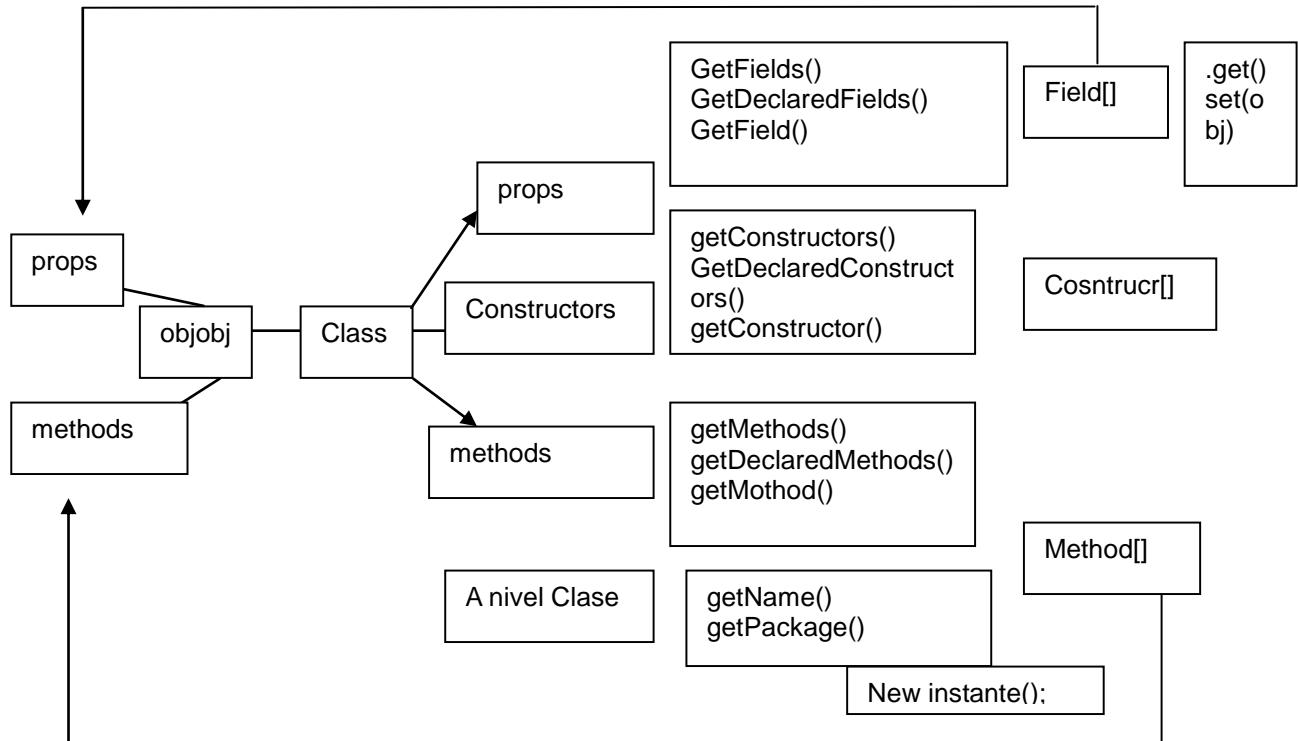
public static void cargar_Escuchadores(List<JComponent> lista_componentes,
                                         Object escuchador) {
    for (JComponent componente : lista_componentes) {
        Class objeto_atratar = componente.getClass();
        if (escuchador instanceof ActionListener) {
            Method metodos_clase[] = objeto_atratar.getMethods();
            int numero_elementos = metodos_clase.length;
            for (int i = 0; i < numero_elementos; i++) {
                if (metodos_clase[i].getName().equalsIgnoreCase(
                    "addactionlistener")) {
                    Object parametro[] = { escuchador };
                    try {
                        metodos_clase[i].invoke(componente,
parametro);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
        if (escuchador instanceof FocusListener) {
            Method metodos_clase[] = objeto_atratar.getMethods();
            int numero_elementos = metodos_clase.length;
            for (int i = 0; i < numero_elementos; i++) {
                if (metodos_clase[i].getName().equalsIgnoreCase(
                    "addfocuslistener")) {
                    Object parametro[] = { escuchador };
                    try {
                        metodos_clase[i].invoke(componente,
parametro);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
        if (escuchador instanceof MouseListener) {
            Method metodos_clase[] = objeto_atratar.getMethods();
            int numero_elementos = metodos_clase.length;
            for (int i = 0; i < numero_elementos; i++) {
                if (metodos_clase[i].getName().equalsIgnoreCase(
                    "addmouselistener")) {
                    Object parametro[] = { escuchador };
                    try {
                        metodos_clase[i].invoke(componente,
parametro);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}

```

En el proyecto del máster Gestión Comercial tenemos las clases Uso_Propiedades y Cargar_Valores donde tenemos estas clases y otras ya construidas y podemos consultarlas y usarlas en nuestros proyectos.

La reflexión puede usarse también para leer propiedades de una clase a través de sus métodos get y pasarlo a otra a través de sus métodos set. La clase Cargar_Valores de nuestro proyecto se encarga de eso para cargar los DTO. Para que esto sea posible por reflexión **las propiedades de ambas clases deben llamarse igual y ser del mismo tipo**, además por supuesto ser ambas javabeans.

La reflexión también permite la instanciación dinámica de objetos usando el método **newInstance()**.



BASES DE DATOS

Las bases de datos en Java siempre se basan en JDBC usando SQL para las queries y adaptándonos al estándar de la base de datos que vayamos a usar. Por otro lado existe una tecnología llamada **ORM** (Object Relational Mapping) que permite trabajar con las bases de datos como si fueran objetos debido a que mapean las tablas. Hibernate es un marco de trabajo para trabajar con el modelo del tipo ORM, probablemente sea el mejor de los ORMs.

JDBC

JDBC es un sistema de conexión a bases de datos a alto nivel. No hace distinciones dependiendo de la base datos que utilicemos. Es un sistema abstracto, independiente de la base datos que haya por detrás.

Esta abstracción la consigue gracias a que sus clases son interfaces. El conjunto de clases que maneja una base de datos concreta es lo que llamamos el **driver**.

Es necesario conocer el driver para poder manejarlo bien, es una documentación no demasiado extensa pero es necesario estudiarla bien.

También es necesario saber qué modelo de conexión vamos a usar. Es posible conectarse a las bases de datos de dos formas:

Modelo Conectado; es más fácil de programar, pero no suele usarse sobretodo en J2EE porque mantiene una conexión abierta y constante durante todo el tiempo. Además de que las conexiones que soporta una base de datos son las licencias que se tienen de esa base de datos y esas licencias son muy caras sobretodo con Oracle. Este modelo no se utiliza prácticamente nunca.

Modelo Desconectado; este modelo lo que hace es conectarse, hacer la operación y desconectarse. Con lo cual el ahorro en recursos y licencias es sustancial.

Se suele utilizar un elemento intermedio que es un pool de conexiones, llamado en Java **DataSource**, que es un servicio y gestiona las conexiones con la base de datos manteniendo siempre algunas abiertas. Suele utilizarse este porque el reparto de conexiones es mas eficiente.

La secuencia de pasos para trabajar con bases de datos es:

1. **Cargar el driver.** El cual es una dependencia externa en nuestro proyecto. Se carga usando la clase Class, usando el método static **forName()** que lleva como argumento un String con el nombre de la clase del driver, *full qualificate* (ruta.paquete), esta información está en la documentación del driver. Lanza excepciones activas ClassNotFoundException. Un ejemplo seria `Class.forName(Oracle.JDBC.Driver.OracleDriver);` Oracle de forma gratuita ofrece el driver thin.
2. **Establecer la conexión con la base de datos.** Para eso creamos un objeto de la clase

Connection (es una interfaz), este objeto ya lleva implementado el suficiente código java para poder establecer conexiones remotas, independientemente de donde este el servidor. Para crear un objeto Connection hacemos así:

```
Connection conex = DriverManager.createConnection(parámetros);
```

Hay que saber los parámetros que recibirá el método **createConnection** ya que eso depende del driver que vayamos a utilizar. Lanza excepciones activas.

Para el caso del proyecto del máster se usan tres parámetros de tipo String, que son:

La URL. Es la cadena de conexión. Esa cadena de conexión debe tener el protocolo de conexión en minúscula (jdbc), dos puntos (:), el subprotocolo, que es la identificación de la base de datos (oracle), dos puntos (:), el tipo, que en oracle pueden ser dos THIN (conexión con la base de datos oracle) o OCI (conexión con un cliente de oracle). La maquina donde está la base de datos (host), que en oracle siempre comienza con arroba (@), dos puntos (:), el puerto, dos puntos (:), el XID, que es el tipo de base de datos. En el proyecto del máster será

`jdbc:oracle:thin:@a3-1m:1521:XE`

jdbc: protocolo

Oracle: subprotocolo

Thin: tipo de driver

@a3-1m: host bbdd. @indica enlace externo

1521: Puerto de comunicaciones

XE: SID. Tipo de BD oracle

El nombre de Usuario. Esta registrado en la base de datos con una serie de permisos, dados anteriormente por el DBA.

La contraseña. Registrada también en la BBDD

Las cosas que podamos hacer con la base de datos están limitadas por los permisos que tenga nuestro usuario.

Nuestro usuario en clase será “master” y la clave también “master”. Nuestro usuario en clase tiene permisos de DBA.

3.- **Lanzar la orden a la base de datos.** Será una sentencia SQL. Para eso hay que crear un objeto que puede ser de clase **Statement**, **PreparedStatement** o **CallableStatement**.

Statement permite lanzar sentencias SQL sin variables.

PreparedStatement permite lanzar sentencias SQL con variables. Por ejemplo para consultar los municipios de una comunidad se crea 1 orden para todos los municipios, con una variable para que sea útil a la hora de buscar cualquiera.

CallableStatement permite pedir lanzar “procesos almacenados” de la base de datos.

STATEMENT

Los **statement** se crean a partir de objetos Connection:

```
Statement sta = conexion.prepareStatement("query sql");
```

Para que el programa sea más eficiente, es necesario externalizar los strings de los argumentos.

Una vez creado se lanza la orden, para ello hay que distinguir entre si la orden es una consulta o cualquier otra cosa. Si es una consulta se usa **executeQuery("sql")**, si no es una consulta (drop, insert, etc) se usa **execute("sql")** que devuelve el numero de registros afectados, ambos pueden recibir como argumento una cadena que es la query SQL. Algo así:

```
sta.executeQuery("SELECT * FROM clientes");
```

En el caso de que no sea una consulta (execute) devuelve el número de filas afectadas. Cuando es una consulta (executeQuery) devuelve un Objeto **ResultSet**.

PREPARE STATEMENT

En caso de **PreparedStatement** hay que enviar las variables, por ejemplo cuando se usa un INSERT. Devuelve también las filas afectadas o un ResultSet si es una consulta.

Las variables siempre se representan con una interrogación, por ejemplo:

```
String query = "SELECT * FROM clientes WHERE cod = ?";  
PreparedStatement pta = conexion.prepareStatement(query);  
Pta.setTipo(posición, valor); // Es necesario saber los tipos que usa la  
base de datos para hacer bien esta conversión  
Resultset.executeQuery();
```

Para poder sustituir los valores de las variables en la query SQL representados por interrogaciones, se usan distintos tipos de sets dependiendo del tipo de variable, por eso hay que conocer que tipos de variable que usa la base de datos con la que vamos a trabajar. La posición la da el primer parámetro del set. En el caso anterior **setTipo(1,valor)** usa la posición 1 para sustituir la primera interrogación por el valor indicado como segundo parámetro que tiene que ser un objeto. Hay que conocer las equivalencias entre la BBDD y los tipos de Java. Suelen encontrarse en el driver estas correspondencias.

4.- Cerrar la conexión y recoger el resultSet. El **ResultSet** que se recibe será destruido cuando se cierra la conexión. Un ResultSet es un simple cursor que contiene las filas afectadas por la query sql. Es como un array bidimensional y podemos movernos por las filas que contiene y sus campos. Cada ResultSet sera una colección de DTO . Tiene un principio de fichero BOF y un final EOF. Cuando se recibe el resultset, no hay ningún registro activo ya que el puntero apunta a BOF. Hay que moverlo a algún registro para que contenga algo. Por lo tanto los resultset tienen métodos de movimiento para movernos por el resultset. Estos métodos de movimiento son:

bool← next(): Hace **dos cosas**, **move el puntero** al registro siguiente y si esta al final ya no continua. Además **devuelve un valor booleano** indicando si hay siguiente registro o no. Si el resultset estuviera vacío también funcionaria ya que el resultset tendría BOF y EOF. Se puede aplicar a todos los tipos de resultset (sí, hay varios tipos de resultset).

```
While(resultSet.next()) {}
```

absolute(int): Situa el cursor en la fila indicada en el parámetro, si esa fila esta antes de la posición actual no puede usarse con el resultset por defecto, ya que ese resultset solo permite ir hacia

adelante.

relative(int): Mueve el puntero tantas filas como le indiquemos en su parámetro, si ese valor fuera negativo iría hacia atrás. Si pides un valor mas alla del ultimo se quedará en el ultimo.

previous(): Mueve el cursor a el registro anterior. Siempre que se pueda retroceder.

beforeFirst(): Mueve el cursor a BOF.

afterLast(): Mueve el cursor a EOF.

Para usar estos métodos de desplazamiento adecuadamente es necesario saber en qué registro estamos.

Una vez estamos en el registro que buscamos lo que normalmente hacemos es extraer la información que necesitamos y guardarla. Para obtener los valores del resultset tenemos que saber el tipo que tiene ese valor y usamos el método get que necesitamos en función de ese tipo. Estos **métodos get pueden recibir dos parámetros**, el **nombre** de la columna o la **posición**. Si usamos el nombre, ese nombre será el nombre del campo en la tabla, a no ser que en la consulta SQL se haya especificado “alias”, en cuyo caso se usa el alias. Si usamos la posición, esa posición depende de la consulta SELECT, si en la consulta usamos * (todos) la posición es la que tenga el campo en la tabla, si en la consulta hemos especificado unos campos concretos la posición se corresponde a la que hayamos indicado en el SELECT.

Si el resultset no es de tipo **updatable** los datos que contiene no se pueden modificar.

Suponiendo que el resultset sea updatable podemos **modificar los datos con métodos set**, para lo cual también es necesario saber el tipo del valor que contiene el campo que queremos modificar.

Para poder modificar la base de datos de acuerdo a los cambios hechos sobre el resultset, después de realizar los cambios en los campos que contiene el resultset es necesario confirmarlos usando **updateRow()** para actualizar la base datos trabajando en modo conectado o **cancelRowUpdates()** para descartar esos cambios. Solo funcionan trabajando en modo conectado.

Para borrar un registro completo se usa **deleteRow()** trabajando en modo conectado. En realidad normalmente la información nunca se borra sino que cambia de estado o ubicación. Es decir se cambia el valor de un campo o se cambia el registro de tabla, por ejemplo.

Para crear un registro nuevo, siempre que el resultset sea modificable hay que situar el cursor en un punto donde vamos a poder insertar un registro nuevo usando **moveToInsertRow()**, y después enviar los datos necesarios del registro nuevo usando un set para cada campo según el tipo del que sea el campo. Después de eso usamos **insertRow()** para insertar el registro o bien **cancelRow()** si cambiamos de idea. Esto solo será posible en modelos conectados.

Como hemos dicho **hay varios tipos de resultset**, si nos interesa que el resultset haga otras cosas además de las que hace por defecto hay que declararlo antes del statement, en los argumentos del **createStatement()** o del **prepareStatement()**, después del argumento de la query el **primer parámetro** será el tipo de resultset. Que puede ser:

`ResultSet.FORWARD_ONLY` → solo hacia delante, es el que se usa por defecto

`resultSet.TYPE_SCROLL_INSENSITIVE` → se puede mover hacia delante o hacia atrás, pero no considera los cambios en la base de datos.

`resultSet.TYPE_SCROLL_SENSITIVE` → se puede mover hacia delante o hacia atrás, hace comprobaciones para saber si ha habido cambios en la base de datos. Lo hace con consultas constantes de comprobación, lo que hace que sea muy costoso en conexiones.

El **segundo parámetro** es el permiso que tendrá el resultset, por ejemplo

`resultSet.READ_ONLY` → solo lectura, por defecto. Será la más utilizada.

`resultSet.CONCUR_UPDATABLE` → puede modificar los datos de la DB desde el resultset

Si trabajamos con jdbc 4 o superior, puede ponerse un **tercer parámetro** que se usa para indicar si queremos cerrar o no la conexión después de trabajar con el resultset.

Transacciones

A veces nos puede interesar no enviar consultas a la base de datos hasta que no se completa un ciclo determinado de acciones como en el caso de un pedido, en ese caso se envía un grupo de órdenes a la base de datos y eso se llama **transacción**.

Algunas bases de datos necesitan una **confirmación de las órdenes enviadas** antes de ser ejecutadas en la propia base de datos. El objeto Connection envía esta confirmación a través de la propiedad **autoCommit**, por lo tanto nos conviene establecer esa propiedad en **false** para que no confirme hasta que el pedido esté completamente confirmado. Por lo tanto a nuestra conexión le ponemos `setAutoCommit(false);`

Cuando se envían las órdenes que contiene la conexión debemos ir esperando la respuesta afirmativa de conformidad de la base de datos y si se da el caso de que alguna de las órdenes es rechazada hay que deshacer todas las anteriores (`rollback()`). Si todas las órdenes de la transacción van bien, al final de la transacción es cuando confirmamos la transacción (`conexión.commit()`).

Este envío de transacción va en un try/catch para capturar errores en las órdenes y en el catch pondríamos el rollBack(). Si va todo bien al final de las órdenes de la transacción pondremos el commit().

```
Try{
    Sta.execute();
    Sta.execute();
    Sta.execute();
    Sta.execute();
    Sta.execute();
    Sta.execute();
    .
    connexion.savePoint("nombre");
    .
    Conexión.commit();
}catch(SQLException excepcion){
    Conección.rollBack("nombre");
}
```

Existe lo que se llama `savePoint(String nombre)`; para ponerlo entre ordenes de la transacción de tal manera que si falla alguna orden, el rollback, con el parámetro del nombre del `savePoint()`, desharía las ordenes solo hasta ese `savePoint`, y no todas las ordenes de la transacción. Las ordenes que no se rehacen se quedan por un tiempo en la cache de la bases de datos, si pasado ese tiempo no se completa la transacción la base de datos los elimina.

Al trabajar con transacciones se pueden originar algunos problemas, por ejemplo, cuando varios usuarios acceden al mismo tiempo a una misma tabla trabajando con transacciones, pudiera darse el caso de que más de uno quiera modificar un mismo registro antes de que el primero de ellos haya terminado de ejecutar su transacción completa.

Hay tres tipos de problemas que se dan con las transacciones, son **Lecturas sucias (INSERT)**, **Lecturas No Repetibles (DELETE)** y **Lecturas Fantasma (UPDATE)**.

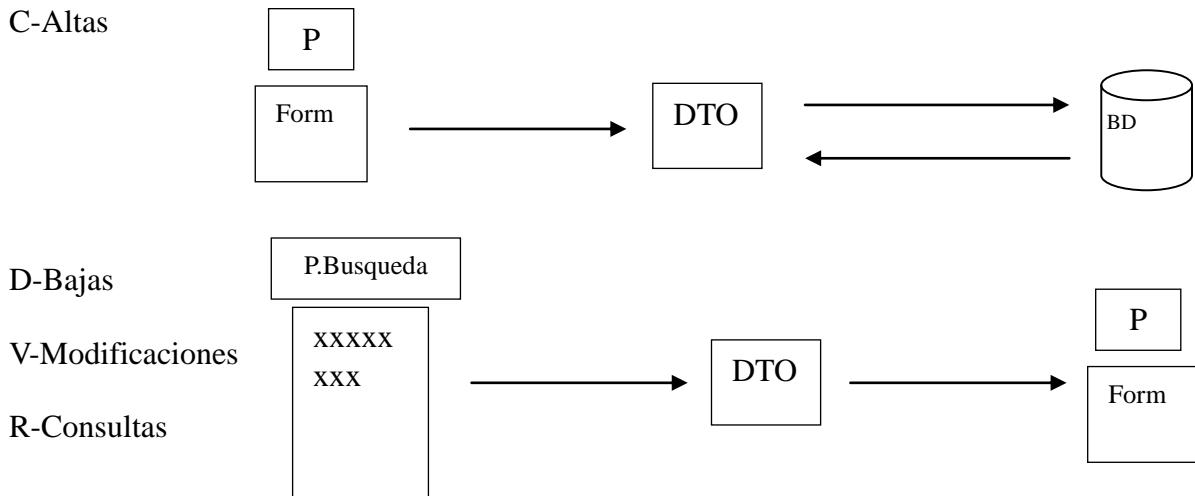
La solución puede ser bloquear la tabla, pero eso no siempre puede ser. El método para bloquear una tabla es `setIsolationLevel()`, que recibe como parámetro estático un nivel de bloqueo determinado según la necesidad y problema concreto, los niveles son constantes de clase de connection, de menor bloqueo a mayor son:

`TRANSACTION.NONE` → no admite transacciones, se confirman las ordenes una a una
`TRANSACTION.READ.UNCOMMITTED` → opción por defecto. No se hace aislamiento
`TRANSACTION.READ.COMMITTED` → bloquea las lecturas sucias (inserciones)
`TRANSACTION.REPEATABLE.READ` → bloquea las lecturas no repetibles (bajas)
`TRANSACTION.SERIALIZABLE` → bloquea las lecturas fantasma (modificaciones)

Estos tipos de problemas con las transacciones requieren soluciones diferentes dependiendo del problema concreto, estas soluciones hay que pensarlas para hacerlas personalizadas a la situación.

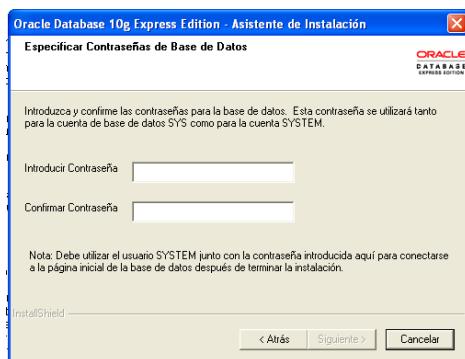
Llevar estos conceptos teóricos a la práctica requiere establecer un patrón de trabajo con una estructura en la que la aplicación llama a una fachada, la cual a su vez se comunica con un DTO (al menos) y un DAO por cada tabla con la que vayamos a trabajar.

Conceptos de trato contra la Base de Datos:



Instalacion Oracle

Ejecutar OracleXEUniv.exe y seguir los pasos. En contraseña nos indica que el usuario es SYS. Asignamos el password deseado. Damos a siguiente hasta que termine.



Es recomendable pasar los servicios de Oracle de automatico a manual para que no arranquen al inicio del sistema (OracleServiceXE y OracleXETNSListener).

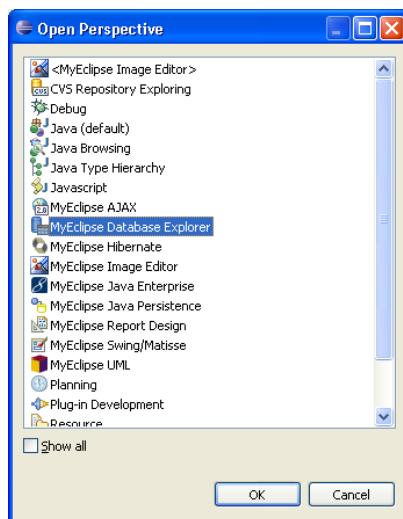
Luego iniciamos el administrador de BBDD con el navegador, ponemos user y pass. Para llenar la BBDD nos metemos en sql y por sentencias.

Creamos el TABLESPACE

Creamos el USER

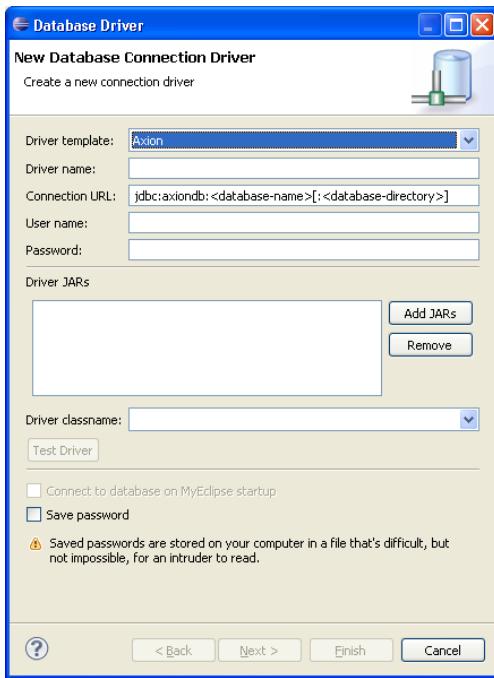
Creamos permisos

En MyEclipse tenemos la perspectiva **DataBaseExplorer** para poder ver la estructura de las bases de datos.).



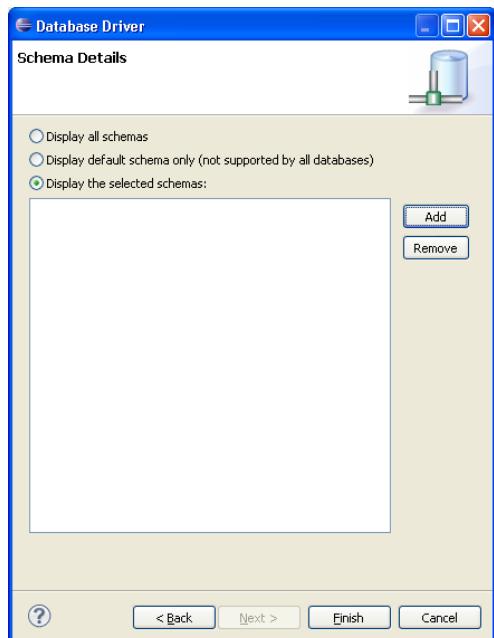
Para usarlo hay establecer la conexión. Necesitamos tener el driver que vamos a necesitar. En el caso del proyecto del master es el driver ojdbc14.jar dentro de la carpeta base de datos/oracle. Para ello botón derecho en la pestaña DB Browse, base de datos (Master). Y New

`jdbc:oracle:thin:@localhost:1521:XE`

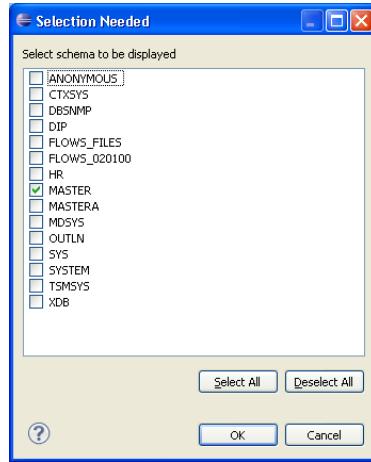


El driver template para el proyecto es “Oracle Thin Driver”. En driver name el nombre que queramos, en connection url la cadena de conexión tal como hemos visto más arriba. El nombre de usuario es el que nosotros tengamos, en nuestro caso master y en password lo mismo. Después pulsamos en Add JARs y buscamos el driver que ya hemos dicho que el del proyecto esojdbc14.jar. Podemos pulsar la conexión en el botón Test Driver.

A continuación Next, aparece la siguiente ventana.



Marcamos “Display the selected schemas” y pulsamos “add” y elegimos la(s) base de datos que necesitamos, que en nuestro caso es MASTER.



Pulsamos en Ok y después en Finish.

En el eclipse se mostrarán las bases de datos, tablas y campos seleccionados y podemos ver los tipos de cada campo y lo que necesitemos para crear nuestras clases. Mirar en la pestaña Table/Object Info.

Column Name	Data Type	Size	Decimal Digits	Default Value	Accept Null Value	Unique	Comments
CODIGO_CLIENTE	NUMBER	5	0		NO	YES	
NOMBRE_CLIENTE	VARCHAR2	35			NO	NO	
TIPO_EMPRESA	CHAR	1			YES	NO	
CIF_CLIENTE	VARCHAR2	10			YES	NO	
NIF_CLIENTE	VARCHAR2	10			YES	NO	
NOMBRE_COMERCIAL_CLIENTE	VARCHAR2	40			YES	NO	
FECHA_ALTA_CLIENTE	DATE	7			YES	NO	
FECHA_BAJA_CLIENTE	DATE	7			YES	NO	
CALLE_CLIENTE	VARCHAR2	40			YES	NO	
CIUDAD_CLIENTE	VARCHAR2	35			YES	NO	
CP_CLIENTE	VARCHAR2	5			YES	NO	
PROVINCIA_CLIENTE	VARCHAR2	35			YES	NO	
TELEFONO_CLIENTE	VARCHAR2	9			YES	NO	
FAX_CLIENTE	VARCHAR2	9			YES	NO	
EMAIL_CLIENTE	VARCHAR2	35			YES	NO	
CONTADO_CLIENTE	CHAR	1			YES	NO	
CREDITO_CLIENTE	CHAR	1			YES	NO	
CIFRA_VENTAS_CLIENTE	NUMBER	11	2		YES	NO	
BENEFICIOS_CLIENTE	NUMBER	11	2		YES	NO	
PREPAGO_CLIENTE	CHAR	1			YES	NO	
CAPITAL_CLIENTE	NUMBER	11	2		YES	NO	
INMOVILIZADO_CLIENTE	NUMBER	11	2		YES	NO	
PASIVO_CLIENTE	NUMBER	11	2		YES	NO	
ACTIVO_CLIENTE	NUMBER	11	2		YES	NO	

Esta vista de mi eclipse también nos permite probar consultas SQL. Si damos con botón derecho en el schema que estamos utilizando hay una opción “New SQL Editor”. Que abre un editor donde escribimos nuestra query. La podemos probar pulsando el triangulo verde y nos arroja el resultado en la parte de abajo.

SQL

Es un lenguaje estándar para comunicación con bases de datos. En realidad no es 100% estándar, ya que las bases de datos tienen sus propias particularidades sobre el SQL para manejar sus propias bases de datos, sobretodo Oracle que es la que estamos usando en el máster.

Con lo cual, cada base de datos es diferente y deben ser bien conocidas si vamos a trabajar con ellas.

SQL es un lenguaje de comandos, no es un lenguaje de programación. Estos comandos se dividen en cuatro grupos para facilitar su estudio y comprensión. Estos grupos, en Oracle, son **DDL**,(Data Definition Language), **DML** (Data Manipulation Language), **TCL** (Transaction Control Language) y **SCL** (Session Control Language).

El que nosotros usaremos más habitualmente será el DML. Estos son algunos de los comandos más usados:

DDL	DML	TCL	SCL
CREATE	INSERT	COMMIT	ALTER SESSION
ALTER	UPDATE	SAVEPOINT	SET ROLE
DROP	DELETE	ROLLBACK	
GRANT	SELECT		
REVOKE	LOCK TABLE		
AUDIT			
NOAUDIT			
ANALYZE			
RENAME			
TRUNCATE			

CREATE, sirve para crear cualquier objeto en la base de datos

ALTER, modificación de objetos

DROP borrado de objetos

GRANT establece permisos a los usuarios

REVOKE quita permisos a los usuarios

AUDIT seguimiento de ejecución de órdenes para conocer sus costes

NOAUDIT

ANALYZE

RENAME cambio de nombre a objetos

TRUNCATE partir tablas

LOCK TABLE bloquea la tabla

COMMIT confirmación de órdenes.

ALTER SESSION manipulación de la session

SET ROLE asignación de roles a usuarios (los roles contienen permisos)

SQL no es case sensitive.

CREATE se usa para creación de tablas y otras cosas, siempre va seguido del tipo de objeto que

vamos a crear, **el concepto de Base de Datos NO existe en Oracle**. Se usa el concepto de **Table Space**.

Las tablas se organizan dentro de table spaces, es una organización lógica y se guardan en ficheros llamados **datafiles**.

Para crear un table space se usa el comando `CREATE TABLESPACE nombre` a continuación se crea el data file `CREATE DATAFILE nombreFichero size 100M`

El nombre del **tablespace** solo admite letras y números hasta 32 caracteres.

El nombre del **datafile** también tiene una limitación de tamaño y al crearse se le asigna también el tamaño, es nuestro caso ha sido 100 megas.

Cuando se crea un datafile se deben crear también usuarios, todo usuario debe tener su contraseña (clave) y además a los usuarios se les asigna un tablespace:

```
CREATE USER nombreUsuario IDENTIFY BY clave DEFAULT TABLESPACE nombreTablespace
```

A los usuarios se les deben asignar permisos a través de roles. El roll **dba** es un role que asume todos los permisos:

```
GRANT DBA TO nombre_usuario
```

Para crear tablas se usa **CREATE TABLE**, en las tablas hay registros y los registros tienen campos, todos los registros de una tabla tienen los mismos campos. Dentro de un tablespace no puede haber dos tablas con el mismo nombre. Los campos deben tener también nombres únicos dentro de una tabla. El tipo de cada campo es algo muy distinto en cada gestor de base de datos, dependiendo del tipo del campo puede ser necesario indicar el tamaño de ese campo

```
CREATE TABLE nombreTabla
{
    nombreCampo tipo (tamaño),
    nombreCampo2 tipo (tamaño),
    nombrecampo3 tipo (tamaño)
}
```

Los tipos en Oracle son:

VarChar2 y Char (Varchar está obsoleto): son cadenas de caracteres, la diferencia es que char es una cadena de longitud fija que siempre reserva ese tamaño a ese campo se use o no y no admite mas, mientras que varchar2 es un campo de texto de longitud variable, teniendo como máximo el tamaño indicado. El tamaño en ambos casos será un numero entero. Char admite 2000 caracteres como máximo y varchar2 4000 caracteres. **En Java su equivalencia es String.**

Number (digitos, decimales): Para definir campos numéricos, se define por dos valores; el primero es la cantidad total de digitos, el segundo es, de esos números cuantos van a ser decimales. Por ejemplo Number(5,2) podría ser 457,54. Dependiendo del tamaño y sus decimales, en java su equivalente será int, long, float, doublé, etc.

Date es el tipo usado para fechas y horas. Funciona internamente en milisegundos igual que en java pero su fecha de inicio no es el 1 de Enero de 1970, sino el 31 de Diciembre del 4713 antes de cristo. Este tipo no recibe tamaño. Date tiene un subtipo que es **timestamp**, el cual mide en milisegundos las horas, se usa para mediciones de alta precisión.

CLOB, Character Long Object, admite hasta 4 gigas de caracteres de texto.

BLOB, Binary Long Object. Lo mismo que CLOB pero para ficheros binarios, como imágenes, video, sonido, etc.

NVARCHAR2, NCLOB. Todos los tipos que comienzan por N están asociados a un servicio de oracle llamado National Service Language, que tiene que ver con internacionalización para diferencias de caracteres según la cultura o el idioma. El DBA debe configurar la base de datos para este tipo de campos.

A los campos también se les pueden dar **restricciones** (constraints), las cuales han de cumplirse al 100% en todos los campos cuando se envía un comando u oracle rechazará la transacción completa.

Las restricciones pueden ponerse al principio o al final del comando de creación de los campos, también usando un alter table.

`CONSTRAINT nombreRestriccion (opcional)`

El nombre de la restricción es opcional pero es recomendable ponérselo y además, que sea identificativo siguiendo este patrón: nombreTabla-nombrecampo-dosletras de tipo de restricción, ya que cuando se produce un error oracle nos indica la restricción violada y si no ponemos un nombre significativo, oracle le da un nombre interno incomprensible. Hay seis tipos de restricciones: **PK, UQ, NN, CK, RF, FK**

PK: Primary Key, la clave primaria en Oracle es una restricción, puede haber tablas sin clave primaria, pero es recomendable y práctica de buen uso poner siempre una clave primaria en todas las tablas (para Hibernate es obligatorio). Un campo o campos de Clave Primaria es aquel cuyo contenido **identifica de forma inequívoca a un registro**. Las tablas solo pueden tener una clave primaria, pero si hubiera más de un campo que pudiera ser clave primaria, esos campos pueden ser considerados claves secundarias, los cuales son también únicos, pero no han sido tomados en cuenta como clave primaria.

UQ: Unique Restriction, es una clave secundaria. Solo puede haber una clave de este tipo, al igual que las claves primarias. Admite Null.

Las claves primarias y secundarias tienen un fichero índice que se crea automáticamente por oracle, y es el que se usa para las búsquedas de registros de forma que no sea necesario leer las tablas completas. El sistema de búsqueda en los índices permite búsquedas rápidas debido a los algoritmos de búsqueda binarias o dicotómicas que dividen los índices por dos a partir de las claves primarias y secundarias. Los índices están ordenados por clave primaria.

NN: Not Null, es un campo que no puede estar vacío.

CK: Es una condición que debe cumplirse para este campo

RF: Reference Key. Tiene que ver con las relaciones con otras tablas. La integridad referencial quiere decir que se evitan errores del tipo de asignar un pedido a un cliente inexistente.

FK: Foreign Key. También tiene que ver con las relaciones con otras tablas. Es un campo común con otra tabla que define la relación con esa otra tabla.

Para definir una restricción hemos dicho que se hace así:

```
CONSTRAINT nombreRestriccion restriction(dos letras)
```

En el caso de las **CK** (check) debe cumplirse una condición esa condición se especifica poniendo entre paréntesis, a continuación de las dos letras CK, el nombre del campo, operador, valor.

El operador que se usa no es siempre igual a los que conocemos por SQL, ya que existen tres que son **LIKE**, **IN** y **BETWEEN**

LIKE nos sirve para condicionales en campos de tipo de caracteres y son para buscar búsquedas por aproximación, es decir admite meta caracteres (expresiones regulares) que en oracle son % y _
% es cualquier cantidad de caracteres
_ es cualquier carácter (solo uno)

IN sirve para buscar un valor dentro de una lista y devuelve true si está en la lista, la lista va después del IN con los valores entre paréntesis separados por comas.

BETWEEN busca entre un rango de valores, la sintaxis es paréntesis punto punto valor final, así: (inicio .. final). Muy útil para campos de tipo Date (fechas u horas).

Los campos auto-numéricos no existen como tipo en oracle se crean como **sequence**, **con create**, así: **CREATE SEQUENCE nombre**. Los secuenciadores son objetos aparte que no tienen nada que ver con ninguna tabla.

Instalación de Oracle XE

El modelo XE es la edición express, que es la adecuada para pruebas en casa sobre Windows. Se puede descargar de oracle gratuitamente. Nosotros tenemos una copia, es un archivo llamado OracleXEUniv.exe que se instala como cualquier programa Windows y que nos pedirá la carpeta de instalación y la contraseña del administrador.

Una vez instalada la base de datos veremos que además habrá instalado otras cosas. La instalación se hace como un servicio, con lo cual es buena idea desinstalar el servicio desde panel de control – Herramientas administrativas y en servicios buscamos los servicios de oracle que serán varios y el llamado **oracleXETNSListener** los detenemos y después le cambiamos el tipo de inicio con botón derecho – propiedades y cambiamos de automático a **manual**. De esta forma podremos iniciar la base de datos cuando nosotros queramos.

Para arrancar la base de datos encontraremos una opción en Inicio - programas instalados que usaremos para arrancar o parar la base de datos. Al pulsarlo se abre una ventana de consola donde se ejecuta el script. Después de que se haya ejecutado podemos cerrar esa ventana.

Además de la base de datos se instala un pequeño entorno de gestión al que podemos acceder por el enlace “Ir a la página principal de la base de datos” dentro del directorio de la instalación de oracle en Inicio – todos los programas (win xp). Es una página web donde nos pide usuario y contraseña la primera vez podemos usar el usuario system y la contraseña que le hayamos indicado durante la instalación.

Los ficheros de texto que tenemos en la carpeta base de datos/bd-proyecto llamados TABLAS_ORACLE_PROYECTO.sql y datos_prueba_proyectoJ2EE.sql tienen los comandos sql para la creación de las tablas y registros que necesitamos para el máster.

Antes de nada nos logeamos con el usuario system y creamos el usuario master que es el que usamos en el proyecto. Después nos deslogueamos de system y volvemos a entrar como master. Ya como usuario master podemos crear las tablas y poblarlas de registros para que esas tablas estén relacionadas con nuestro usuario master.

Comandos DML

INSERT

Las inserciones de registros deben cumplir a rajatabla las restricciones.

Si le vamos a dar valores a todos los campos no hay que indicar cuales son, usaríamos VALUE directamente seguido de la información, si no vamos a darle valor a todos ponemos sus nombre seguidos de comas

```
INSERT INTO nombreTabla (valores en el orden natural) // caso de
insertar valores en todos los campos
INSERT INTO nombreTabla (nombreCampos) VALUE (valores en el mismo orden)
```

Las tablas oracle, al no tener auto-numérico, necesitan una clave principal, y un secuenciador (SEQUENCE) que será llamado al crear un INSERT ([obj.secuenciador.nextval](#)). La llamada se hace en el campo del VALUE que le corresponda al PRIMARY KEY. El secuenciador automáticamente va incrementando el valor de la clave primaria según la configuración que le hayamos dado y se encarga la base de datos.

DELETE

Este comando puede ser que en realidad no borre un registro, pero no es un tema a tratar ahora.

```
DELETE FROM nombreTabla // Borra la Tabla completa si no se especifica un registro
DELETE FROM nombreTabla WHERE condicion // borra el/los registros que cumplen la
condición
```

Una vez confirmada con COMMIT el borrado ya desaparece de la cache de oracle y no se puede recuperar.

Si el registro que queremos borrar no existe no da un error, simplemente no borra nada.

En la condición los valores numéricos van a pelo, las cadenas y fechas entre comillas simples o dobles.

DELETE no devuelve el numero de filas afectadas pero el método **execute** de Java sí.

UPDATE

Modifica registros de las tablas.

```
UPDATE nombreTabla SET campo = valor, campo2 = valor2 // modifica esos campos
en todos los registros de la tabla
```

```
UPDATE nombreTabla SET campo = valor, campo2 = valor2 WHERE condición // 
modifica los registros que cumplen la condición
```

SELECT

Es el comando más usado y más potente de SQL. Tiene múltiples formatos.

```
SELECT nombresCampos FROM nombresTablas WHERE condicion
```

Pueden especificarse varios campos separados por comas y varias tablas separadas por comas.

Si se especifican campos en un orden determinado llegarán en ese orden al ResultSet. Para especificar que se quieren todos los campos se usa el asterisco (*).

Se pueden especificar ALIAS para los nombres de los campos. En oracle los alias se especifican de esta forma **nombreCampo espacio Alias**, en una consulta SELECT con aliases, el resultset recibirá ese valor con el nombre del alias, no con el nombre del campo y el alias es válido solo para esa consulta, no modifica nada en la tabla. Las tablas también pueden llevar alias de la misma manera, **nombre tabla espacio nombre alias**

Las consultas a más de un tabla a la vez tiene que tener un campo común (**integridad referencial**) para establecer la unión, sino deben usarse productos cartesianos, no existen los JOIN en oracle.

Para realizar uniones de tablas en un select, podemos usar un ejemplo de caso real en una tabla de pedidos, que en realidad son dos tablas (pedidos y lineaPedidos), con un campo común con el mismo nombre (numeroPedido). En pedidos está el pedido en sí y en lineaPedidos están los artículos que conforman ese pedido. Esto se hace usando una pregunta por campo común, dos tablas, una pregunta, tres tablas dos preguntas.

Caso de dos tablas

```
SELECT P.numeroPedido, codigoCliente, codArticulo, unidades  
FROM pedido P, lineaPedido LP  
WHERE P.numeroPedido = LP.numeroPedido
```

Caso de tres tablas

```
SELECT P.numeroPedido, codigoCliente, LP.codArticulo, unidades, stock  
FROM pedido P, lineaPedido LP, articulos  
WHERE P.numeroPedido = LP.numeroPedido  
AND LP.codigoArticulos = articulos.codigoArticulos
```

Se pueden hacer agrupaciones, es decir, se obtienen tantos registros como veces cambia de valor el campo por el que agrupo. El comando es **GROUP BY**.

Un ejemplo de uso de este comando es cuando queremos saber cuantos clientes hay por provincia, solo cuantos. Esta consulta sería así:

```
SELECT nombreProvincia, COUNT(*) FROM clientes GROUP BY nombreProvincia
```

Además a los grupo se les pueden añadir condicionales usando la cláusula **HAVING**, que solo se puede usar cuando haya GROUP BY.

Por ejemplo para obtener el numero de morosos por provincias haríamos esto:

```
SELECT nombreProvincia, COUNT(*) FROM clientes GROUP BY nombreProvincia  
HAVING balanceCliente < 0
```

También es posible ordenar el resultado del SELECT usando la cláusula **ORDER BY nombreCampo**. El orden ascendente se refleja **ASC** y el descendente con **DESC**.

Así mismo es posible hacer consultas de consultas, o lo que es lo mismo hacer subconsultas, es decir un SELECT dentro de otro SELECT.

```
SELECT * FROM nombretabla WHERE nombreCampo operador (SELECT....)
```

Un operador muy útil en estos casos es **IN** ya que puede haber más de un valor devuelto en la subconsulta. También podría usarse **BETWEEN**

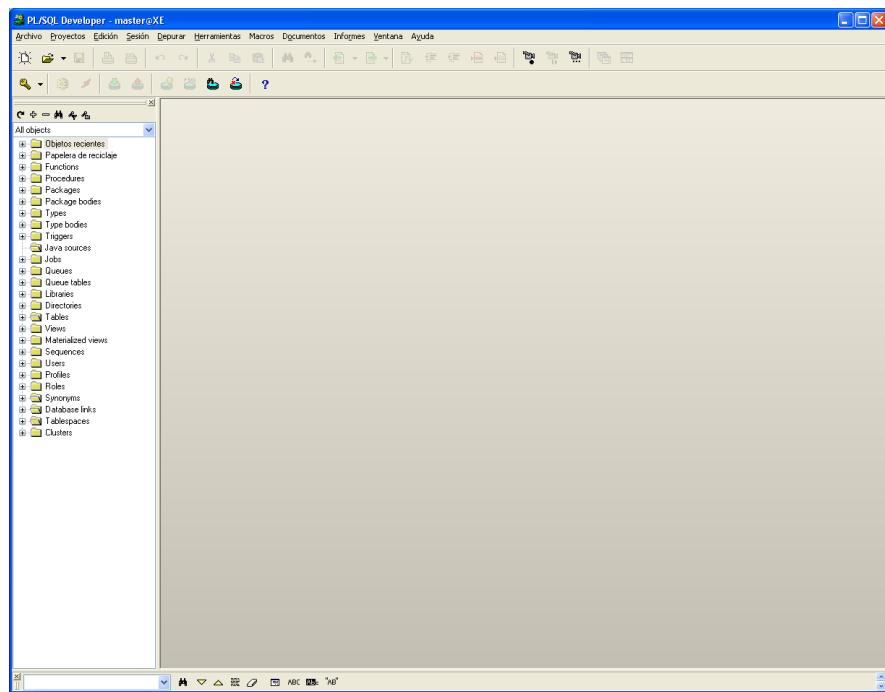
Esto se daría en el siguiente escenario, suponemos que tenemos una tabla que nos da las delegaciones de una empresa en toda España. Otra tabla en la cual defino los departamentos y estos departamentos tienen sus nombres, códigos etc, otra tabla de Empleados con campos comunes, etc. Lógicamente los empleados son distintos de una delegación a otra, si quisieramos saber todos los empleados del departamento de contabilidad, por ejemplo.

En este caso tenemos que consultar la tabla de departamentos para saber el código del departamento de contabilidad y usar ese dato para seleccionar los empleados de ese departamento.

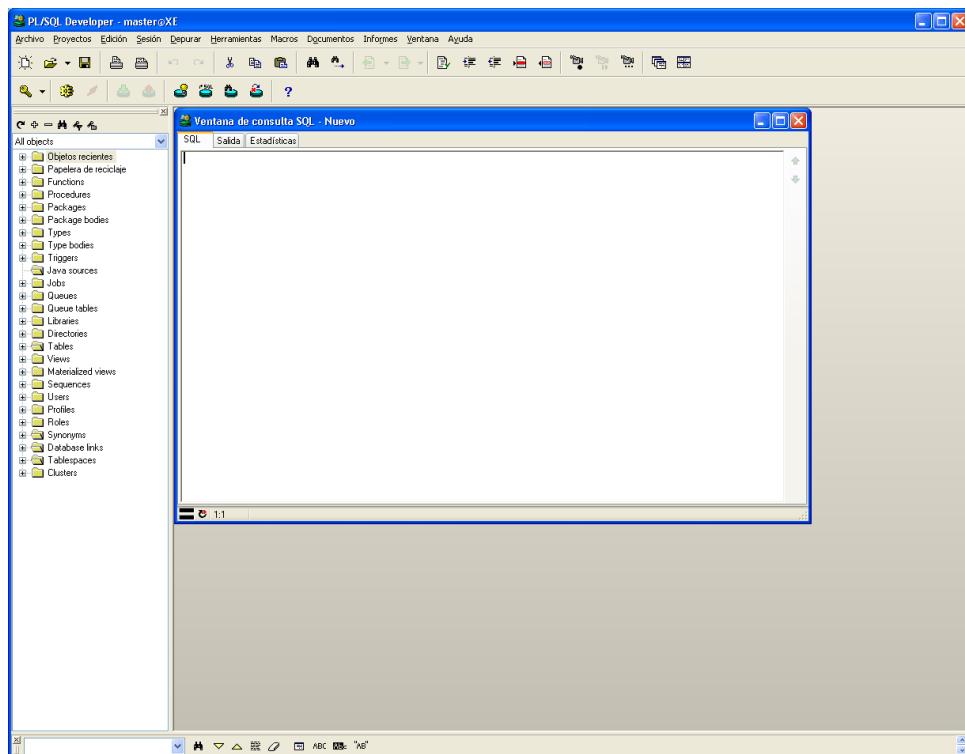
Cuando se nos dé un caso como este es muy aconsejable crear primero la subconsulta y probarla y una vez seguros de que la subconsulta está bien creamos la consulta principal (externa).

Para hacer prácticas con oracle es buena una herramienta llamada **PL/SQL Developer**. En el master tenemos la versión 7 con dos claves que podemos usar para usarlo indefinidamente.

Una vez instalado tiene este aspecto.



Para abrir una ventana en la que escribir consultas se va a Archivo - Nuevo – ventana de consulta SQL.



Tenemos un archivo en formato Word en la carpeta base de datos llamado EJERCICIO.DOC que es un examen.

Una página web que es de ayuda para la sintaxis del SQL de oracle y demás es <http://ora.u440.com/>

Análisis de Bases de Datos

Para poder manejar eficientemente las bases de datos es muy importante saber porque se diseñan de esa manera las bases de datos relacionales. Para eso hay que emplear las técnicas de análisis de bases de datos.

Primer paso, partiendo de una hoja en blanco el analista comienza a crear una lista de **entidades candidatas**, entendiéndose por **entidad una tabla**, así como un **atributo es un campo** de una tabla y **una tupla es un registro**.

Esta lista de entidades candidatas es un listado de tablas que definen el tipo de información que tienen que mover y/o almacenar.

El segundo paso es definir los atributos (campos) de cada una de esas entidades (tablas). Este es un paso mucho más largo. Ya que hay que definir el tipo, tamaño, etc, de cada uno de los campos de cada entidad.

Como **tercer paso** se realiza un estudio de las claves, donde hay que localizar las claves primarias, secundarias y las claves ajenas (foreign key FK). Estas claves pueden ser atributos ya definidos o se tienen que crear. Este estudio se hace sin tener las entidades relacionadas.

El **cuarto paso** es el estudio de las relaciones entre las entidades. Se decide cuales han de tener una relación directa y cuáles no. Cuando se detecta una relación entre entidades se calcula la **cardinalidad**, es decir en una relación por ejemplo entre una entidad cliente y una entidad pedido se ve, a cuantos clientes les corresponde un pedido? y viceversa, a cuantos pedidos les corresponde un cliente?, de forma que la clave principal de uno de ellos se convierte en una clave ajena del otro. Es decir en una relación de muchos a 1 como la que hay entre pedidos a clientes, se coloca la clave principal de clientes como clave ajena de pedidos.

Las relaciones pueden ser de 1 a 1, de 1 a muchos y de muchos a 1. Cuando las relaciones son de 1 a muchos en un sentido y de 1 a muchos en el otro sentido, eso se llama una **relación compleja** y no puede crearse sin simplificarla, es decir creando una tabla, llamada “**de cruce**” con los registros principales implicados en esta relación de ambas tablas.

Teniendo lo anterior hecho quedaría un **último paso** que es la **normalización**. Se trata de comprobar que cada una de las entidades cumple una serie de reglas. Esas reglas son las **reglas normales**. Se han creado para evitar errores de diseño.

Las tres primeras (hay unas 12 ó 14) formas normales más habituales son:

- 1.- Se dice que una entidad se encuentra en primera forma normal (1F) si, y solo si, cada uno de los campos contiene un único valor para cada registro. Evita repeticiones.
- 2.- Compara todos y cada uno de los campos de la tabla con la clave primaria. Si todos los campos dependen directamente de la clave primaria se dice que la tabla está en segunda forma normal. Encuentra dependencias débiles y cuando se encuentran indica que ese campo debería estar en otra tabla donde si tenga una dependencia directa con su clave primaria.

3.- Se dice que una tabla está en tercera forma normal si, y solo si los campos de la tabla dependen únicamente de la clave primaria. Es decir que no dependen unos de otros. Aquel atributo que no cumple la forma normal debe estar en otra tabla donde si lo cumplira o en una tabla nueva.

Cuando se detecta que una entidad no cumple alguna de las formas normales se debe crear una nueva entidad sacando esos registros que no cumplen la norma a otra entidad donde sí se cumplan las normas con la cardinalidad adecuada.

Una vez hecha la normalización resultara en una base de datos fácilmente ampliable y este es el soporte que es pasado a diseño donde se decide qué gestor de base de datos se va a usar.

HIBERNATE

Es un marco de trabajo de tipo **ORM** (Object Relational Mapping) “Mapeo Objeto Relacional”. Se encarga de los detalles sucios de este trasiego de trabajo con las bases de datos y lo hace muy bien.

También genera automáticamente el código SQL para atacar a la base de datos.

Establece estrategias de cacheo y de acceso.

Es una herramienta complicada de uso.

Es libre y gratuito, aunque a partir de la versión 3 se hizo cargo de ella JBoss. www.hibernate.org

Se establece una norma de trabajo igual para todas las bases de datos y tablas y de esa manera se pueden automatizar los procesos de trabajo.

Las restricciones son que la base de datos debe estar bien **normalizada**, con lo cual no siempre puede usarse. Por ejemplo **todas las tablas deben tener clave primaria**.

Como hibernate es capaz de hacer muchas cosas él solo, se le debe explicar cómo hacerlo. Esta explicación se da a partir de ficheros XML.

Con esto deducimos que lo primero que hay que hacer es crear la estructura. El fichero principal de configuración para hibernate se llama **hibernate.cfg.xml**

Ese fichero tiene tres tipos de información:

1. Datos de conexión a la base de datos con jdbc
2. Información de la infraestructura. Donde están y cómo se llaman los archivos de configuración de las tablas (clases de persistencia) con las que vamos a trabajar.
3. Datos de configuración de hibernate [opcional]. Hibernate ya tiene opciones por defecto y pueden ser modificadas aquí.

El libro **Developer Guide** de Hibernate explica que hibernate está perfectamente ajustado para funcionar en todos los casos, con lo cual si vamos a cambiarlos en los archivos descriptores hay que estar muy seguro de lo que se hace.

A partir de ahora en el máster toda la configuración se hará a través de ficheros XML. Con lo cual vamos a aprender muchos nombres de etiquetas.

Hibernate es un conjunto de archivos JAR que contienen todas las clases que necesitamos y nosotros escribiremos las clases que van a usar esas clases. Que básicamente serán las fachadas, ya que todo lo demás lo hace hibernate.

hibernate.cfg.xml

Comienza con dos encabezados (prologo) como cualquier archivo XML y a continuación una

etiqueta de apertura **<session-factory></session-factory>**, sería así:

```
<session-factory>

// datos de configuración de hibernate
// hay que indicarle la base de datos para que sepa que dialecto SQL debe usar (son clases)
<property name="dialect">
    org.hibernate.dialect.oracle10Dialect // para versiones 9 o
superior de oracle
</property>
// información de la conexión
<property name="....">
Dentro de cada name va:
    connection.url
    connection.driver_class
    connection.password
    connection.username

</property>

// información de la infraestructura de la base de datos
<mapping resource="ruta de paquetes y nombre del fichero" />
// nombre del fichero debería llamarse nombreTabla.hbm.xml
// lleva una etiqueta mapping por cada tabla que queramos usar, más una general

// otras properties son show.sql, format.sql y use.sql.comment
// generan información sobre el funcionamiento de hibernate hay que ponerlas en true para que se
activen, se usan en fase desarrollo luego en fase de producción se quitan
<property name="show.sql"> // muestra la orden
    true
</property>
<property name="format.sql"> // muestra la orden bien indentada para su
mejor comprensión
    true
</property>
<property name="use.sql.comment"> // coloca a la sentencia sql una línea
que indica desde qué método se ha ejecutado esa orden
    true
</property>
</session-factory>
```

Este es el fichero principal de configuración, pero puede haber más.

Ejemplo completo:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<!-- Generated by MyEclipse Hibernate Tools. -->
<hibernate-configuration>

    <session-factory>
```

```

<property name="dialect">
    org.hibernate.dialect.Oracle9Dialect
</property>
<property name="connection.url">
    jdbc:oracle:thin:@a3-1m:1521:XE
</property>
<property name="connection.username">master</property>
<property name="connection.password">master</property>
<property name="connection.driver_class">
    oracle.jdbc.driver.OracleDriver
</property>
<property name="myeclipse.connection.profile">master</property>
<mapping resource="com/atrium/hibernate/Provincias.hbm.xml" />
<mapping resource="com/atrium/hibernate/Municipios.hbm.xml" />

</session-factory>

</hibernate-configuration>

```

Hibernate puede conectarse también a través de un pull de conexiones para conectarse a un servicio igual que jdbc.

Como hemos dicho, para que hibernate funcione bien hay que darle todos los detalles de las tablas y sus campos, al menos de aquellos que vayamos a utilizar. Esto se hace a través de ficheros descriptores como hemos visto, y una tabla puede necesitar más de un fichero descriptor dependiendo del uso que le vayamos a dar.

Dentro del session factory, tendremos una etiqueta mapping por cada objeto de persistencia(DTO). Debemos tener un fichero por cada

En los **ficheros descriptores de las tablas** se usa la etiqueta <class>, estos ficheros son **secundarios**, y describen un mapeado (clave/valor). Unen una clase del DTO con una tabla . En los JavaBeans, las propiedades tienen que ser siempre objetos para trabajar con **hibernate**. Todas las tablas tienen que tener clave primaria, es decir, deben estar normalizadas.

Por ejemplo sería:

Fichero.xbm.xml

```

<hibernate-mapping>
    <class      name="paquete.nombreClase"          table="nombreTabla"
    schema="tableSpace">
        <ID      name="nombrePropiedad"      clavePrimaria"      type="tipo
        propiedad(objeto solo)">
            <column name="nombreCampoClavePrimaria"
            precision="numeroDigitos" scale="numeroDecimales" />
            <generator class="assigned" />
        </ID>
        <property name="nombrePropiedadClase" type="...">
            <column name="nombreCampo" ... />
        </property>
    </class>
</hibernate-mapping>

```

ID se encarga de definir únicamente la clave principal de la tabla.

Cuando la clave principal se compone de más de un campo la etiqueta ID debe ser **composite-id** y debe llevar en su cuerpo las etiquetas key-property con su column igual que las etiquetas id. Además debe crearse una clase que represente la clave principal esa clase debe ser un javabean que sobrescribe la clase equals y la clase hashCode, esta clase la crea hibernate automáticamente.

La etiqueta **property** (campo) habla de las demás etiquetas. Un property por etiqueta (campo) que queramos manejar.

El tipo, dentro de ID, debe especificarse siempre con paquete + tipo (no primitivo).

Generator es usado solo para las altas (insert), tiene tres opciones; se lo damos nosotros por programa (**assigned**), lo genera la base de datos o lo genera hibernate.

Si usamos un generator “sequence” hay que darle el nombre a mano en el archivo de configuración, así:

```
<generator class="sequence">
    <param name="sequence">nombreSecuenciador</param>
</generator>
```

Los generadores son objetos que no tienen nada que ver con las tablas.

Nos faltaria contarle a hibernate las relaciones:

Obj 1 – 1 Obj

Obj 1 – M Colecc Obj

Colecc Obj M – 1 Obj

Colecc Obj M – M Colecc Obj

Las colecciones suelen ser set para que no haya objetos repetidos.

Las relaciones funcionan por lazy need (carga vaga). Si no se pide el campo de relacion no viene

Un ejemplo real de archivo descriptor para una tabla “PROVINCIAS” sería este:

```
<hibernate-mapping>
    <class     name="com.atrium.hibernate.Provincias"      table="PROVINCIAS"
schema="MASTER">
        <id   name="codigoProvincia" type="java.lang.Byte">
            <column name="CODIGO_PROVINCIA" precision="2" scale="0" />
            <generator class="assigned" />
        </id>
        <property name="provincia" type="java.lang.String">
            <column name="PROVINCIA" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

Existen plugins en MyEclipse que facilitan la labor con hibernate.

Como dijimos, hibernate es un conjunto de clases que valen para hacer un determinado trabajo.

El propósito de usar Hibernate es aislar la base de datos y olvidarnos de ella para poder trabajar con ella como si fueran objetos. Hibernate, a través de esos objetos, trabajará con la base de datos actualizándola, accediendo a ella, etc. Por eso a Hibernate hay que explicarle muy bien la estructura de la base de datos, eso se hace a través de los ficheros descriptores explicados más arriba. Además de eso, tenemos que crear nuestras clases para trabajar con Hibernate (facade), lo que hasta ahora eran los DTO ahora serán las **clases de persistencia**. Para usar estas clases de persistencia usaremos el patrón DAO (clases que generará hibernate) que contendrán las órdenes de acceso, pero en este caso en vez de tener las ordenes de acceso de jdbc tendrá las de hibernate. Necesitaremos también una clase para la conexión (que también creará hibernate a partir de los datos que le daremos).

Es decir, las dos clases principales dentro de hibernate serán **SessionFactory** y **Session**:

SessionFactory leerá los ficheros de configuración, también comprueba que la configuración es correcta y además también creará los objetos session (conexión) que nosotros manearemos, **cada objeto session representa una conexión a la base de datos**. Esta clase es pesada y solo necesitaremos uno para cada base de datos. **Lo que entendíamos por conexión ahora se llama Session.**

Este objeto **Session** será el que reciba las ordenes para la base de datos, desaparecen los resultset, connection, statement, preparestatement, etc. Todo lo crea Hibernate internamente.

Tanto **Session** como **SessionFactory** son interfaces, y se trabaja con ellas a través de las clases DAO que nos ha generado Hibernate.

Para dar un **alta** se usa el método **save(obj)** de la clase Session. Sería `Session.save(object);` el parámetro object que recibe save es la clase de persistencia y puede tener tres estados:

- 1.- La clase de persistencia antes de ser enviada está en un estado que se llama en **tránsito**.
- 2.- Una vez que la clase de persistencia está enviada a la base de datos a través del método save se dice que se encuentra en un estado **persistente**, es decir la conexión está abierta y la información está tanto en la clase como en la base de datos.
- 3.- cuando cerramos la conexión, hibernate ya no está al tanto de los cambios que pueda haber en la base de datos y se dice que su estado es **desconectado**.

Con lo cual el método save solo puede recibir un objeto en estado de transito.

Existe también el método **saveorupdate(obj)**, el cual hará una **modificación o un alta** dependiendo de si el objeto ya existe o no. Es decir primero hace un select y dependiendo de si el registro existe o no hará un update o un insert.

Para **modificar** existe el método **update(obj)**. Que puede modificar todo menos la clave principal.

Para **borrar** existe el método **delete(obj)**. Donde obj puede ser solo la clave principal, los demás campos no son necesarios.

En hibernate, al contrario que JDBC, el auto-commit está en FALSE. Por lo tanto, siempre después de un comando atómico hay que transaccionar. Estas cosas, que están hechas al contrario de lo que es el estándar se llaman **antipatrones**.

Este antipatón puede llevar a un error muy crítico ya que al no confirmar por defecto, si nos olvidamos, oracle no da error sino que guarda en cache la consulta a la espera de recibir el commit, por otro lado, Hibernate guarda también en su cache la orden dando la apariencia de que la orden es correcta.

Un ejemplo de código sería este:

```
Transaction trans = obj.session.beginTransaction();
Try {
    Obj.session.save(transito);
    trans.commit();
} catch {
    trans.rollback();
}
```

Para las consultas **SELECT** que reciben un solo registro, la clase session tiene un método llamado **get(obj, objClavePrincipal)**, **obj** es la clase de persistencia que tendrá los campos que queremos consultar, y **objclavePrincipal** es un objeto que representa la clave principal, es decir si es un int tendrá que ser un Integer. Este método **get sirve para obtener un solo registro** y nos devolverá un objeto que será nulo si esa clave primaria no existiera.

Para otros tipos de consultas SELECT que devuelven más de un registro se usa la interface **query** que se crearía así:

```
Query consulta = obj.session.createQuery();
```

Con lo cual para trabajar con query hay que tener instancia de session y de sessionFactory.

Query utiliza el lenguaje SQL de Hibernate llamado **HQL** (Hibernate Query Language) y que es independiente de la base de datos con la que se trabaje.

Como hibernate no usa ni tablas ni campos HQL no los tiene tampoco, con lo cual HQL solo maneja objetos y propiedades de objetos. Las sentencias comienzan por **FROM nombreClase** que sería el equivalente a **SELECT * FROM nombreClase**

Una consulta sería:

```
FROM nombreClase AS nombreInstancia WHERE nombreObjPropiedad operador ...
```

En el caso de las relaciones de tablas, Hibernate las resuelve igualmente a través de objetos. En las relaciones de tipo 1 a muchas, muchas a 1, 1 a 1 y muchos a muchos. Hibernate las resuelve así:

1 a 1: son dos objetos

1 a M: es 1 objeto y una colección de objetos

M a 1: Una colección de objetos a 1 objeto

M a M: Una colección de objetos a una colección de objetos

Estas colecciones se crean usando el paradigma de la **carga vaga**. Es decir están vacías y se les da contenido solo cuando se necesita acceder a ellas. Por lo tanto el objeto tiene que ser **persistente**, la

sesión debe estar abierta (la conexión). No puede estar en tránsito ni desconectado. Con lo cual, si la conexión está cerrada hay que reabrirla y para eso se usa el método **merge(obj desconectado)** que devuelve un objeto persistente (conectado).

Pongamos el supuesto caso de una base de datos con tres tablas usuarios, roles y tareas, que se relacionan de manera que cada usuario solo tiene un rol, un rol puede tener muchos usuarios y cada rol se compone de muchas tareas, y cada tarea puede pertenecer a muchos roles.

Esto en código es así:

En el caso de una **relación muchas a 1** se usa la etiqueta **<many-to-one>**

```
<many-to-one name="nombrePropiedad" class="clasePersistencia.nombre">
```

Esta etiqueta many-to-one puede tener otros atributos como **fetch** y **lazy** que sirven para especificar cómo (fetch = “select”) y cuando (lazy = “true”) se satisface la relación y no llevan valores por que ya los tienen por defecto. Fetch tiene también la opción de JOIN que creará un JOIN en vez de un SELECT. Dentro de many-to-one va la etiqueta **<column>** con la propiedad name para el nombre del campo y los atributos habituales de precisión, length, etc.

```
<many-to-one name="nombrePropiedad" class="clasePersistencia.nombre">
    <column name... />
</many-to-one>
```

En el caso de una **relación 1 a muchos** se usa la etiqueta **<one-to-many>**

```
<one-to-many name="nombrePropiedad" class="clasePersistencia.nombre">
```

En este caso **la propiedad será una colección**. Va dentro de una etiqueta **set** **<set name="nombreColección">**, dentro de set va una etiqueta **<key>** que recibe la columna que se va a relacionar en el lado de los muchos.

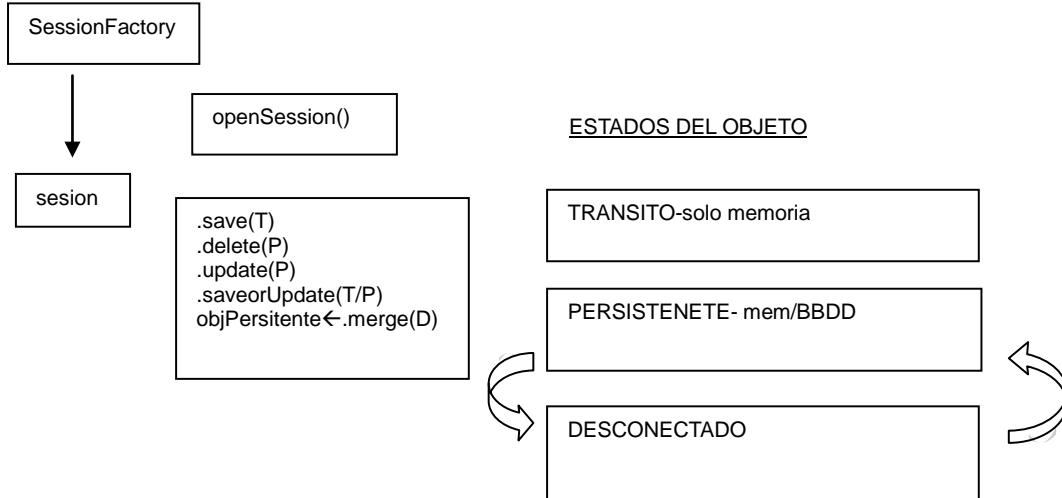
```
<set name="nombreColección">
    <key>
        <column name="nombreCampo" precision... />
    </ key>
    <one-to-many
        name="nombrePropiedad"
        class="clasePersistencia.nombre">
    </ set>
```

En el caso de una **relación muchos a muchos** la etiqueta es **<many-to-many>**. Hay que decirle la columna por la cual tiene que buscar ese valor. Y darle la propiedad **entity-name** que indica la clase (paquete + nombre) con la que se relaciona. También hay que decirle cual es la **clase de cruce**, dentro de una etiqueta **<set>**. Dentro de set en una etiqueta **<key>** va la columna con la que se relaciona.

```
<set name="nombre" name="tablaCruce" schema="baseDatos">
    <key>
        <column name="..." />
    </key>
    <many-to-many entity-name="paqueteNombreClase">
        <column name="..." ... />
    </many-to-many>
</set>
```

Podríamos también poner el caso de **inventar una relación que no existe** y Hibernate lo aceptaría, este caso podría darse en un proyecto real en el que **queremos relacionar dos tablas que no tienen relación en la base de datos**.

Para poder hacer esto tenemos que crear nosotros la relación en los archivos de configuración de Hibernate. Siguiendo las indicaciones sobre las relaciones explicadas más arriba.



Ver `HibernateSessionFactory.java`

Un **objeto en transito** es un objeto que esta solo en memoria, no en la base de datos. Cuando hacemos una consulta a la BBDD y lo guardamos en un objeto tenemos un **objeto persistente**, esta en memoria y bbdd (sesion abierta). Cuando tenemos un objeto con info cargada de la bbdd y se cierra la sesion, es un **objeto desconectado** y no satisface la carga vaga.

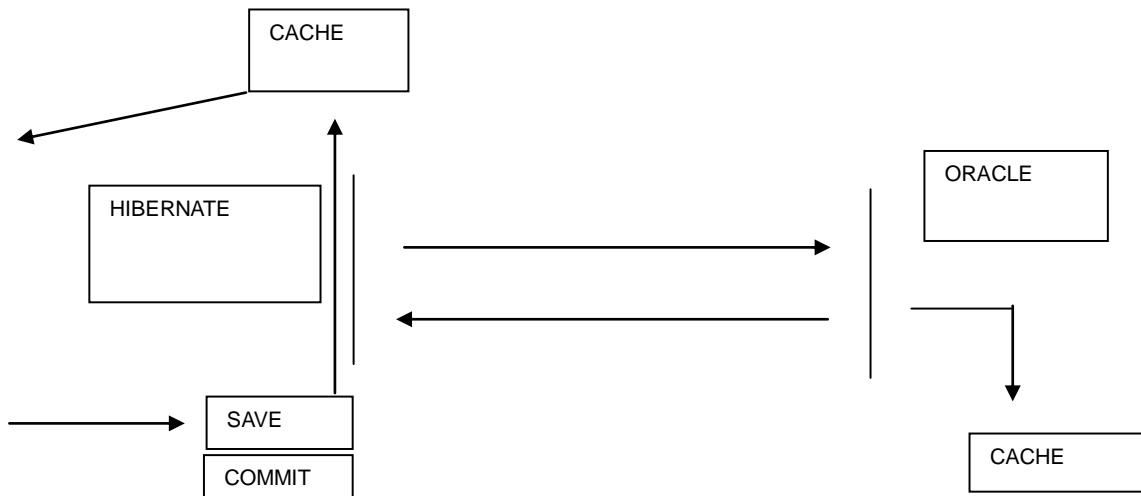
El metodo `.merge()` devuelve un objeto desconectado.

Hibernate tambien trata las transacciones a traves del objeto session

```

Transeaction tran = Session.beginTransaction();
Try{
    .....
    Tran.commit();
} catch(HibernateException HE) {
    Tran.Rollback
}
  
```

Es importante saber donde se implementa la transaccion. Tanto la transacción como el cierre de la conexión se colocan en la fachada, no en el DAO. En el DAO deben estar las operaciones atomicas. Hibernate tiene el autocomit a false, por lo que no se realizan por defecto. Hay que transaccionar las operaciones atomicas, es decir, un save, un update, un delete.



LAS CONSULTAS

Las consultas en hibernate se hace en HQL. Para hacer una consulta llamamos al objeto **session** y a su **metodo createQuery("HQL")**. No hay select en HQL

Objeto HQL:

```

FROM Clase TO nombre_objeto
WHERE  nombre_objeto.propiedad(.propiedad si la primera prop es un
objeto)
OR/AND

```

Las condiciones se aplicaran a las propiedades de los objetos. Una vez tenemos el objeto hql hacemos:

```

List<consulta>.list();
.uniqueResult();
.setFetchSize();ç

```

Ejemplo:

```

Public void alta_Provincia(Provincias provi){

    Transaction Tran = null;
    Try{
        Tran = pro_DAO.getSesion().beginTransaction();
        Pro_DAO.save(prove);
        Tran.commit();
    }catch{
        Tran.rollBack();
    }
}

```

```
    Pro.DAO.getSession().close();  
}  
}
```

Ejemplo alta

```
Public static void main(String args[]){  
    Gestion_Provincias ges_pro = new Gestion_Provincias();  
    Provincias provi_nueva = new Provincias();  
    provi_nueva.setCodigoProvincia(new byte((byte)83));  
    provi_nueva.setProvincia("prueba_j511");  
    ges_pro.altaProvincia(provi_nueva);  
}
```

La concurrencia de sesiones para que cada DAO tenga una sola sesión lo lleva a cabo el sessionFactory a partir de su objeto configuración. Con el método threadLocal controlará que no haya concurrencia de sesiones. Puedo tener más de un sessionFactory, ya que sus sesiones son diferentes entre sí. Este escenario será necesario cuando nos conectemos a varias bases de datos.

Cuando la clave primaria son dos campos... propiedades privadas, constructor sin argumentos, accesores públicos y sobreescribir métodos hashCode e equals. HashCode resuelve problemas de identidad, ya que el contenido puede ser igual para dos objetos distintos.

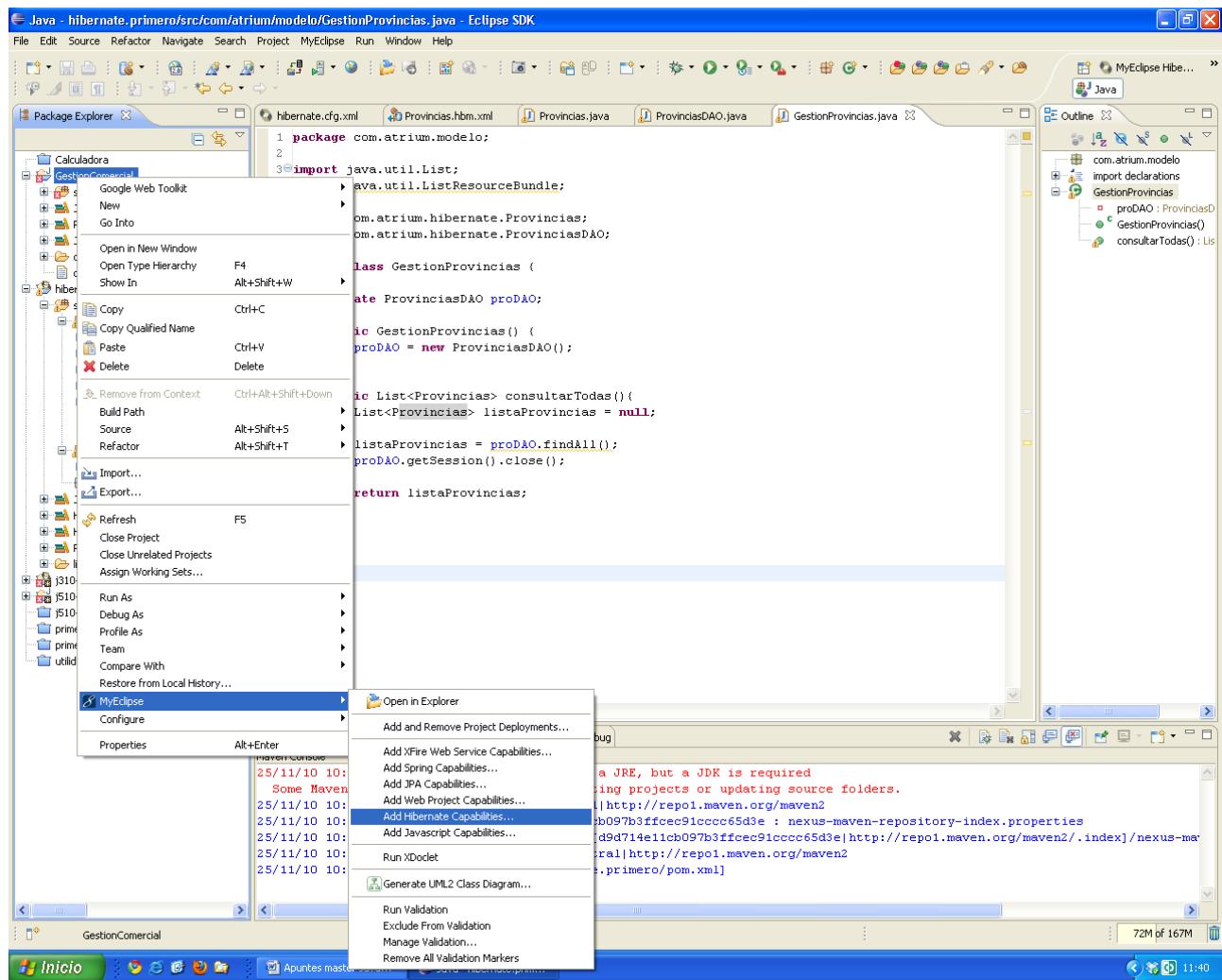
Ver Gestion_Municipios. Habrá que añadir al DAO el método consultar_PorProvincia

Instalación de Hibernate con MyEclipse

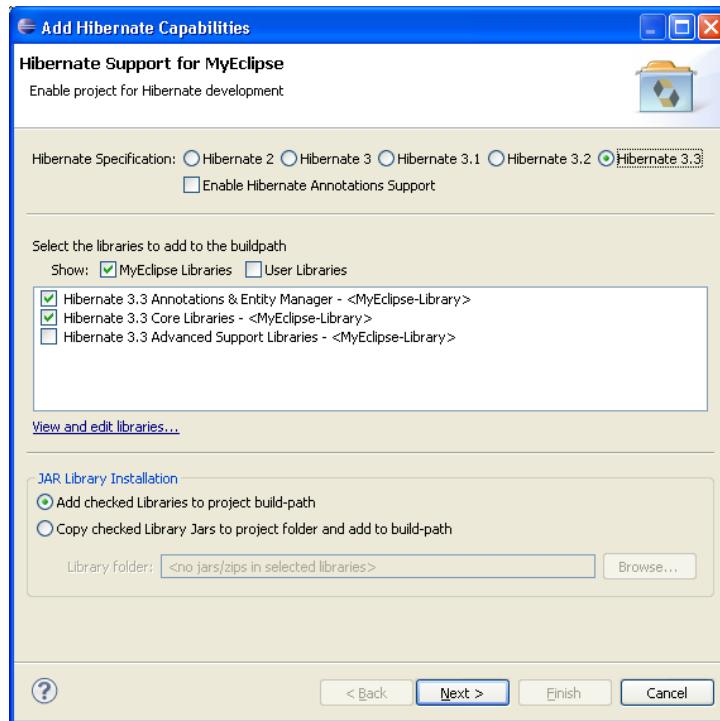
Como hemos dicho MyEclipse tiene un plugin de ingeniería inversa para Hibernate dentro del DataBase Explorer que se usa para se generen todos los archivos de configuración y otras clases de las que hemos estado hablando como clases de persistencia y DAOs.

El proceso de creación es el siguiente:

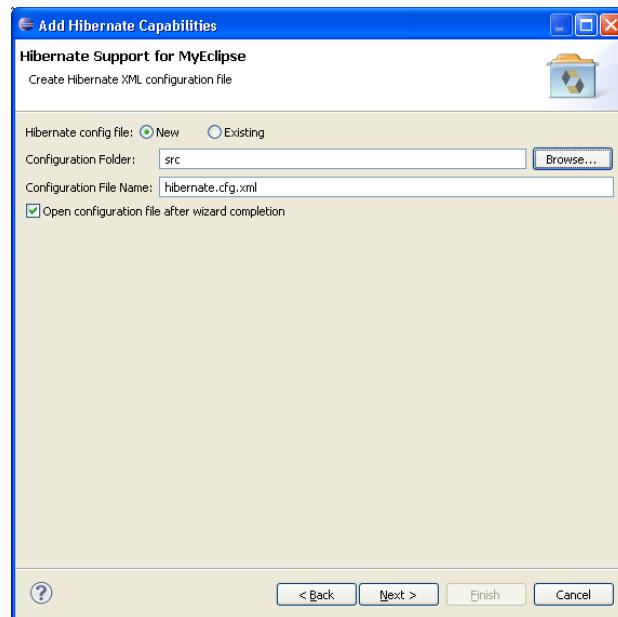
Teniendo nuestro proyecto creado con un paquete donde vayamos a tener nuestras clases de Hibernate y MyEclipse instalado, pulsamos con botón derecho sobre el nombre del proyecto, se abre un menú contextual, vamos a MyEclipse – Add Hibernate Capabilities...



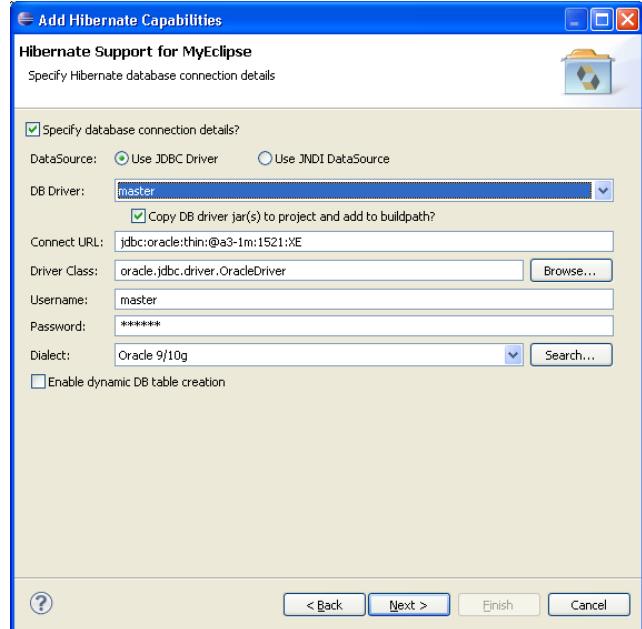
Se abre un asistente y lo dejamos con las opciones marcadas por defecto



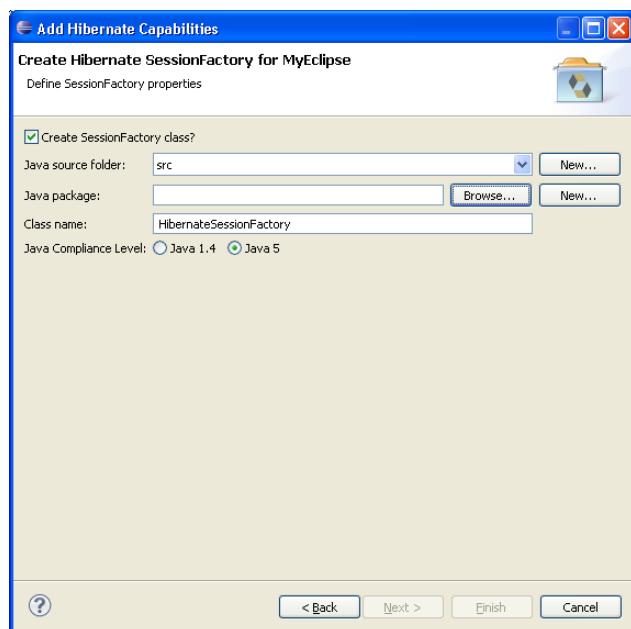
Pulsamos Next > y en Browse buscamos el paquete que hemos creado para las clases de hibernate



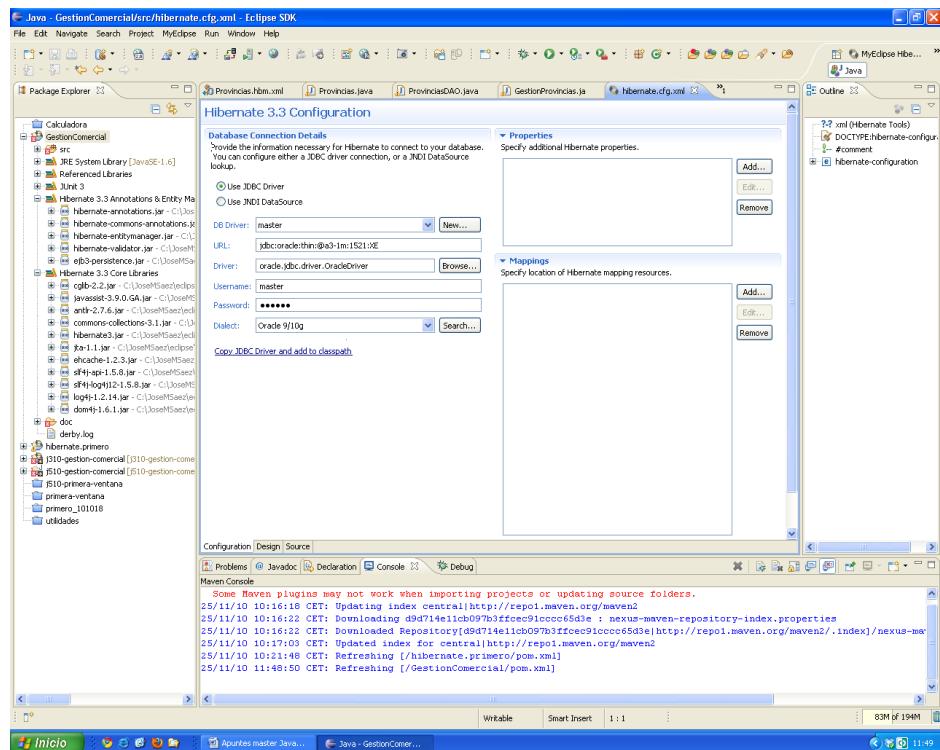
Al pulsar Next > se abre la ventana donde ponemos los datos de la conexión con la base de datos aunque probablemente en la pestaña de DB Driver encontraremos ya esos datos si hemos creado ya la conexión anteriormente.



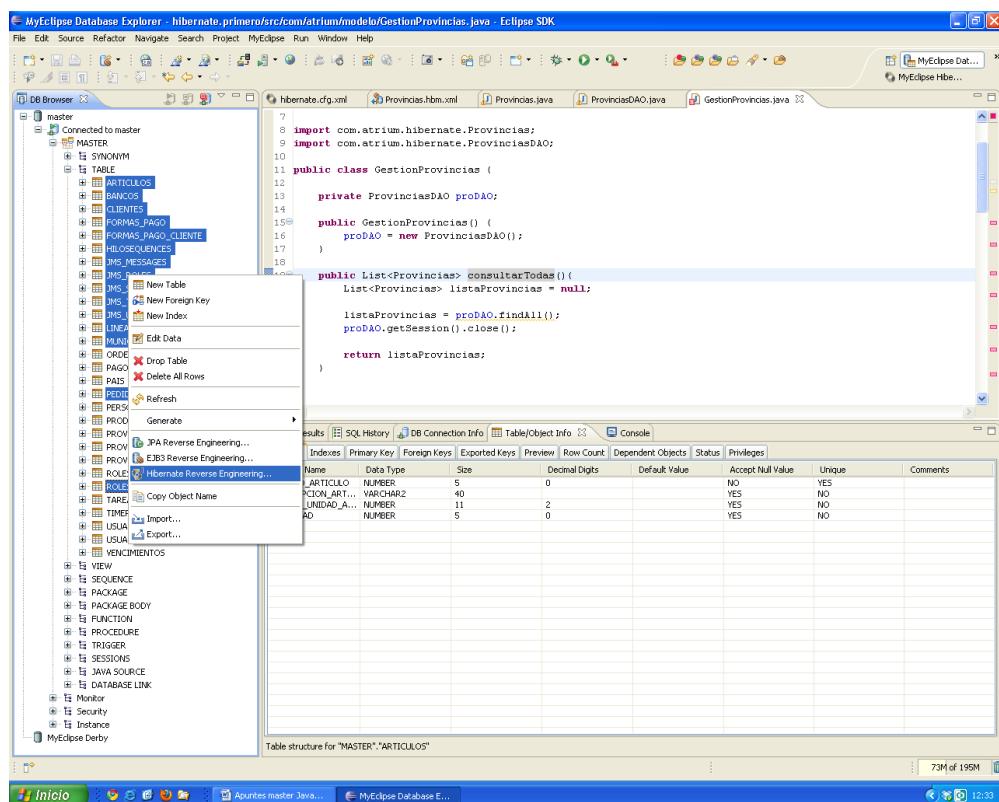
Pulsamos Next > después de tener los datos de la conexión correctos



En esta ventana vamos a decidir si queremos que se cree la clase SessionFactory que por supuesto será que sí, esta clase es similar al a que creamos nosotros para Conexión, pero de hibernate. Buscamos el paquete donde pondremos lo de hibernate pulsando el botón Browse de Java Package. Y pulsamos Finish. Se nos importa todo lo que necesitamos a los paquetes y referencias, además se crea el fichero descriptor con los detalles de la conexión.

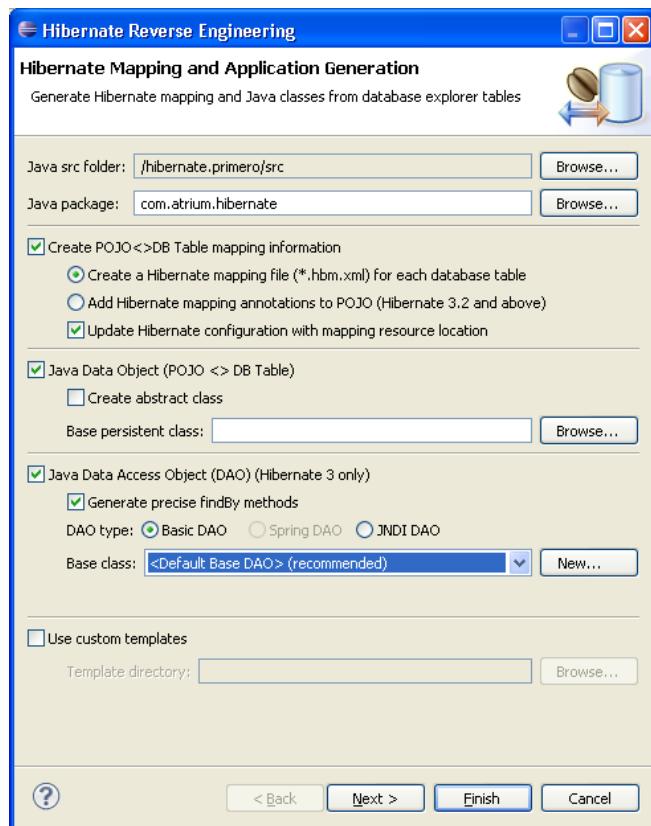


A partir de aquí **abrimos el plugin de Database explorer**, nos conectamos a la base de datos y elegimos las tablas que queremos que trabajen con hibernate

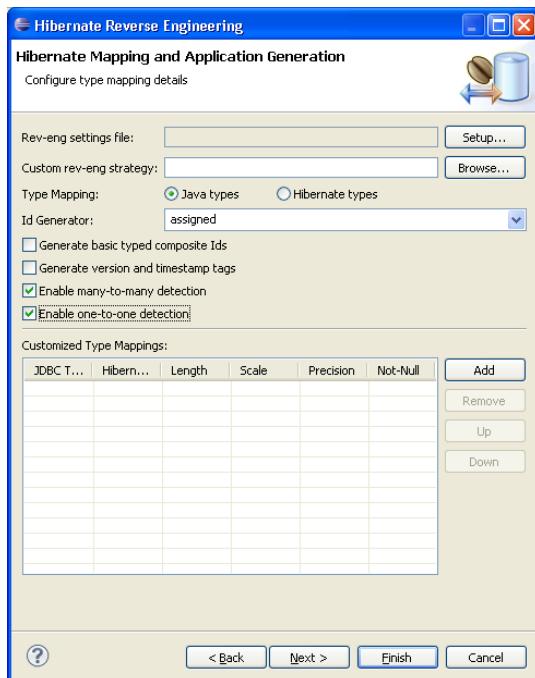


Pulsando con botón derecho sobre las clases seleccionadas sale un menú contextual donde elegimos **Hibernate Reverse Engineering...**

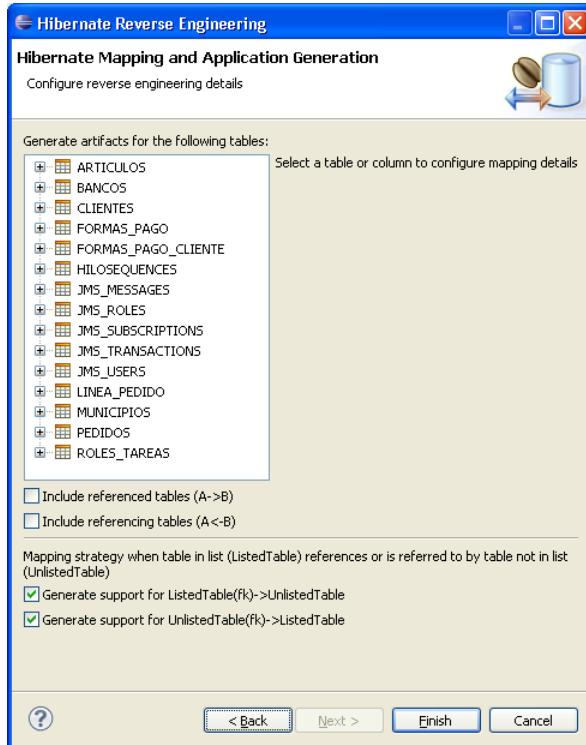
Aparece una ventana MUY IMPORTANTE para la generación del mapeo



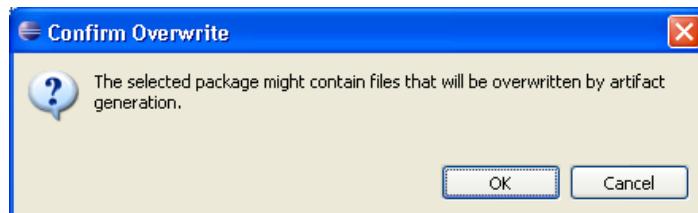
Buscamos el paquete donde ponemos las clases de Hibernate, marcamos las opciones necesarias tal como se ven en la ventana de arriba, y pulsamos Next >



En esta ventana elegimos **assigned** en Id Generator y **marcamos las dos casillas últimas**. Pulsamos Next >



Aparece una ventana donde podemos detallar los campos de cada tabla y así obtener lo que vamos a necesitar de la base de datos exclusivamente. Pulsamos Finish y nos pide confirmación.



Al pulsar OK se generan las clases de persistencia y los DAOs.

Solo nos queda para nosotros crear las clases de Fachada y algunas particularidades de algún DAO que pudiéramos necesitar.

J2EE

Java 2 Enterprise Edition (J2EE) es una especificación que se encuentra compuesta de varias partes. Esta formada por un número de tecnologías diferentes y una especificación que indica cómo deben trabajar estas tecnologías conjuntamente. Las partes principales del entorno J2EE son:

Componentes de aplicación

1. **Aplicaciones Clientes**: estos componentes son clientes gruesos implementados mediante aplicaciones java tradicionales. Acceden al servidor de aplicaciones utilizando el mecanismo conocido como Remote Method Invocation (RMI).
2. **Applets**: son clientes que suelen formar parte del interfaz de usuario gráfico y que se ejecutan dentro de un navegador web.
3. **Servlets y JavaServer Pages**: componentes que nos permiten generar aplicaciones web dinámicas y son los que vamos a tratar en este master.
4. **Enterprise JavaBeans (EJB)**: componentes que se ejecutan en un contenedor (contenedor EJB) en el servidor de aplicaciones que gestiona y trata gran parte de su funcionamiento interno.

Contenedores: cada tipo de componente se ejecuta dentro de un contenedor, este contenedor ofrece al componente una serie de servicios en tiempo de ejecución. Hay un tipo de contenedor para cada tipo de componente. Los contenedores más interesantes son los que permiten ejecutar servlets y páginas JSP, y el contendor EJB que gestiona los componenetes Enterprise JavaBeans (EJB).

Drivers gestores de recursos: es un driver que ofrece alguna clase de conectividad con un componente externo. Algunos de estos drivers son por ejemplo los APIs utilizados en JDBC, java messaging service (JMS) o JavaMail.

Bases de Datos: las bases de datos dentro de la plataforma J2EE son accesibles a través del API que ofrece el acceso a datos en Java, que no es otro que JDBC.

En la arquitectura cliente/servidor ambas partes son independientes.

Conceptos de comunicación web

HTTP->Hyper text Transfer Protocol. O **HTTPS** (Security). CORBA->protocol entre mainframes.

El protocolo de transferencia de [hipertexto](#) (**HTTP**, *HyperText Transfer Protocol*) es el [protocolo](#) usado en cada transacción de la Web ([WWW](#)). HTTP fue desarrollado por el consorcio [W3C](#) y la [IETF](#). HTTP define la sintaxis y la semántica que utilizan los elementos software de la arquitectura web (clientes, servidores, [proxies](#)) para comunicarse. Es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor. Al cliente que efectúa la petición

(un [navegador](#) o un [spider](#)) se lo conoce como "user agent" (agente del usuario). A la información transmitida se la llama recurso y se la identifica mediante un [URL](#). Los recursos pueden ser archivos, el resultado de la ejecución de un programa, una consulta a una [base de datos](#), la traducción automática de un documento, etc. HTTP es un protocolo sin estado, es decir, que no guarda ninguna información sobre conexiones anteriores. El desarrollo de aplicaciones web necesita frecuentemente mantener estado. Para esto se usan las [cookies](#), que es información que un servidor puede almacenar en el sistema cliente. Esto le permite a las aplicaciones web instituir la noción de "sesión", y también permite rastrear usuarios ya que las cookies pueden guardarse en el cliente por tiempo indeterminado.

URL significa *Uniform Resource Locator*, es decir, localizador uniforme de recurso. Es una secuencia de caracteres, de acuerdo a un formato estándar, que se usa para nombrar recursos, como documentos e imágenes en Internet, por su localización. El formato general de un URL es: *protocolo://máquina/directorio/archivo* También pueden añadirse otros datos: *protocolo://usuario:contraseña@máquina:puerto/directorio/archivo* Por ejemplo: <http://es.Wikipedia.org/>

Cliente pesado, cualquier programa que pueda comunicarse con mi servidor JAVA o NO. Tengo que cargar la JVM de Sun, si es JAVA o de la plataforma que sea, por ejemplo .NET. (Es decir tengo que plataformear la máquina).

Cliente ligero, cualquier navegador de internet. En el lado del cliente tiene que recibir lo que entiende el navegador, básicamente HTML.

Un [servidor](#) web se mantiene a la espera de *peticiones HTTP* por parte de un [cliente HTTP](#) que solemos conocer como [navegador](#). El cliente realiza una petición al servidor y éste le responde con el contenido que el cliente solicita. A modo de ejemplo, al teclear www.wikipedia.org en nuestro navegador, éste realiza una petición HTTP al servidor de dicha dirección. El servidor responde al cliente enviando el código HTML de la página; el cliente, una vez recibido el código, lo interpreta y lo exhibe en pantalla. Como vemos con este ejemplo, el cliente es el encargado de interpretar el código HTML, es decir, de mostrar las fuentes, los colores y la disposición de los textos y objetos de la página; el servidor tan sólo se limita a transferir el código de la página sin llevar a cabo ninguna interpretación de la misma.

HTML, siglas de **HyperText Markup Language** (*Lenguaje de Marcas de Hipertexto*), es el [lenguaje de marcado](#) predominante para la construcción de [páginas web](#). Es usado para describir la estructura y el contenido en forma de texto, así como para complementar el texto con objetos tales como imágenes. HTML se escribe en forma de "etiquetas", rodeadas por [corchetes angulares](#) (<,>). HTML también puede describir, hasta un cierto punto, la apariencia de un documento, y puede incluir un [script](#) (por ejemplo [Javascript](#)), el cual puede afectar el comportamiento de [navegadores web](#) y otros procesadores de HTML.

El **Document Object Model** (una traducción al español para nada literal, pero apropiada, podría ser **Modelo en Objetos para la representación de Documentos**), abreviado DOM, es esencialmente un modelo computacional a través de la cual los programas y [scripts](#) pueden acceder y modificar dinámicamente el contenido, estructura y estilo de los documentos [HTML](#) y [XML](#). Su objetivo es ofrecer un [modelo orientado a objetos](#) para el tratamiento y manipulación en tiempo real (o en forma dinámica) a la vez que de manera estática de [páginas de internet](#). El responsable del DOM es

el consorcio [W3C](#) (*World Wide Web Consortium*). En efecto, el DOM es una [API](#) para acceder, añadir y cambiar dinámicamente contenido estructurado en documentos con lenguajes como [ECMAScript \(Javascript\)](#).

CSS (Cascading Style Sheets). Las **hojas de estilo en cascada** son un lenguaje formal usado para definir la presentación de un documento estructurado escrito en [HTML](#) o [XML](#) (y por extensión en [XHTML](#)). El [W3C](#) (World Wide Web Consortium) es el encargado de formular la especificación de las [hojas de estilo](#) que servirán de estándar para los [agentes de usuario](#) o [navegadores](#). La idea que se encuentra detrás del desarrollo de CSS es separar la *estructura* de un documento de su *presentación*. Por ejemplo, el elemento de HTML <H1> indica que un bloque de texto es un encabezamiento y que es más importante que un bloque etiquetado como <H2>. Versiones más antiguas de HTML permitían atributos extra dentro de la etiqueta abierta para darle formato (como el color o el tamaño de [fuente](#)). No obstante, cada etiqueta <H1> debía disponer de la información si se deseaba un diseño consistente para una página, y además, una persona que lea esa página con un [navegador](#) pierde totalmente el control sobre la visualización del texto.

Lenguaje programación JavaScript → Manipula objetos
Document Object Model DOM → objeto implícitos creados por el nav.
Cascading Style Sheets CSS → hojas de estilo

Todas juntas forman el DHTML (DYNAMIC HTML) que dependen del navegador y de su versión. Es decir tendrás que programar para plataformas cruzadas.

AJAX, acrónimo de *Asynchronous JavaScript And XML* ([JavaScript](#) asíncrono y [XML](#)), es una técnica de desarrollo [web](#) para crear aplicaciones interactivas o [RIA](#) (Rich Internet Applications). Éstas se ejecutan en el [cliente](#), es decir, en el navegador de los usuarios y mantiene comunicación [asíncrona](#) con el servidor en segundo plano. De esta forma es posible realizar cambios sobre la misma página sin necesidad de recargarla. Esto significa aumentar la interactividad, velocidad y [usabilidad](#) en la aplicación. AJAX es una combinación de **cuatro** tecnologías ya existentes:

1. [XHTML](#) (o [HTML](#)) y [hojas de estilos en cascada](#) (CSS) para el diseño que acompaña a la información.
2. [Document Object Model](#) (DOM) accedido con un lenguaje de scripting por parte del usuario, especialmente implementaciones [ECMAScript](#) como [JavaScript](#) y [JScript](#), para mostrar e interactuar dinámicamente con la información presentada.
3. El objeto [XMLHttpRequest](#) para intercambiar datos asíncronicamente con el servidor web. En algunos [frameworks](#) y en algunas situaciones concretas, se usa un objeto [iframe](#) en lugar del XMLHttpRequest para realizar dichos intercambios.
4. [XML](#) es el [formato](#) usado comúnmente para la transferencia de vuelta al servidor, aunque cualquier formato puede funcionar, incluyendo HTML preformateado, texto plano, [JSON](#) y hasta [EBML](#).

Como el [DHTML](#), [LAMP](#) o [SPA](#), AJAX no constituye una tecnología en sí, sino que es un término que engloba a un grupo de éstas que trabajan conjuntamente.

PROTOCOLO HTTP

Tiene las siguientes restricciones:

1. Funciona a **ciclo petición/respuesta**. El usuario (CLIENTE) no puede hacer nada hasta que responda el SERVIDOR (seg..minutos). Para disimular esto tengo el AJAX, (para hacerle creer al usuario que no estoy esperando respuesta del servidor).
2. En el envío entre cliente/servidor sólo va texto.
3. HTTP es protocolo sin estado, todas las peticiones que se realicen son independientes unas de otras. Hay que programar para conservar el estado, que rol tiene, que usuario es).

Una vez se teclea la dirección de internet, (<http://www.----.com>), se entra en la aplicación. Toda aplicación J2EE, necesita un servidor WEB. Es decir, que para poder ejecutarlo necesitamos un **servidor WEB** (Programa JAVA, preparado para atender o escuchar todas las peticiones y servicios de un programa J2EE). Estos servidores web hacen la mayor parte del trabajo de guardar el estado.

SERVIDOR WEB

Un **servidor web** es un [programa](#) que implementa el [protocolo HTTP](#). Este protocolo está diseñado para transferir lo que llamamos [hipertextos](#), páginas web o páginas [HTML](#): textos complejos con enlaces, figuras, formularios, botones y objetos incrustados como animaciones o reproductores de música. Es un programa que se ejecuta continuamente en un ordenador (también se emplea el término para referirse al ordenador que lo ejecuta), manteniéndose a la espera de peticiones por parte de un cliente (un navegador de [Internet](#)) y que responde a estas peticiones adecuadamente, mediante una [página web](#) que se exhibirá en el navegador o mostrando el respectivo mensaje si se detectó algún error.

Existen muchos Servidores WEB en el mercado, (gratuitos y de pago). Todos funcionan de manera diferente.

COMERCIALES: WEBS SPHERE, WEBLOGIC
 LIBRES: APACHE, TOMCAT, JBOSS

El servidor puede estar ejecutando varias aplicaciones.

CONTEXTO

Cada aplicación es como una CAJA, dentro de la cual sólo tendrá visibilidad de aquellos objetos que estén en el mismo contexto (como la visibilidad de objetos de JAVA).

Contexto Servidor:

Un servicio a nivel de servidor tendrá visibilidad para todas las aplicaciones que se estén ejecutando en ese servidor. Por ejemplo DATA SOURCE, el pool se conecta con la Base de Datos. (Para facilitar la gestión de servicios existen tecnologías como SPRING, para servicios avanzados).

También puede haber objetos fuera de nuestro servidor, en otro contexto. Para esto tenemos API de JAVA, como:

1. **RMI (Java Remote Method Invocation)** es un mecanismo ofrecido en [Java](#) para invocar un [método](#) remotamente.

2. **JNDI** La **Interfaz de Nombrado y Directorio Java** es una [Interfaz de Programación de Aplicaciones](#) para servicios de [directorio](#). Esto permite a los clientes descubrir y buscar objetos y nombres a través de un nombre y, como todas las APIs de [Java](#) que hacen de interfaz con sistemas host, es independiente de la implementación subyacente. Adicionalmente, especifica una [interfaz de proveedor de servicio](#) (SPI) que permite que las implementaciones del servicio de directorio sean integradas en el framework. Las implementaciones pueden hacer uso de un servidor, un fichero, o una base de datos; la elección depende del vendedor.
3. **JTA** (del inglés **Java Transaction API - API para transacciones en Java**) es parte de **Java EE APIs**, JTA establece una serie de Interfaces java entre el manejador de transacciones y las partes involucradas en el sistema de transacciones [distribuidas](#): el servidor de aplicaciones, el manejador de recursos y las aplicaciones transaccionales.
4. Un **servicio web** (en inglés *Web service*) es un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en [redes de ordenadores](#) como [Internet](#). La [interoperaabilidad](#) se consigue mediante la adopción de [estándares abiertos](#). Tu pides un servicio y no importa en qué dirección está alojado. (Ejemplo: el tiempo para mañana).
5. Servicio Transversal, conecta dos aplicaciones para pasarse información.

Contexto aplicación:

Está dentro del contexto Servidor y es donde está la lógica de nuestro programa.

Contexto Sesión:

Está dentro del contexto aplicación. El servidor de aplicaciones decide a qué aplicación van las peticiones, esto lo hace automáticamente el servidor web. Si es la primera petición de ese cliente a esa aplicación entonces, abre un objeto que se llama SESIÓN. Si no es la primera, se la envía a la sesión abierta para ese cliente en el contexto de aplicación.

Cómo se sabe si la sesión está abierta o no. La primera vez que llega, no lleva id.de sesión, cuando el servidor responde le añade la id de sesión que acaba de abrir para ese cliente, después el cliente hará todas las peticiones con esa identificación. Es una clave única para cada sesión de 32 a 64 caracteres alfanuméricos.

Si esta sesión está mucho tiempo en memoria se puede optar por cerrarla temporalmente, serializándola, para después recuperarla, si hay una nueva petición; o si ha excedido el TIME OUT (por defecto 30 minutos), cerrándola definitivamente.

Es lo que utilizaremos para conservar la información entre petición y petición.

Contexto Request/Response

Están dentro del contexto sesión. Duran, lo que dure el tratamiento de la petición. Un usuario tendrá tantos contextos de petición y respuesta como peticiones haga y un solo contexto de sesión.

REQUEST->Información que manda el usuario en las peticiones.

RESPONSE->escribo la respuesta, se la mando al cliente, en HTML (lo que es capaz de interpretar).

HERRAMIENTAS

Para procesar la petición J2EE nos facilita las herramientas:

SERVLET (clases), JSP (etiquetas), JSTL (tags), UEL .

1. Los **servlets** son objetos que corren dentro del contexto de un [contenedor de servlets](#) (ej: [Tomcat](#)) y extienden su funcionalidad. También podrían correr dentro de un [servidor de aplicaciones](#) (ej: OC4J Oracle) que además de contenedor para servlet tendrá contenedor para objetos más avanzados como son los [EJB](#) (Tomcat sólo es un contenedor de servlets). La palabra *servlet* deriva de otra anterior, [applet](#), que se refería a pequeños programas escritos en Java que se ejecutan en el contexto de un navegador web. Por contraposición, un *servlet* es un programa que se ejecuta en un servidor. El uso más común de los *servlets* es generar páginas web de forma dinámica a partir de los parámetros de la petición que envíe el navegador web.
2. **JavaServer Pages (JSP)** es una tecnología [Java](#) que permite generar contenido dinámico para web, en forma de documentos [HTML](#), [XML](#) o de otro tipo. Esta tecnología es un desarrollo de la compañía [Sun Microsystems](#). La Especificación JSP 1.2 fue la primera que se liberó y en la actualidad está disponible la Especificación JSP 2.1. Las JSP's permiten la utilización de código Java mediante [scripts](#). Además es posible utilizar algunas acciones JSP predefinidas mediante etiquetas. Estas etiquetas pueden ser enriquecidas mediante la utilización de Librerías de Etiquetas ([TagLibs](#) o Tag Libraries) externas e incluso personalizadas.
3. La tecnología **JavaServer Pages Standard Tag Library (JSTL)** proporcionada por [Sun Microsystems](#) extiende las ya conocidas **JavaServer Pages (JSP)** proporcionando cuatro librerías de etiquetas (Tag Libraries) con utilidades ampliamente utilizadas en el desarrollo de páginas web dinámicas. Estas librerías de etiquetas extienden de la especificación de JSP (la cual a su vez extiende de la especificación de Servlet). Su API nos permite además desarrollar nuestras propias librerías de etiquetas.
4. **UEL->Unified Expression Language**

Además a parte hay especificaciones (Framework) JSF, STRUT.

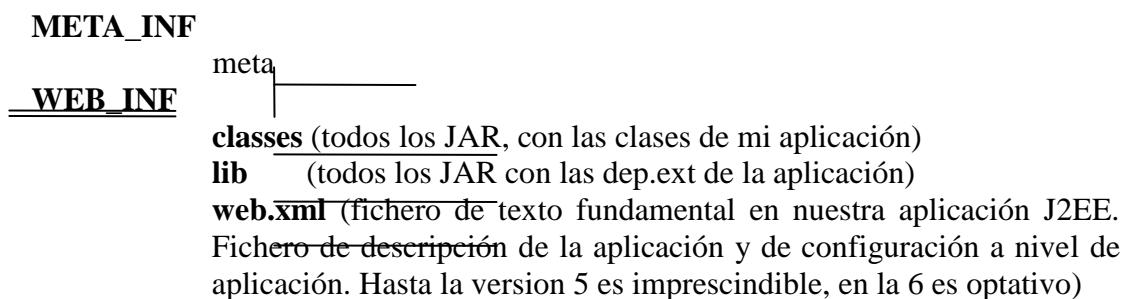
JavaServer Faces (JSF) es un [framework](#) para aplicaciones [Java](#) basadas en web que simplifica el desarrollo de [interfaces de usuario](#) en aplicaciones [Java EE](#). JSF usa [JavaServer Pages \(JSP\)](#) como la tecnología que permite hacer el despliegue de las páginas, pero también se puede acomodar a otras tecnologías como [XUL](#).

Struts es una herramienta de soporte para el desarrollo de [aplicaciones Web](#) bajo el [patrón MVC](#) bajo la plataforma [J2EE](#) (Java 2, Enterprise Edition). Struts se desarrollaba como parte del proyecto [Jakarta](#) de la [Apache Software Foundation](#), pero actualmente es un proyecto independiente conocido como Apache Struts.. Struts permite reducir el tiempo de desarrollo. Su carácter de "[software libre](#)" y su compatibilidad con todas las plataformas en que Java Enterprise esté disponible, lo convierte en una herramienta altamente disponible.

DESPLIEGUE DE UNA APLICACIÓN J2EE

Toda aplicación J2EE tiene que tener una estructura física mínima. Carpetas y directorios que siempre tienen que existir, de esa manera.

WEBROOT (Nombre del proyecto)



INTRODUCCIÓN AL SERVLET

Clase principal en J2EE, coge la petición y manda la respuesta. Tiene una subclase, que se llama **HTTPSERVLET** (para protocolo HTTP). El servidor lo instancia por mí, además llama a los métodos que necesita y los usa, es lo que se denomina:

CONCEPTO DE CICLO DE VIDA. (A partir de una instanciación de un objeto por parte del servidor de forma automática, este llama cuando le interesa a estos métodos. Yo tengo que saber como se llaman y cuando los llaman, para programar dentro del método lo que quiero que se haga en ese momento).

Método **Init()**, se llama una vez cuando se crea el SERVLET

Método **Desttroy()**, cuando se pone el objeto a disposición del recolector de basuras.

MVC-Sigue el patrón Modelo Vista Controlador, el controlador es el SERVLET porque recibe y envía, es lo que define el flujo.

De la clase HTTPSERVLET uso los métodos según el tipo de petición que le llegue, hay un método para cada tipo.

Fundamentalmente usaremos, **doGet()** Y **doPost()**.

doGet() ,la petición get cuando pulso un hipervínculo. Tiene un tamaño de max 4 k, que se pueden mandar, no es seguro porque se cachea, se refleja en la barra del navegador y como consecuencia se guarda en el historial. Dejando una puerta abierta para que entren en la aplicación.

doPost(), admite hasta 64K, no se refleja en la línea del navegador.

Reciben dos argumentos, **doGet(HttpServletRequest, HttpServletResponse){ }**
 doPost(HttpServletRequest, HttpServletResponse){ }

El http servlet lo instancia el servidor, entonces sabe que método instanciar, porque se lo indicamos en el fichero de configuración **web.xml**. Es decir le decimos de qué clase tiene que crear los servlet.

El servlet es el controlador y solo se usa para hacer logging.

FICHERO web.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
----- toda aplicacion debe contener al menos un servlet
<servlet>
    <description>This is the description of my J2EE component</description>
    <display-name>This is the display name of my J2EE component</display-name>
    <servlet-name>Primer_Servlet</servlet-name>
    <servlet-class>com.atrium.controlador.Primer_Servlet</servlet-class>
</servlet>
-----
<servlet-mapping>
    <servlet-name>Primer_Servlet</servlet-name>
    <url-pattern>/servlet/Primer_Servlet</url-pattern>
        Se pueden utilizar metacaracteres: /Texto /*.do /*.faces /*.iface
        (barra aquí, no en formulario)
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file> (página de inicio Bienvenida)
    </welcome-file-list>
</web-app>
```

El welcome-file-list. Cuando una aplicación recibe una petición en la que no se la pide nada, se le devuelve el welcome-file, donde va la ruta de la pagina de inicio y su nombre. La crea MyEclipse automáticamente.

Instalación de Tomcat en MyEclipse

1.-Vuelvo a instalar Eclipse, en otra carpeta

2.-Instalo server: El fichero de Tomcat lo descomprimo.(apache tomcat 6.0.13)

Opciones:

a.-full, http/1.1 ,puerto 8080 (si convive con oracle otro > 8086)

b.-User (admin) password (admin)

Luego paso el servicio a manual Panel control-Herramientas Administrativas-Servicios

3.-Plug in, de myeclipse donde está eclipse.

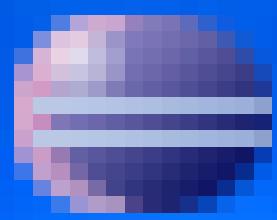
4.-Opciones de configuración del servidor:

Desde myeclipse, preferences, myeclipse, servers, tomcat, tomcat 6.x

X Enable, home directory (donde está el Tomcat) selecciono 6.0 y se rellenan el resto de opciones.

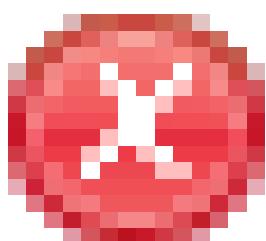
JDK Y ADD donde está el JDK, Browse, archivos de programa, jdk 1.6.0 tiene que salir muchas librerías).

Navigate. Search.



Tomcat

JRE Definition



The JRE

zf

j5

Top 5 Java Interview Questions

Desde myeclipse icono ordenador Tomcat 6.x y le doy a start.

Una vez en consola aparece info server start up (server levantado). Tengo que estar en la vista MyeclipseJavaEnterprise.

Como comienzo un proyecto:

-Selecciono new/web project

-Java EE5.0 finish. Entonces aparecen tres carpetas:

-web inf->web xml(source), index.jsp(plantilla)

New Package:

 New Servlet, 4 dejo arriba el resto de opciones lo deselecciono.

 V Generate map, para que me genere el código en web.xml.

Desplegar:

Icono deploy Myeclipse

 Selecciono el proyecto

 Add selecciono Tomcat 6.x

Después abro el browser, bola mundo.myeclipseweb.

Dirección: http://localhost:8080/nombreproyecto

Botón derecho ver código fuente.

Conexión Oracle:

Oracle thin driver

Jdbc:oracle:thin:@atrium3_1:1521:XE

Master(user)

Master(pass)

Odbc14.zip c:\master\oracle\odbc14.zip

Display de selected schemas. V Master (selecciona lo que interesa)

Usuarios Table/object info

Botón derecho accede a cosas de BBDD y selecciono DDL.

EJEMPLO “HOLA MUNDO”

Utilizamos la pagina index para redireccionar al componente que me interesa. En este caso al servlet.

Primer_Servlet.java

```
package com.atrium.controlador;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import sun.security.msccapi.RSACipher;
import com.atrium.modelo.Acceso_Usuarios;
import com.atrium.modelo.Conexion;

public class Primer_Servlet extends HttpServlet {
    public Primer_Servlet() {
        super();
    }
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        doPost(request, response);
    }
    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        Conexion con=new Conexion();
        Acceso_Usuarios ac=new Acceso_Usuarios(con.getCon());
        String sql= "select * from usuarios where nombre_usuario='"+request.getParameter("nombre_usuario")+
        "' and password='"+request.getParameter("password")+"'";
        ResultSet rs = ac.consultar_Usuarios(sql);
        try {
            if (rs.next()){
                //guardo los datos del usuario en la sesion
                HttpSession sesion=request.getSession();
                sesion.setAttribute("nombre_usuario", rs.getString("nombre_usuario"));
                sesion.setAttribute("password",rs.getString("password") );
                sesion.setAttribute("codigo_rol",new Integer(rs.getInt("codigo_rol")));
                //pasamos del servlet a otro componente, el dispatcher le cede el control a otro
                RequestDispatcher rqd=request.getRequestDispatcher("/jsp/usuario_correcto.jsp");
                rqd.forward(request, response);// se va al objeto que he creado salvo el request y response
            } else {
                //pasamos del servlet a otro componente, el dispatcher le cede el control a otro
                RequestDispatcher rqd=request.getRequestDispatcher("/jsp/usuario_incorrecto.jsp");
                rqd.forward(request, response);// se va al objeto que he creado salvo el request y response
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        try {
            con.getCon().close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%
String path = request.getContextPath();
String basePath =
request.getScheme() + "://" + request.getServerName() + ":" + request.getServerPort() + pa
th + "/";
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
```

```

<head>
  <base href="<%basePath%>">

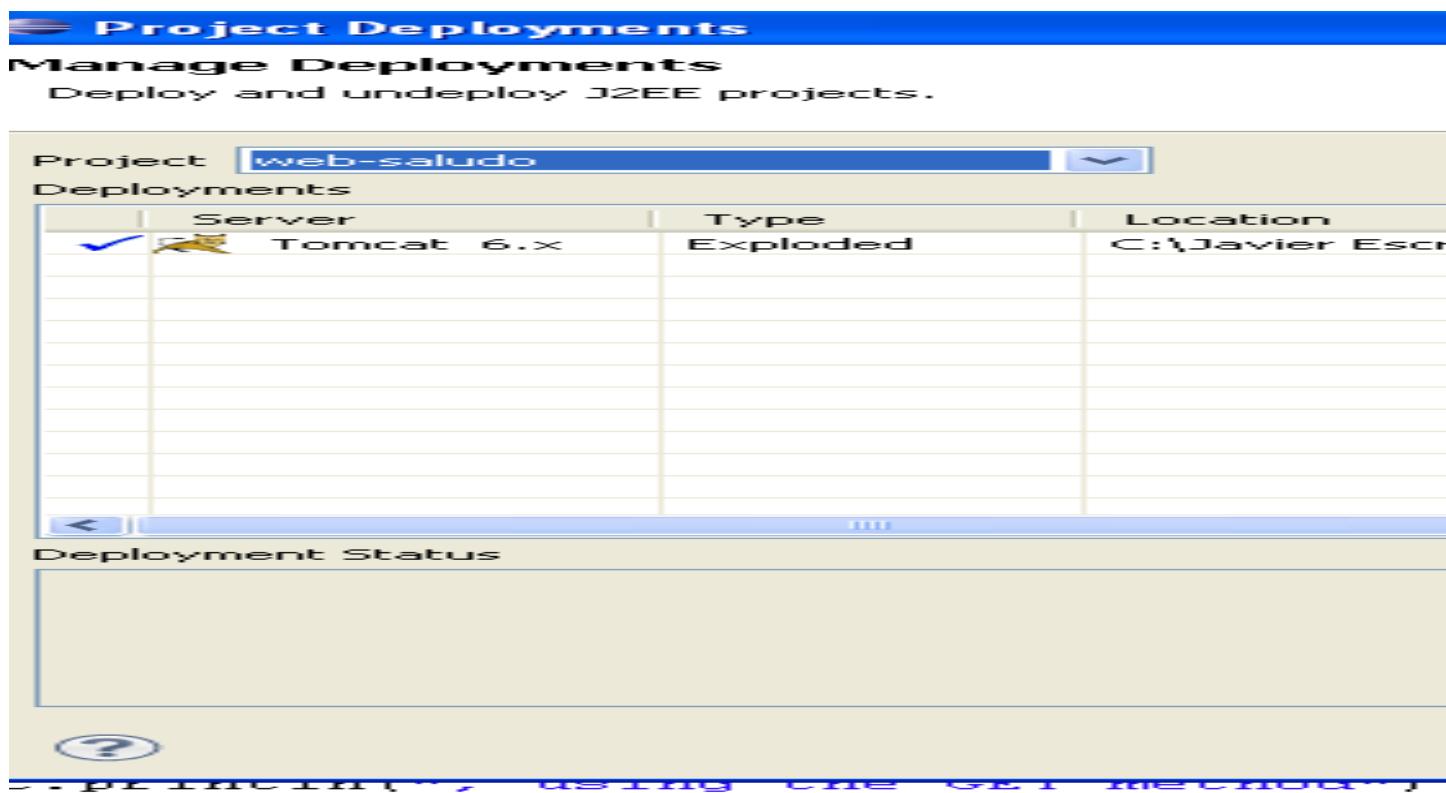
  <title>My JSP 'index.jsp' starting page</title>
  <meta http-equiv="pragma" content="no-cache">
  <meta http-equiv="cache-control" content="no-cache">
  <meta http-equiv="expires" content="0">
  <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
  <meta http-equiv="description" content="This is my page">
  <!--
  <link rel="stylesheet" type="text/css" href="styles.css">
  -->
</head>

<body>
  <jsp:forward page="/Saludo"></jsp:forward>
</body>
</html>

```

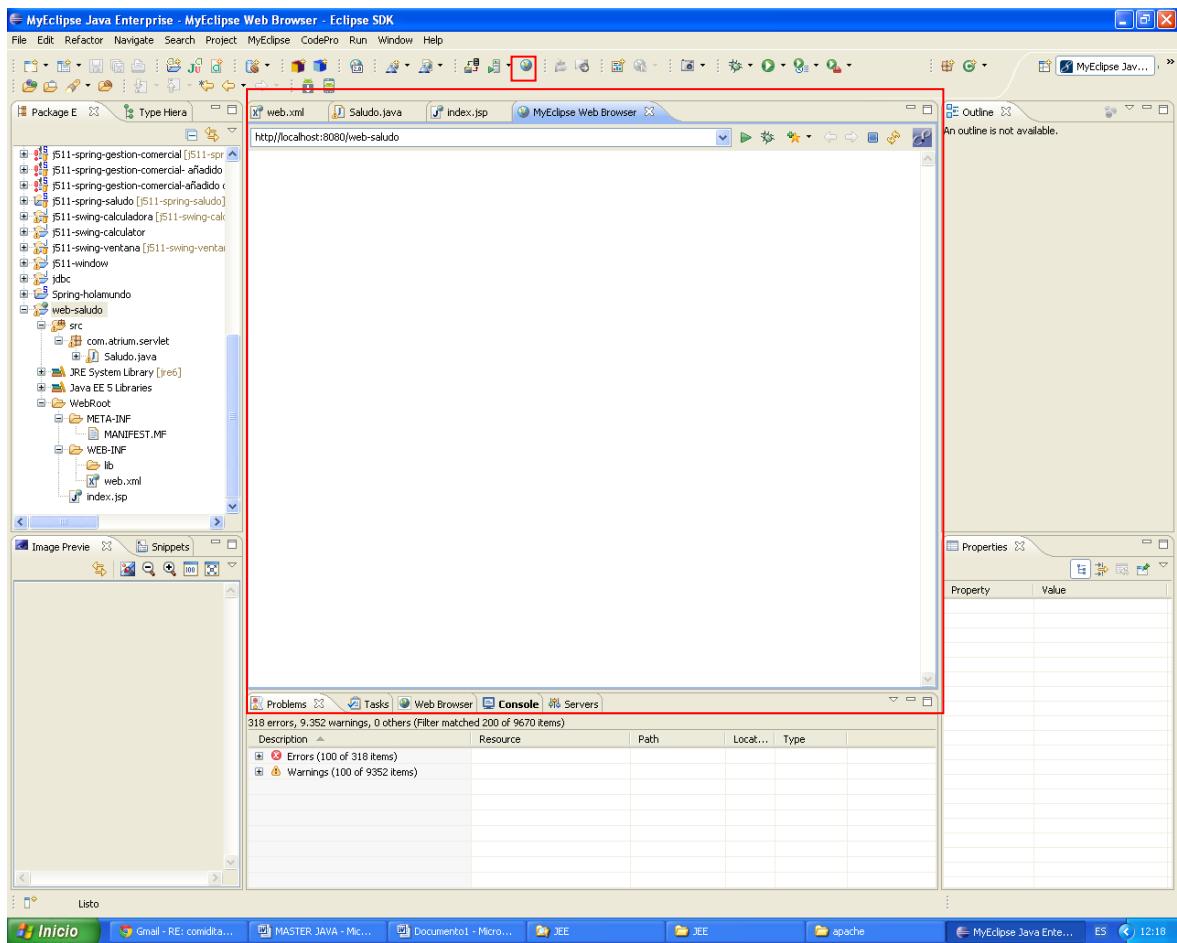
EJERCICIO DE LOGGEADO:

la primera vez tendremos que establecer el proyecto deployment



Hay q hacer un add y elegir tomcat 6.

Para probarlo tendremos que usar un navegador. Por ejemplo el de MyEclipse

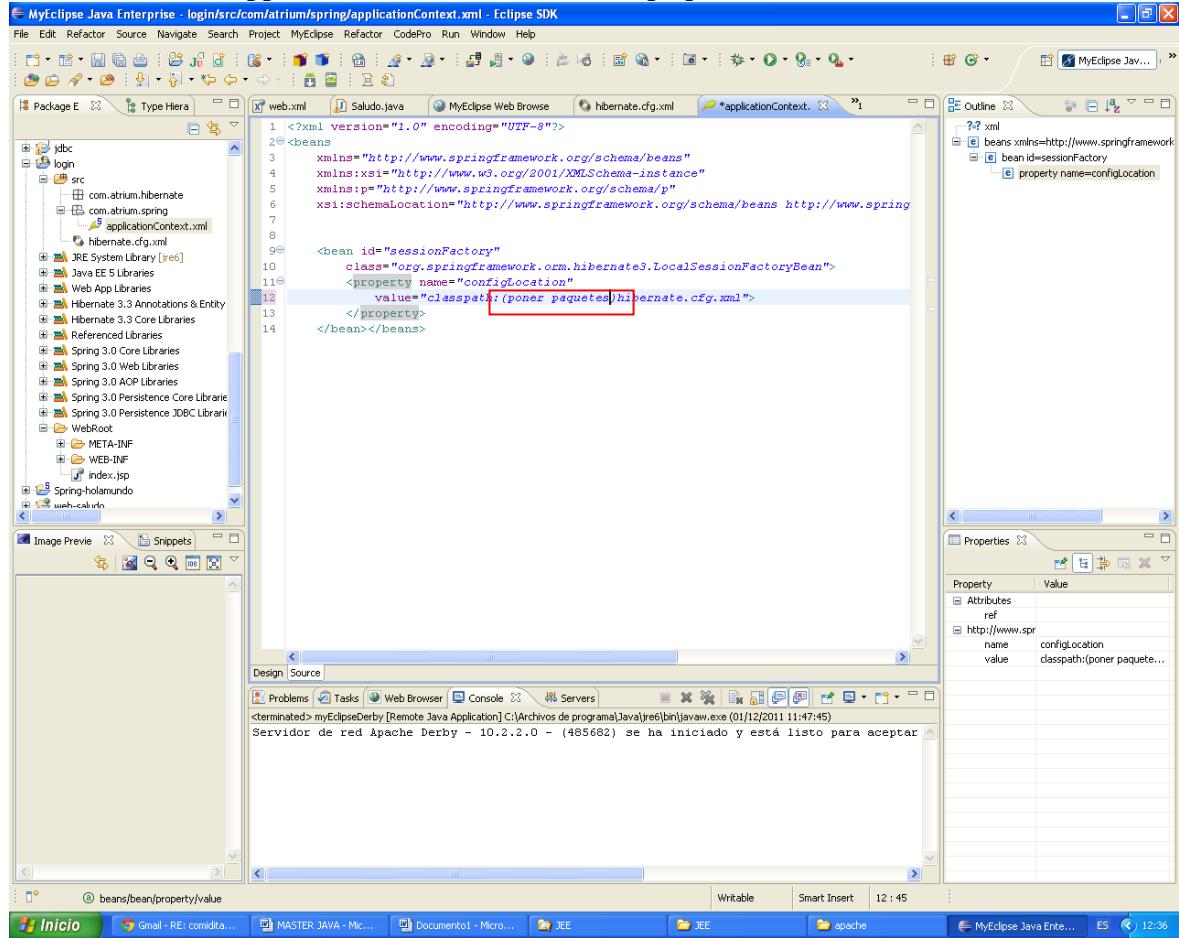


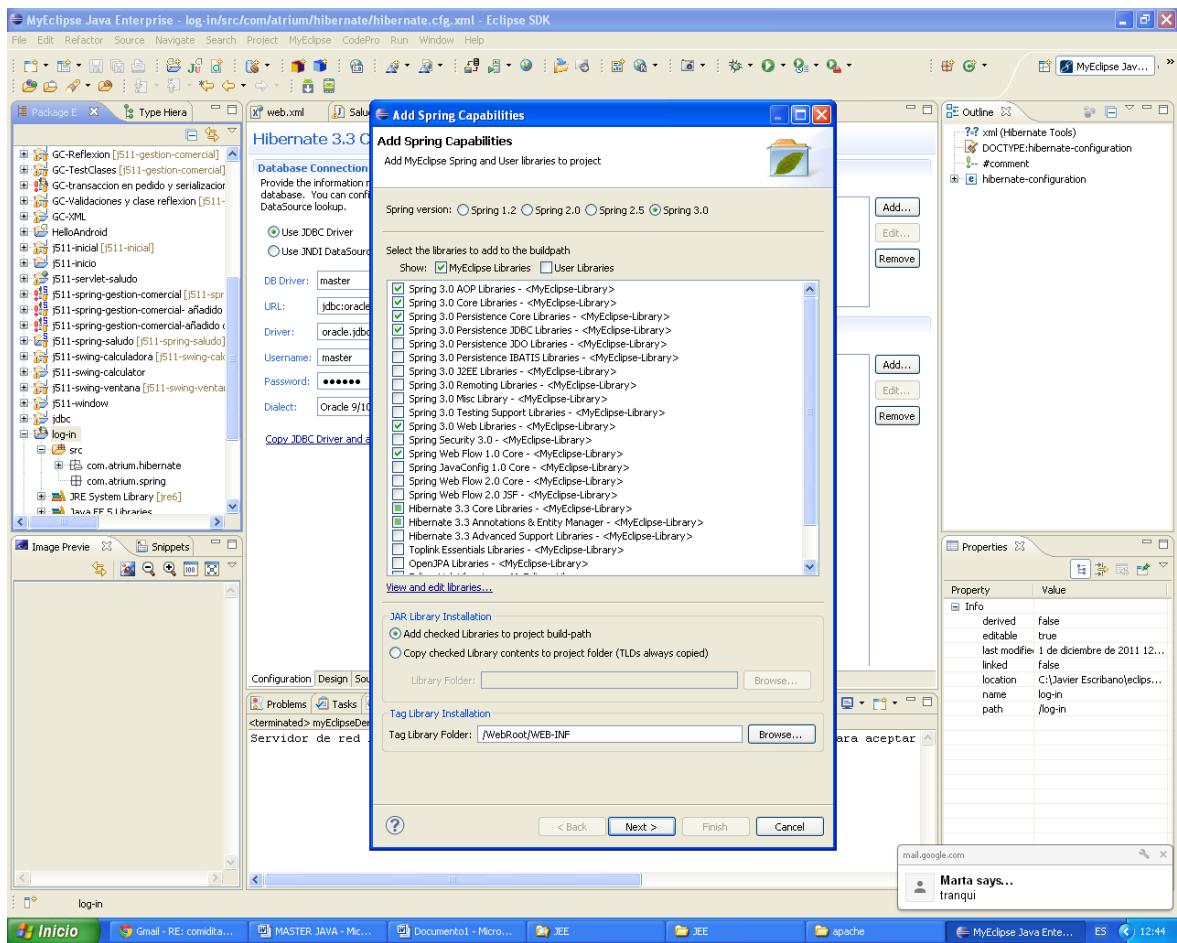
En la barra ponemos la dirección

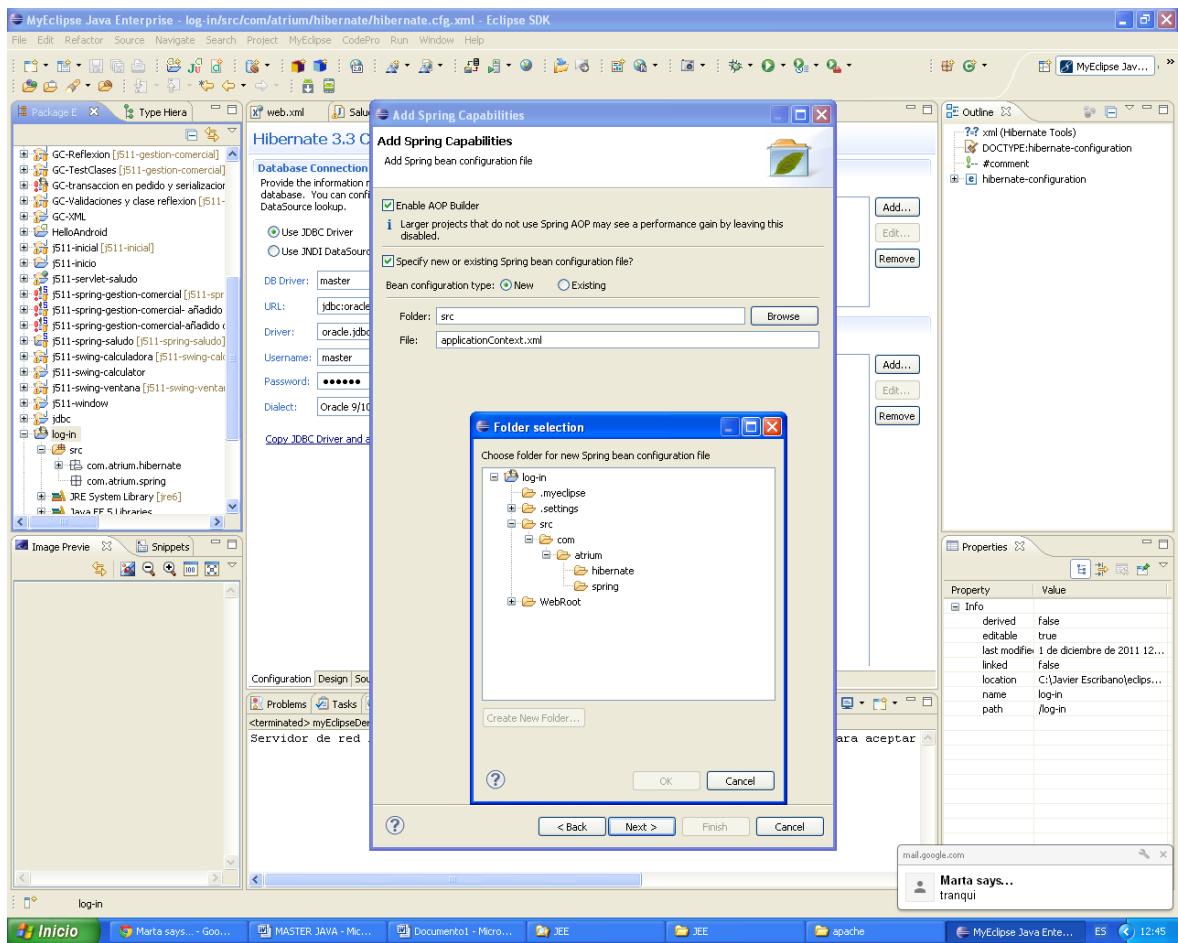
Ejemplo de login

Poner capacidad de hibernate antes que las de spring, sin el session factory pq eso lo gestionara spring

El asistente en el applicationContext no coloca el paquete





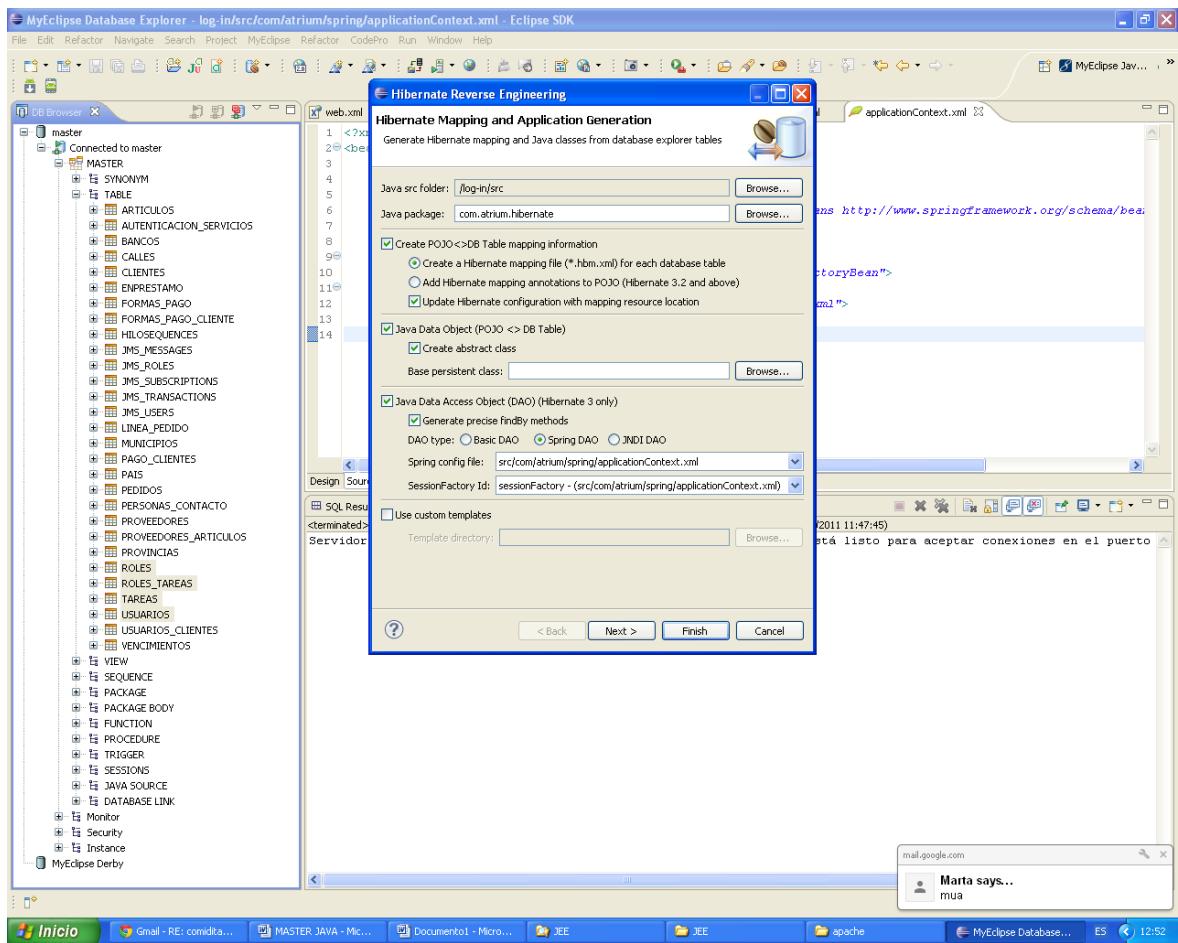


En el WebRoot tenemos que crear la carpeta JSP y dentro un new JSP con la opcion Default

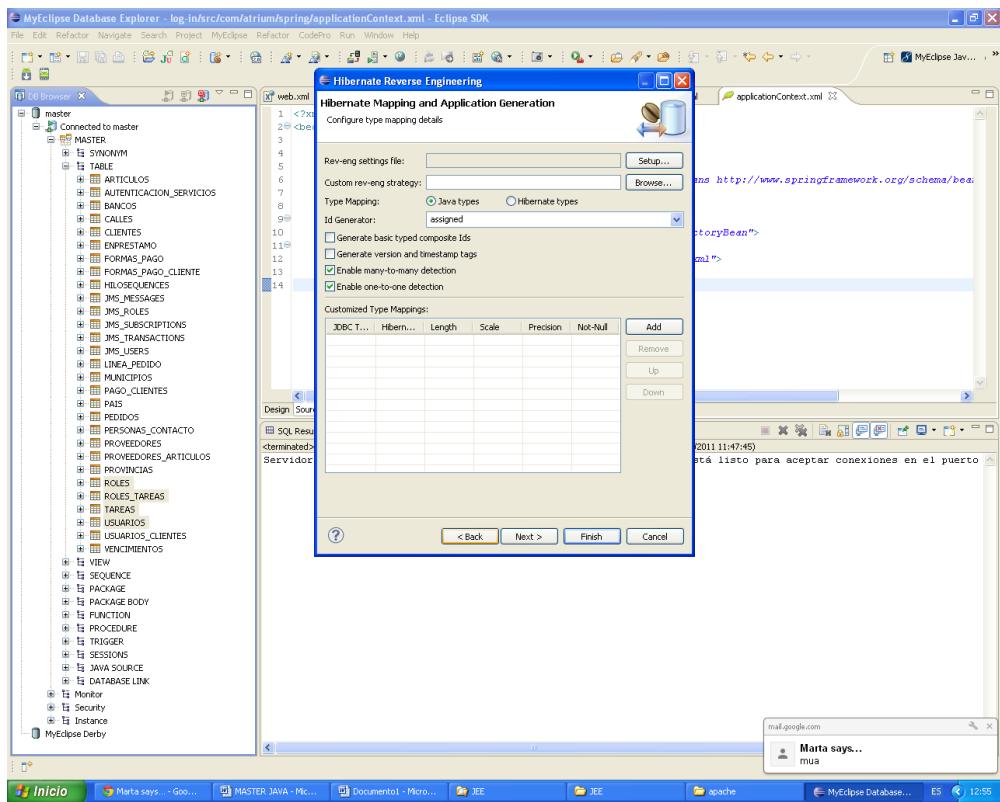


Nos creamos un formulario con un cuadro name otro password y un boton de confirmacion

Tenedremos que hacer ingenieria inversa para crear los componentes de spring o de hibernate. Para ello utilizamos las tablas roles , roles_tareas, tareas y usuarios



Aunque seleccionemos 4 tablas solo nos saca archivos de hibernate (xbm.xml) para tres por que una es una tabla de cruce y la gestiona el solo. Esto se consigue al establecer el reconocimiento de many-to-many



Y el valor assigned como idGenerator. Finish applicationContext

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation"
            value="classpath:com/atrium/hibernate/hibernate.cfg.xml">
        </property>
    </bean>

    <!--DAOS-->
    <bean id="UsuariosDAO" class="com.atrium.hibernate.UsuariosDAO">
        <property name="sessionFactory">
            <ref bean="sessionFactory" />
        </property>
    </bean>
    <bean id="RolesDAO" class="com.atrium.hibernate.RolesDAO">
        <property name="sessionFactory">
            <ref bean="sessionFactory" />
        </property>
    </bean>

```

```

<bean id="TareasDAO" class="com.atrium.hibernate.TareasDAO">
    <property name="sessionFactory">
        <ref bean="sessionFactory" />
    </property>
</bean></beans>

```

Hibernate.cfg.xml

Aquí tenemos el mapeado de las clases

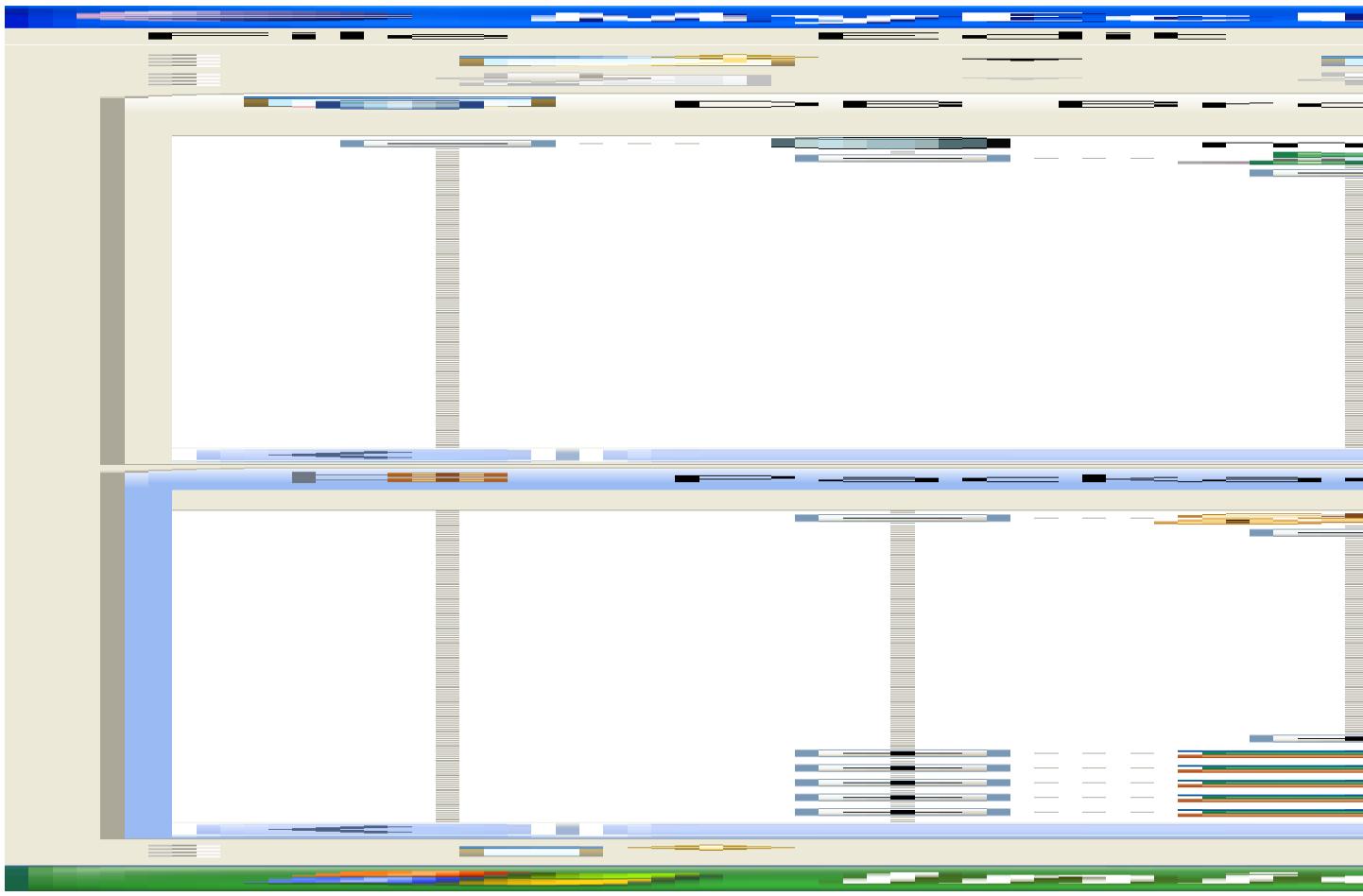
```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<!-- Generated by MyEclipse Hibernate Tools. -->
<hibernate-configuration>

    <session-factory>
        <property name="dialect">
            org.hibernate.dialect.Oracle9Dialect
        </property>
        <property name="connection.url">
            jdbc:oracle:thin:@a3-1m:1521:XE
        </property>
        <property name="connection.username">master</property>
        <property name="connection.password">master</property>
        <property name="connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="myeclipse.connection.profile">master</property>
        <mapping resource="com/atrium/hibernate/Usuarios.hbm.xml" />
        <mapping resource="com/atrium/hibernate/Roles.hbm.xml" />
        <mapping resource="com/atrium/hibernate/Tareas.hbm.xml" />
    </session-factory>
</hibernate-configuration>

```



En el paquete de hibernate tenemos todos los daos y archivos descriptores de las clases

Ahora tenemos que crearnos las interfaces de las fachadas para implementar spring

IGestion_Usuario

Publicamos los metodos que utilizamos en las calase que lo implementan

Gestion_Usuario.java

Comprobar_Credenciales

Por medio del objeto usu.DAO.finBy(name);

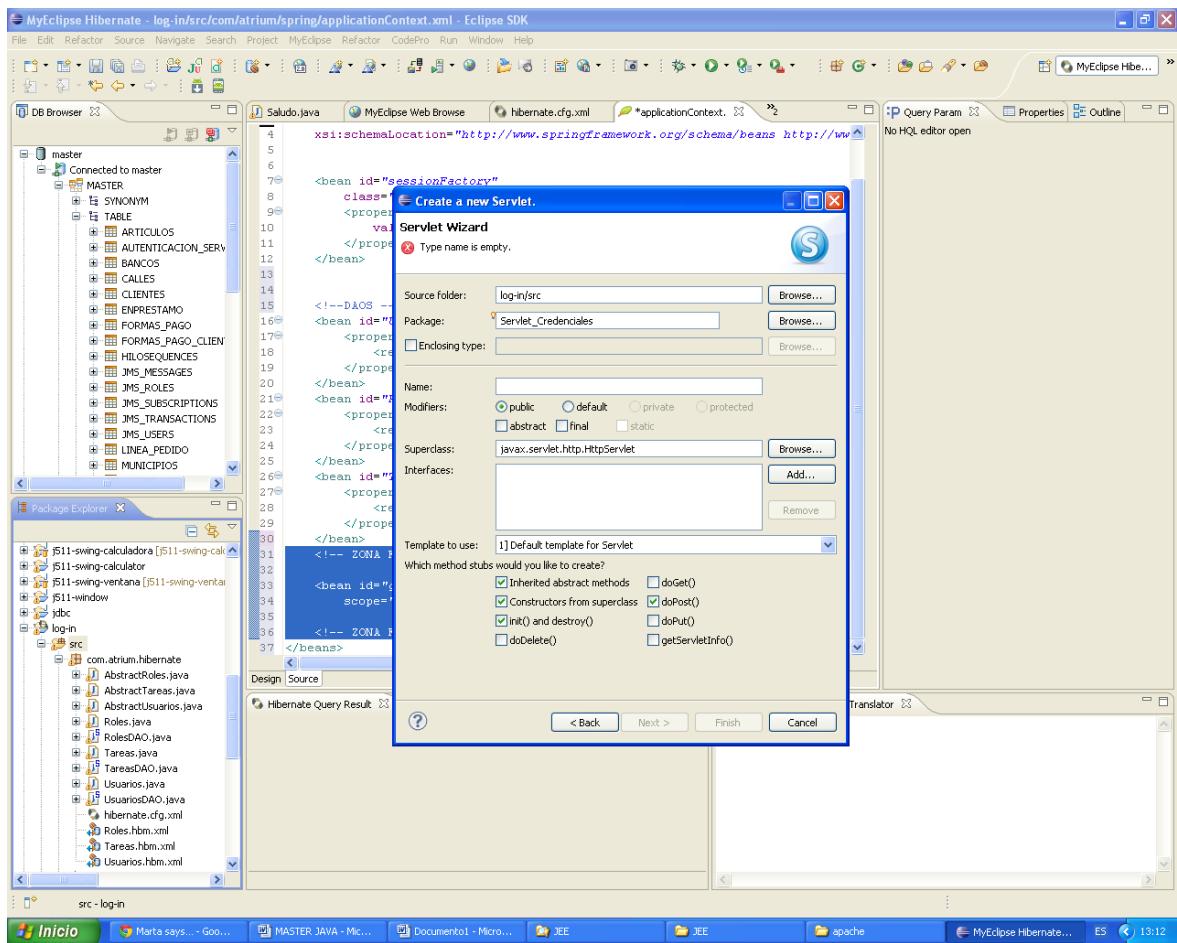
Tendremos que declararlo en el application context.

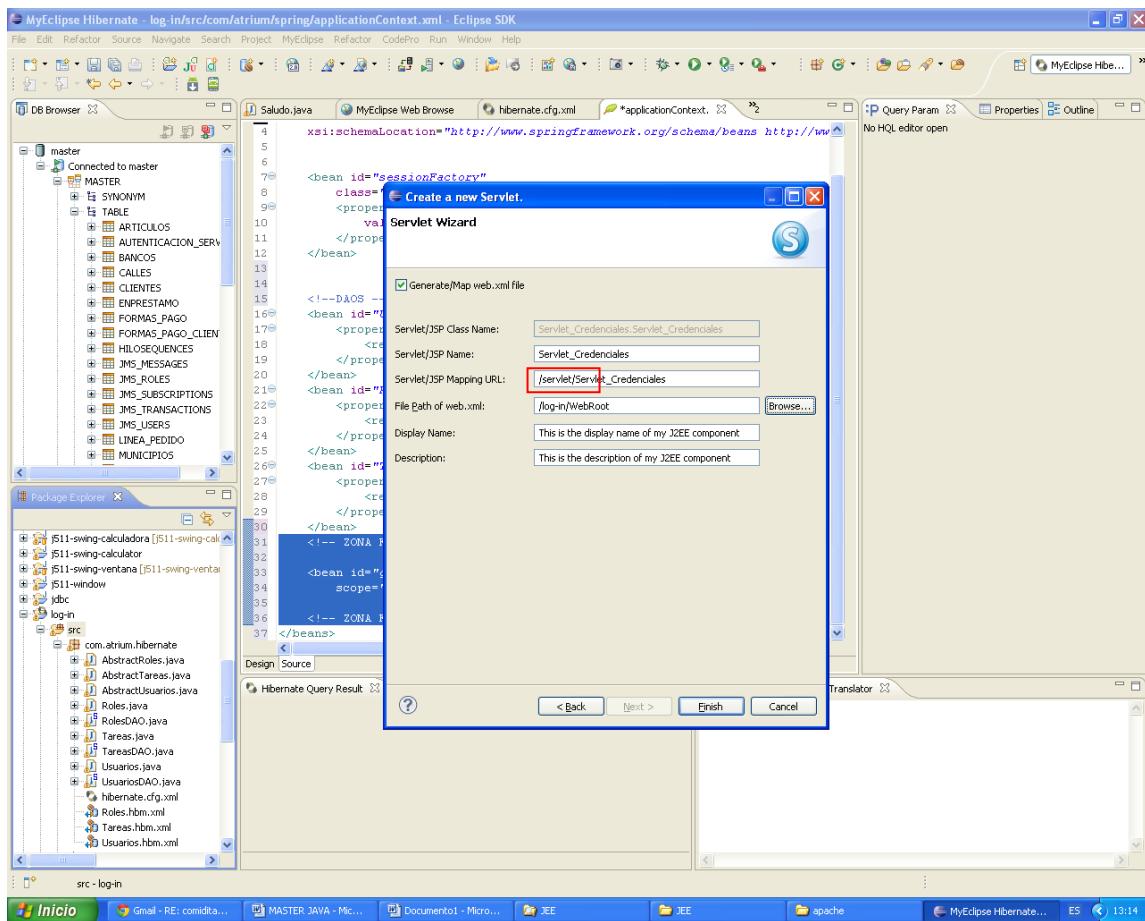
<!-- ZONA FACHADAS -->

```
<bean id="ges_usuario" class="com.atrium.modelo.Gestion_Usuarios"
      scope="prototype"/>
```

<!-- ZONA FACHADAS -->

Creamos un nuevo servlet





A partir de aquí es cuando la cosa cambia por ser una aplicación web. Tendremos que utilizar el servletContext. El applicationContext tiene que estar dentro de el. Con MyEclipse esto esta implementado con el paquete web app Libraries.

Colocamos el escuchador en el web.xml. (poner <!--CARGA EL APPLICATIONCONTEXT DE SPRING-->)

<listener></listener>

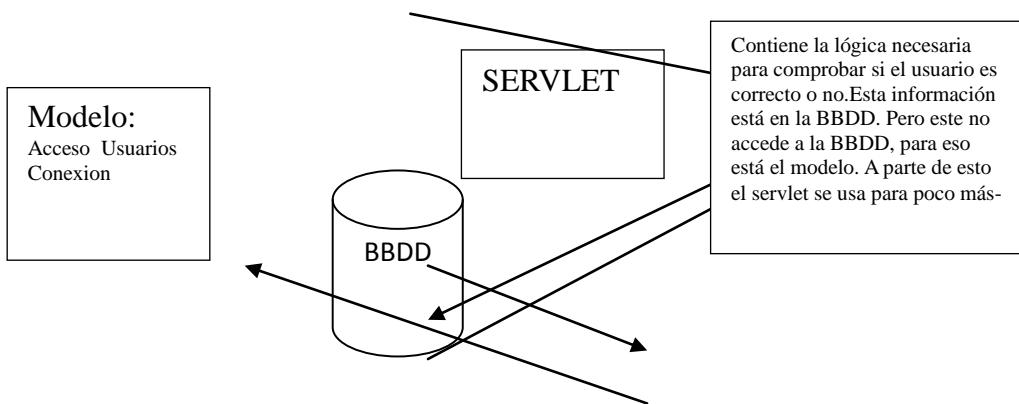
En el param-value colocamos la trayectoria. Puede ponerse mas de un valor separado por "," . Tambien es importante ver que se pone la ruta real de ejecución WEB-INF/classes/com/atrium/spring/applicationContext.xml

```

1 package Servlet_Credenciales;
2
3 import java.io.IOException;
4
5 public class Servlet_Credenciales extends HttpServlet {
6
7     /**
8      * Constructor of the object.
9      */
10    public Servlet_Credenciales() {
11        super();
12    }
13
14    /**
15     * Destruction of the servlet. <br>
16     */
17    public void destroy() {
18        super.destroy(); // Just puts "destroy" string in log
19        // Put your code here
20    }
21
22    /**
23     * The doPost method of the servlet. <br>
24     */
25    public void doPost(HttpServletRequest peticion,
26                      HttpServletResponse respuesta) throws ServletException, IOException {
27
28        String nombre_usuario = peticion.getParameter("nombre");
29        String clave = peticion.getParameter("clave");
30
31        // CONSULTA A BBDD
32        IGestion_Usuarios ges_usu = (IGestion_Usuarios) WebApplicationContextUtils
33            .getWebApplicationContext(peticion.getSession())
34            .getServletContext().getBean("ges_usu");
35
36    }
37
38    /**
39     * Initialization of the servlet. <br>
40     */
41    /**
42     * @throws ServletException
43     *         if an error occurs
44     */
45    public void init() throws ServletException {
46        // Put your code here
47    }
48
49 }

```

Conseguimos el applicationContext en el apartado //CONSULTAMOS BBDD



HttpSession sesión= request.getSession(); coge la sesión. Se cogen los parámetros de la sesión para guardarlos

Request Dispatcher-> para despacharlo en otro sitio.

Rqd.forward-> pasa a la siguiente .jsp., indicada en el dispatcher.

Contexto sesión-->httpsession, Cambio contexto ->HttpDispatcher, Contexto aplicación->HttpContext.

SOLUCION:

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

<servlet>
    <description>This is the description of my J2EE component</description>
    <display-name>This is the display name of my J2EE component</display-name>
    <servlet-name>Primer_Servlet</servlet-name>
    <servlet-class>com.atrium.controlador.Primer_Servlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>Primer_Servlet</servlet-name>
    <url-pattern>/servlet/Primer_Servlet</url-pattern>
</servlet-mapping>
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Primer_Servlet.java

```
package com.atrium.controlador;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import sun.security.msapi.RSACipher;
import com.atrium.modelo.Acceso_Usuarios;
import com.atrium.modelo.Conexion;

public class Primer_Servlet extends HttpServlet {
    public Primer_Servlet() {
        super();
    }
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        doPost(request, response);
    }
    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        Conexion con=new Conexion();
        Acceso_Usuarios ac=new Acceso_Usuarios(con.getCon());
        String sql= "select * from usuarios where nombre_usuario='"+request.getParameter("nombre_usuario")+
        "' and password='"+request.getParameter("password")+"'";
        ResultSet rs = ac.consultar_Usuarios(sql);
        try {
            if (rs.next()){
                //guardo los datos del usuario en la sesion
                HttpSession sesion=request.getSession();
                sesion.setAttribute("nombre_usuario", rs.getString("nombre_usuario"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```

sesion.setAttribute("password",rs.getString("password"));
sesion.setAttribute("codigo_rol",new Integer(rs.getInt("codigo_rol")));
//pasamos del servlet a otro componente, el dispatcher le cede el control a otro
RequestDispatcher rqd=request.getRequestDispatcher("/jsp/usuario_correcto.jsp");
rqd.forward(request, response);// se va al objeto que he creado salvo el request y response
}
else {
//pasamos del servlet a otro componente, el dispatcher le cede el control a otro
RequestDispatcher rqd=request.getRequestDispatcher("/jsp/usuario_incorrecto.jsp");
rqd.forward(request, response);// se va al objeto que he creado salvo el request y response
}
} catch (SQLException e) {
e.printStackTrace();
try {
con.getCon().close();
} catch (SQLException e) {
e.printStackTrace();
}
}
}
}

```

CONEXIÓN.JAVA.

```

package com.atrium.modelo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;

public class Conexion {
    private Connection con=null;
    private ResultSet rs=null;
    /**
     * @param args
     */
    public Conexion(){
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
        } catch (ClassNotFoundException e) {
            System.out.println("Error en la carga del driver");
        }
        try {
            con=DriverManager.getConnection("jdbc:oracle:thin:@atrium3_1:1521:XE","master", "master");
        } catch (SQLException e) {
            System.out.println("Error en la creacion de la conexion");
        }
    }
    public void Proceso(){
        Acceso_Usuarios ac=new Acceso_Usuarios(this.getCon());
        rs=ac.consultar_Usuarios("select * from usuarios");
        System.out.print("Nombre_usuario ");
        System.out.print("Password ");
        System.out.println("Codigo de Rol");
        try
        {
            while(rs.next()){
                System.out.print(rs.getString("nombre_usuario")+" ");
                System.out.print(rs.getString("password")+" ");
                System.out.println(rs.getInt("codigo_rol"));
            }
        }
    }
}

```

```

        }
    }catch(SQLException esql){ }
}
public Connection getCon() {
    return con;
}
public void setCon(Connection con) {
    this.con = con;
}
}

```

Acceso_Usuarios.java

```

package com.atrium.modelo;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Acceso_Usuarios {
    private Connection con=null;
    private Statement sta=null;
    private ResultSet rs=null;

    public Acceso_Usuarios(Connection con) {
        this.con=con;
        this.setCon(con);
        try {
            sta=con.createStatement();
        } catch (SQLException e) {
            System.out.println("Error en la creacion del statement");
        }
    }
    public ResultSet consultar_Usuarios(String sql){
        try {
            rs=sta.executeQuery(sql);
        } catch (SQLException e) {
            System.out.println("Error en la creacion del ResultSet");
        }
        return rs;
    }
    public Connection getCon() {
        return con;
    }
    public void setCon(Connection con) {
        this.con = con;
    }
    public ResultSet getRs() {
        return rs;
    }
    public void setRs(ResultSet rs) {
        this.rs = rs;
    }
}

```

INDEX.JSP

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%
String path = request.getContextPath();
String basePath = request.getScheme()+"//"+request.getServerName()+":"+request.getServerPort()+path+"/";
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <base href="<%=basePath%>">
        <title>My JSP 'index.jsp' starting page</title>
        <meta http-equiv="pragma" content="no-cache">
        <meta http-equiv="cache-control" content="no-cache">
        <meta http-equiv="expires" content="0">
        <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
        <meta http-equiv="description" content="This is my page">
        <!--
        <link rel="stylesheet" type="text/css" href="styles.css">
        -->
    </head>
    <body>
        <form action="servlet/Primer_Servlet" method="post">
            <table>
                <tr>
                    <td>Nombre:</td>
                    <td><input type="text" name="nombre_usuario" maxlength="10"></td>
                </tr>
                <tr>
                    <td> Clave:</td>
                    <td><input type="password" name="password"></td>
                </tr>
                <tr>
                    <td> <input type="submit" value="Log in" name="Login in">
                        <input type="reset" value="Cancel" name="cancel"></td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

Usuario_correcto.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%
String path = request.getContextPath();
String basePath = request.getScheme()+"//"+request.getServerName()+":"+request.getServerPort()+path+"/";
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <base href="<%=basePath%>">
        <title>My JSP 'usuario_correcto.jsp' starting page</title>
        <meta http-equiv="pragma" content="no-cache">
        <meta http-equiv="cache-control" content="no-cache">
        <meta http-equiv="expires" content="0">
        <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
        <meta http-equiv="description" content="This is my page">
        <!--
        <link rel="stylesheet" type="text/css" href="styles.css">
        -->
    </head>
    <body>
        usuario correcto. <br>
    </body>
</html>
```

GENERADORES Y ESCUCHADORES DE EVENTOS

A partir de la 2.3 se añadieron a los servlet, generadores y escuchadores de eventos. Para poder programarlos, las clases escuchadores, de hacen a partir de los interfaces listener, para ello tengo que saber que eventos se lanzan y como se llaman los escuchadores. Los lanzan los servlets.

Aplicación: (Contexto)

ServletContextListener (Interface)

- context**Initialized**(ServletContextEvent)
- context**Destroyed**(ServletContextEvent)
-

ServletContextAttributeListener

- Attribute**Added**(ServletContextAttributeEvent)
- Attribute**Replaced**(ServletContextAttributeEvent)
- Attribute**Removed**(ServletContextAttributeEvent)

Session: (Contexto)

HttpSessionListener (Interface)

- session**Created**(httpSessionEvent) sesión creada
- session**Destroyed**(httpSessionEvent) sesión cerrada
-

HttpSessionActivationListener

- session**willPasivate**(httpSessionActivateEvent) sesión a serializar
- session**didActivate**(httpSessionActivateEvent) sesión a desserializar
- httpSessionAttributeListener

- Attribute**Added**(HttpSessionBindingEvent)
- Attribute**Remove**(HttpSessionBindingEvent)
- Attribute**Replaced**(HttpSessionBindingEvent)

Los escuchadores se mapean en el XML. Con los escuchadores se sesión.

IMPLEMENTACION DE ESCUCHADORES

Eventos, es el mismo mecanismo que en J2SE, que genera evento, se manda al escuchador.

Para crear escuchador:

```
public class xxxx implements HttpSessionListener {}
```

Normalmente una clase por escuchador. Como le digo al web.xml, donde está el escuchador.

```
<web-app>
    <listener>
        <listener-class>com.atrium....</listener-class>
    </listener>
    <listener>
    </listener>
<web-app>
```

Máximo cinco escuchadores, una clase para cada escuchador. Puedo tener variables que estén disponibles desde que arranca la aplicación en el contexto de aplicación o servlet. Dos opciones,

- **valor fijo** para todos los Servlets(dentro de web-app, fuera de las demás). Es un valor fijo desde que arranca la aplicación, que lee el fichero una sola vez al arrancar la aplicación).

```
<context-param>
```

```

<param-name>Nombre_atributo</param-name>
<param-value>valor_atributo</param-value>
</context-param>
-valor variable para cada Servlet.
<servlet>
    <init-param>
        <param-name>Nombre_atributo</param-name>
        <param-value>valor_atributo</param-value>
    </init-param>
</servlet>

```

Estos atributos los obtengo a partir del objeto:

```

ServletContext xxx=this.getServletContext();
xxx.setAttribute(nombreobj, valor);
xxx.getAttribute(nombreobj);

```

Tanto Servlet, Session y Request, además de estos métodos tienen:

xxx.getAttributeNames(); me devuelve todos los atributos a la vez.

Me da una enumertion, colección. Para tratarlos. Para poder recorrerla tengo un objeto enumeration.

```

Enumeration enu = xxx.getAttributeNames();
while (enu.hasMoreElements())
{
    enu.nextElement();
}

```

El objeto request, tiene además de los métodos anteriores, métodos para acceder a la cabecera normalmente se trabaja con el cuerpo.

La información de la cabecera tiene lo de accesos no válidos, llego a ellos con:

request.getScheme()	<i>me devuelve el tipo de petición</i>
request.getRemoteAddr()	<i>devuelve la IP del cliente</i>
request.getRemoteHost()	<i>me da nombre completo del cliente(netBios)</i>
request.getServerName()	<i>nombre del host del servidor que recibe petición,+1</i>
request.getServerPort()	<i>puerto por donde vienen las peticiones</i>

Para obtener uno o todos a la vez:

Enumeration getHeaders(nombre_atributo); *un atributo con 1 o más valores.*

Ej: Char-Encoding: principal,alternativo,generico

Enumeration getHeadersNames(); *todos a la vez*

ATRIBUTOS Y PARAMETROS

Se estructuran en pares clave-valor.

-Los **parámetros** son los que vienen del cliente, los datos que rellenan en el formulario y nos llegan en el **request**, son texto String, porque en http sólo se puede mandar texto. Los parámetros sólo se pueden recoger y no existe **setParameter**.

-Los **atributos** son los que yo pongo en todo lo que no es request y son **Objetos**.

Parámetros nombres de campos:

```
String rq.getParameter("nombre_parametro");
```

→el nombre_parametro tal como está en el formulario.

Métodos para recuperar los nombres de los parámetros:

Enumeration rq.getParameterNames(); →todos los nombres de los parámetros.

Parámetros valores de campos:

String[] rq.getParameterValues(“nombre_parametro”);

→array de String, hay parámetros que pueden venir con varios valores.

Nombres y valores de los parámetros en un mapa:

Map rq.getParameterMap(); →mapas clave, valor

Además también puedo capturar los parámetros de una forma especial como podría ser:

Binary rq.getInputStream();
Buffered getReader();

Útil para casos muy determinados como pudiera ser recibir datos encriptados.

CONTEXTO DE SESION

Tiene persistencia mientras no se cancele o pase el TIME OUT. El objeto sesión me sirve para guardar todo objeto que necesite que tenga persistencia mientras dure la sesión. Tiene tres métodos,

sesion.setAttribute();
sesion.getAttribute();
sesion.removeAttribute();

J2EE es una especificación me dice como hacer algo, pero no para que hacerlo. Tendría que ir a una implementación para saber que hacer en función de lo queremos.

```
session.setAttribute(“nombre_atributo”, valor);
object ←sesion.getAttribute(“nombre_atributo”);
sesion.removeAttribute(“nombre_atributo”);
```

Genéricos:

Enumeration ← sesion.getAttributeNames(); devuelve todos los atributos

Otros métodos de sesión:

1 sesion.**Isnew();** →devuelve true si no se ha usado nunca el atributo.
2 sesion.**Invalidate();** →cancela la sesión

3 **HttpSession** sesion=request.getSession(false);

Instancia la sesión asociada, creándola si no existe cuando se le pasa true (por defecto si no se le pasa nada), pero en este caso si no existe la sesión no la crea. Este mecanismo es necesario para que nadie se salte el controlador donde se hace el log. Si le pregunto por la sesión asociada al request y es null, entonces le informo de que la sesión está caducada.

```
If (sesión==null)
{Sesión caducada}
```

```
4     Long fec_crea      = sesion.getCreationTime(); → dato en milisegundos de fecha creación  
5     Long fec_ult_acce   = sesion.lastAccessTime(); → idem de ultimo acceso
```

Casi todos los objetos se referencian a partir del **request**, desde aquí voy a la sesión y de ahí a la aplicación.

```
Sesion.getServletContext();
```

ejercicio

Las dependencias externas siempre en proyecto y en servidor de aplicaciones para que los reconozcan los dos. Añado log4j-1.2.15->jar file.

(appenders file <drcha> source le digo donde está el property en servlet init)
Generic Servlet (implement/override Methods).

```
Public void xxxx() throws ServletException{  
PropertyConfigurator.configure(this.getServletContext().getContextPath()  
+"/logs/log4j.properties");}
```

(trayectorias relativas porque en el despliegue de producción no sabemos donde están físicamente los recursos).

Convertir el log en un informe (convertir_log.java)

Hacemos una clase, coge la información que nos interesa del log. Luego ejecuta BIRT, clase ‘Ejecutar-birt.java’. Pero el Birt es una dependencia del servidor web y de la aplicación, se lo añado.

BIRT Runtime, Build Paths Add ext JARS, birt-runtime 2-2-0, pero lo tengo que descomprimir.Hay que coger com.ibm.icu, coreapi, engine api js.

[MyEclipse preference+ myeclipse+severs+Tomcat+Tomcat6.x+paths.

Ejecutar BIRT RCP DESIGNER:

Nuevo /informe en blanco/nombre que le damos y fuente de datos/ botón derecho nueva fuente de datos,(seleccionamos carpeta origen de datos de archivo plano) y ssv (;) Seleccionar carpeta donde está el finchero log.

Use first line column (la primera línea es el nombre de las columnas).

Conjunto de datos/ nuevo / next/ el filtro elijo *.* /tenemos que haber generado previamente nuestro fichero log de salida. Una vez seleccionado el fichero de salida, seleccionamos los campos que queramos. (El fichero es el que hemos filtrado).

Hacemos el informe.

En el texto insertamos una etiqueta y le ponemos el nombre que queramos. Me genero una tabla y voy arrastrando los campos. (La tabla la cojo de palette) que se encuentra en conjunto de datos.

Primera línea de cabecera: nombre columnas

Segunda línea es la fila de detalle.

En el proyecto myeclipse, me creo una carpeta de informes que cuelga de webroot y ahí pego nuestro informe generado en el BIRT.

Creo una carpeta que cuelga del webroot= informes html

En ejecutar BIRT, cambiamos la ruta del BIRT Y buscamos en Report-engine y pegamos esta trayectoria cambiando las barritas. Tenemos que buscar dentro del BIRT, la ruta completa donde está la carpeta report engine.

En la segunda línea de diré donde quiero que escriba sus logs. En design donde está el fichero del informe y como se llama.Donde y como quiero que se llame el informe de salida.

Options.setOutputFilename("_____").

Servlets

Los servlets son clases Java que se ejecutan en un servidor de aplicación, para contestar a las peticiones de los clientes. Aunque no se encuentran limitados a un único protocolo de comunicaciones podemos decir que se utilizan únicamente con HTTP, por lo que el servidor de aplicaciones pasa a denominarse servidor web.

Este tipo de servlets HTTP son los que vamos a tratar en este master que como hemos dicho, en la práctica son los únicos que se utilizan.

La especificación servlet 2.2 la encontramos implementada en dos paquetes que forman parte de la plataforma J2EE. El paquete **javax.servlet**, que define el marco básico que se corresponde con los servlets genéricos y el paquete **javax.servlet.http** que contiene las extensiones que se realizan a los servlets genéricos necesarias para los servlets HTTP.

Los servlets pueden utilizar todos los APIs de Java, además tienen las mismas características que define Java, y por lo tanto son una solución idónea para construir aplicaciones web de forma independiente del servidor y del sistema operativo.

El servidor debe cumplimentar el requisito de poder implementar la especificación Java Servlet. Hay varias opciones respecto a esto, el propio servidor Web puede implementar por si mismo La especificación Java Servlet, es decir, que además de ser servidor Web es contenedor de Servlets, o bien podemos instalar junto al servidor Web un añadido que funciona como un motor o contenedor de Servlets.

Un servlet también se define por los trabajos que realiza, estas tareas, en el orden lógico son:

Leer los datos enviados por el usuario: normalmente estos datos se indican a través de formularios HTML que se encuentran en páginas web.

Buscar otra información sobre la petición que se encuentra incluida en la petición HTTP: esta información incluye detalles como las capacidades y características del Navegador, cookies, nombre de la máquina cliente, etc.

Generar los resultados: este proceso puede requerir acceder a una base de datos usando JDBC, o generar la respuesta de manera directa.

Formatear los resultados en un documento: generalmente implica incluir los resultados en una página HTML.

Asignar los parámetros apropiados de la respuesta HTTP: implica indicar al navegador el tipo de documento que se le envía (por ejemplo HTML), asignar valores a cookies, etc.

Enviar el documento al cliente: el documento se puede enviar en formato de texto (HTML), formato binario (imágenes GIF), o incluso formato comprimido (ZIP).

Una de las cosas más interesantes de esta tecnología es la capacidad que tiene de generar páginas

web de forma dinámica.

Con los servlets, la máquina virtual de Java (JVM) trata cada petición como un hilo de ejecución (thead) ligero, no en un pesado proceso del sistema. Con los servlets existirá una única copia de la clase del servlet y varios hilos de ejecución. Además un servlet se mantiene en memoria entre distintas peticiones.

Los servlets se pueden comunicar de manera directa con el servidor web y pueden mantener la información entre varias peticiones simplificando las técnicas de mantenimiento de sesión.

Los servlets son independientes de la plataforma por lo tanto un servlet puede ejecutarse igual de bien sobre un servidor Apache o sobre un servidor IIS que soporte servlets..

Los servlets se ejecutan de forma segura dentro de un contenedor de servlets. Son baratos ya que existen contendores servlet totalmente gratuitos, como el servidor web Apache.

Explicaremos más adelante como instalar un contenedor Apache Tomcat en el entorno de trabajo de MyEclipse.

Estructura Básica de un Servlet

Un servlet va a ser invocado a través de una petición de un usuario desde un navegador web, esta petición puede ser realizada de varias formas, a través de un enlace o a través de un formulario HTML. En el primer caso utilizamos el método GET de HTTP, en el segundo caso indicaremos en el atributo METHOD de la etiqueta FORM de HTML el método de petición HTTP que utilizaremos que podrá ser GET o POST.

Para que una clase pueda ser considerada un Servlet y tratada como tal **debe heredar de la clase javax.servlet.http.HttpServlet**. La clase HttpServlet es una clase abstracta que representa a los servlets HTTP y de la que deben heredar todas las clases que deseen ser servlets. Esta clase ofrece una serie de métodos que son **doGet()**, **doDelete()**, y **doPost()** que se pueden sobreescribir en el servlet.

Lo más normal es que el servlet sobreescriba los métodos doGet(), o doPost() o ambos. Cada uno de estos métodos se ejecutar cada vez que el usuario envíe desde su navegador una petición HTTP mediante el método GET o POST respectivamente. Es muy común llamar a uno de estos métodos desde el otro doGet() o doPost(), de forma que el servlet se ejecute con independencia del tipo de petición que se le ha realizado.

Tanto el método doGet como el método doPost tienen **dos parámetros**: un objeto tipo **HttpServletRequest** y un objeto **HttpServletResponse**. Éstas son dos interfaces pertenecientes al paquete javax.servlet.http, que representan respectivamente una petición del protocolo HTTP y una respuesta del protocolo HTTP. Como podemos apreciar la relación entre los servlets y el protocolo HTTP no es casual.

El interfaz **HttpServletRequest** ofrece una serie de métodos mediante los cuales podemos

obtener información relativa a la petición HTTP realizada, como puede ser información incluida en los formularios, cabeceras de petición HTTP, el nombre de la máquina del cliente, etc. Se puede considerar que esta interfaz representa la entrada realizada por el usuario.

El interfaz **HTTPServletResponse** nos permite especificar información de respuesta a través del protocolo HTTP, esta información pueden ser cabeceras de respuesta del protocolo HTTP, códigos de estado, y lo más importante nos permite obtener un objeto **PrintWriter**, utilizado para enviar el contenido del documento al usuario, se limitan a ejecutar **sentencias println()** sobre el objeto PrintWriter para generar la página correspondiente. Se puede considerar que este interfaz representa la salida que se envía al usuario.

Los métodos doGet() y doPost() lanzan dos excepciones, **ServletException** e **IOException**.

Como mínimo, un servlet debe importar el paquete javax.servlet para poder tratar las excepciones ServletException y **el paquete javax.servlet.http**, para poder heredar de la clase HttpServlet y poder utilizar los objetos que representan la petición HTTP del usuario y la respuesta que se envía. **También** es muy común tener que importar **el paquete java.io** para usar el objeto PrintWriter de HTTPServletResponse.

En el paquete javax.servlet, correspondiente a los servlets más genéricos, podemos encontrar los siguientes elementos, que serán comentados más adelante:

Interfaces:

RequestDispatcher, Servlet, ServletConfig, ServletContext, ServletRequest, ServletResponse, SingleThreadModel.

Clases:

GenericServlet, ServletInputStream y ServletOutputStream.

Excepciones:

ServletException y UnavailableException.

En el paquete javax.servlet.http, correspondiente a los servlets más concretos, es decir a los servlets HTTP, encontramos estos elementos:

Interfaces:

HttpServletRequest, HttpServletResponse, HttpSession, HttpSessionBindingListener y HttpSessionContext.

Clases:

Cookie, HttpServlet, HttpSessionBindingEvent y HttpUtils.

Existen dos clases abstractas muy relacionadas con los servlets, la clases javax.servlet.http.**HttpServlet** que representa los servlets HTTP y de la que deberán heredar todos los servlets que se ejecuten en un servidor web, y por otro lado la clase javax.servlet.**GenericServlet**, que es la superclase de la clase HttpServlet y que representa a los servlets de forma genérica e independiente del protocolo utilizado por el cliente para realizar las peticiones al servlet. Veamos estas dos clases:

<u>HTTPServlet</u>	Como hemos dicho representa los servlets HTTP y todo servlet debe heredar de ella y sobreescribir al menos uno de sus métodos. Esta clase ofrece algunos métodos que se corresponden con los distintos tipos de peticiones, y además ofrece un método que tiene que ver con el ciclo de vida de un servlet, el ciclo de vida de un servlet lo veremos en detalle más adelante. Los métodos que se pueden sobreescribir en nuestros servlets, menos el método getLastModified lanzan dos excepciones: javax.servlet.ServletException y java.io.IOException.
void doDelete(HttpServletRequest req, HttpServletResponse res)	este método será invocado por el servidor (contenedor) a través del método service() de la clase HttpServlet, para permitir que el servlet trate una petición DELETE del protocolo HTTP. La operación DELETE permite a un usuario eliminar un documento o página web del servidor.
void doDelete(HttpServletRequest req, HttpServletResponse res)	este método será invocado por el servidor (contenedor) a través del método service() de la clase HttpServlet, para permitir que el servlet trate una petición DELETE del protocolo HTTP. La operación DELETE permite a un usuario eliminar un documento o página web del servidor.
void doGet(HttpServletRequest req, HttpServletResponse res)	este método será invocado por el servidor (contenedor) a través del método service() , para permitir que el servlet trate una petición GET .
void doOptions(HttpServletRequest req, HttpServletResponse res)	al igual que los anteriores este método será invocado por el servidor (contenedor) a través del método service() para permitir que el servlet trate una petición OPTIONS .
void doPost(HttpServletRequest req, HttpServletResponse res)	lo mismo para peticiones tipo POST .
void doPut(HttpServletRequest req, HttpServletResponse res)	lo mismo para peticiones tipo PUT . Esta petición PUT permite al usuario enviar un fichero al servidor de forma similar a como se haría usando el protocolo FTP.
void doTrace(HttpServletRequest req, HttpServletResponse res)	lo mismo para peticiones tipo TRACE , la cual devuelve las cabeceras enviadas junto con esta petición de vuelta al cliente, por lo que se usa para depuración, este método no suele sobreescibirse en los servlets.
long getLastModified(HttpServletRequest res)	devuelve en milisegundos la hora y fecha en la que el objeto HttpServletRequest se modificó por última vez.
void service(HttpServletRequest req, HttpServletResponse res)	recibe las peticiones estandar del protocolo HTTP y las redirecciona a los métodos doXXX adecuados definidos en la clase del servlet, se puede decir que es el punto común de entrada de todos los tipos de peticiones realizadas a un servlet. No es necesario ni recomendable sobreescibir este método. Es el método que tiene que ver con el ciclo de vida definido por los servlets.
void service(ServletRequest req, ServletResponse res)	este método redirige todas las peticiones que realizan los clientes al método service anterior. No hay ninguna necesidad de sobreescibir este método.

<u>GenericServlet</u>	Representa a los servlets de forma genérica y de forma independiente al protocolo utilizado para recibir y servir las peticiones de los clientes. Implementa las interfaces Servlet y ServletConfig, pertenecientes también al paquete javax.servlet. De esta clase hereda la clase vista antes HttpServlet, ya que debe acomodar las especificaciones de los servlets genéricos a los servlets HTTP. Esta clase ofrece una serie de métodos del ciclo de vida de los servlets, y otros sobre la configuración del servlet, además permite llevar un registro de la actividad del éste. No la usaremos nunca de manera directa, sino a través de la clase HttpServlet, que es clase hija de GenericServlet.
void destroy()	este método pertenece al ciclo de vida definido para los servlets y es llamado por el contenedor de servlets para indicar que el servlet va a finalizar su servicio, es decir, el servlet va a ser destruido.
String getInitParameter(String nombre):	devuelve un String que contiene el valor del parámetro de inicialización pasado por parámetro, o null si el parámetro no existe.
Enumeration getInitParameterNames()	devuelve el nombre de todos los parámetros de inicialización del servlet como un objeto java.util.Enumeration que contiene objetos String, uno por cada nombre del parámetro. Devolverá un objeto Enumeration vacío si el servlet no contiene parámetros de inicialización.
ServletConfig getServletConfig()	devuelve el objeto ServletConfig asociado al servlet. El objeto ServletConfig es un objeto de configuración utilizado por el contenedor de servlets para pasar información a un servlet durante su proceso de inicialización.
ServletContext getServletContext():	devuelve una referencia al objeto ServletContext en el que se está ejecutando el servlet. El interfaz ServletContext define un conjunto de métodos que el servlet puede utilizar para comunicarse con su contenedor de servlets, por ejemplo, para obtener el tipo MIME de un fichero, redirigir peticiones o escribir en un fichero de registro.
String getServletInfo():	devuelve información relativa al servlet como puede ser el autor, versión o copyright. Por defecto devuelve un String vacío, si queremos devolver información relevante hay que sobreescribir este método.
String getServletName()	devuelve el nombre de la instancia actual del servlet.
Void init(ServletConfig config)	perteneciente al ciclo de vida. Opuesto al método destroy(). Es invocado por el contenedor de servlets para indicar que el servlet se ha puesto en servicio. Recibe como parámetro un objeto ServletConfig que representa la configuración de inicialización que se va a aplicar a este servlet.
Void init()	es la segunda versión del método anterior, y se usa cuando el servlet no posee parámetros de configuración.
Void log(String mensaje)	escribe el mensaje pasado como parámetro en el fichero de registro (log) de los servlets, el mensaje va precedido por el nombre del servlet que lo ha generado. En el caso del servidor Jakarta Tomcat, este fichero de registro se llama SERVLET.LOG y se encuentra en el directorio c:\jakarta-tomcat\logs
Void log(String mensaje, Throwable t)	escribe un mensaje explicativo y un volcado de pila para una excepción Throwable especificada, en el fichero de registro correspondiente, precedido también por el nombre del servlet.
Void service(ServletRequest req, ServletResponse res)	método perteneciente al ciclo de vida de los servlets, y que ya conocemos de la clase HttpServlet, ya que en esta clase es donde está sobrescrito. Es invocado

	por el contendor de servlets para permitir que el servlet responda a una petición.

Ciclo de Vida de un Servlet

Cuando se crea un servlet, va a existir una única instancia de este servlet, que por cada petición que le realicen los usuarios del mismo creará un hilo de ejecución en el que se tratará el método doGet() o doPost() correspondiente.

Cuando el servlet es creado por primera vez se invoca el método init(), por lo tanto este método contendrá el código de inicialización del servlet.

Después de esto, cada petición realizada por un usuario sobre el servlet se traduce en un nuevo hilo de ejecución que realiza una llamada al método service(). Múltiples peticiones concurrentes normalmente generan múltiples hilos de ejecución que llaman de forma simultánea al método service(), aunque el servlet puede implementar un interfaz especial que permite que únicamente se pueda ejecutar un hilo de ejecución al mismo tiempo.

El método service() invocará los métodos doGet() o doPost() o cualquier otro método doXXX() dependiendo del tipo de petición HTTP recibida.

Finalmente, cuando el servidor decide descargar el servlet, se ejecuta anteriormente el método destroy().

Como se puede ver, el ciclo de vida de un servlet sigue el siguiente esquema:

Ejecución del método init() para inicialización del servlet.

Sucesivas ejecuciones del método service() en distintos hilos de ejecución, que resultan en una llamada al método doXXX.

Ejecución del método destroy() para realizar labores de liberación de recursos.

Visto en más detalle el ciclo de vida, podemos ver que el método init() es invocado cuando el servlet es creado y NO es llamado de nuevo con cada nueva petición.

El servlet puede ser creado cuando un usuario utiliza por primera vez el servlet a través de la URL o bien cuando se inicia la ejecución del servidor que contiene los servlets. La forma en que se crea el servlet depende de si el servlet se encuentra registrado en el servidor Web o no. Si no se encuentra registrado, el servlet se crea la primera vez que un usuario lo invoque, pero si se encuentra registrado en el servidor se creará cuando se inicie la ejecución del servidor.

El método init() ofrece dos versiones, como ya hemos visto, una de ellas no recibe ningún parámetro y la otra recibe un parámetro que es un objeto ServletConfig. La versión sin parámetros se usa cuando el servlet no necesita leer ninguna configuración que puede variar entre distintos servidores. Un ejemplo de código de un init sin parámetros es:

```
public void init() throws ServletException {  
    // código de inicialización  
}
```

La segunda versión del método init() se utiliza cuando el servlet necesita leer alguna configuración específica antes de inicializarse. Por ejemplo el servlet necesita información sobre la configuración de la base de datos, ficheros de contraseñas, parámetros específicos del servidor, etc. Un ejemplo sería:

```
public void init(ServletConfig config) throws ServletException {  
    super.init(config);  
    // código de inicialización  
}
```

Como vemos esta segunda versión de init() lo primero que hace es llamar al método init() de la clase padre pasándole como parámetro el mismo objeto ServletConfig. Esta sentencia siempre es necesaria ya que de esta forma la superclase también se inicializa de forma adecuada.

La interfaz ServletConfig ofrece un método llamado getInitParameter() que permite obtener el parámetro de inicialización del servlet cuyo nombre pasamos por parámetro como un String. En Tomcat podemos especificar los parámetros de inicialización de un servlet a través de un fichero XML llamado WEB.XML, que lo poseerá cada una de las aplicaciones Web del servidor.

Después de inicializado el servlet, cada vez que el servidor recibe una petición de un servlet, ya creado, el servidor crea un nuevo hilo y llama al método service() del servlet. El método service() comprueba el tipo de petición HTTP (GET, POST, PUT, DELETE, etc) y llama al método correspondiente.

No sobreescribir el método service() permite que se pueda añadir soporte para nuevos servicios más tarde añadiendo la implementación de los métodos doPut(), doTrace(), etc.

Por lo tanto los métodos doXXX son los que contienen todo el código que va a ejecutar el servlet.

En este punto hay que hacer una aclaración. Normalmente el servidor crea una única instancia del servlet, y múltiples hilos de ejecución según las peticiones de usuarios. Si se tienen múltiples peticiones concurrentes de un servlet, se tendrán varios hilos de ejecución concurrentes. Debido a esto los métodos doGet() y doPost() deben sincronizar el acceso a datos compartidos como pueden ser los atributos de un servlet, ya que múltiples hilos pueden acceder a los mismos datos de manera simultánea, para ello se debe hacer uso de la sentencia **synchronized** a la hora de acceder a objetos o información compartida.

Una solución alternativa es la de no permitir los múltiples hilos de ejecución (multithread), de esta forma nos aseguramos que en un mismo instante únicamente hay un hilo de ejecución accediendo a la instancia del servlet. Para ello la instancia de nuestro servlet debe implementar la interfaz javax.servlet.SingleThreadModel.

Si lo hacemos de esta segunda manera se crearan varias instancias del servlet pero cada una de ellas contendrá un único hilo en un mismo espacio de tiempo accediendo a los datos, se crea un pool de

instancias de la clase del servlet y las peticiones se irán sirviendo según se vayan quedando libres estas instancias. Las peticiones se irán encolando para esperar su turno, en el que puedan ejecutar una instancia libre del servlet.

Esta aproximación NO es recomendable, sobretodo para servlets que tienen peticiones muy frecuentes, ya que puede afectar muy seriamente el rendimiento del servidor.

Una vez vista la inicialización y la ejecución del servlet pasamos a ver la finalización. Un servidor puede decidir finalizar un servlet por varios motivos. Puede ser que se lo haya indicado directamente el administrador del servidor o porque el servlet lleva mucho tiempo inactivo. Antes de proceder a la descarga de la instancia del servlet, el contenedor de servlets invoca al método `destroy()` del servlet correspondiente.

Este método `destroy()` permite al servlet realizar tareas de liberación de recursos antes de ser destruido. En este método se suelen cerrar conexiones con bases de datos, finalizar hilos de ejecución, escribir cookies a discos o contadores, cerrar ficheros y otras tareas de limpieza similares.

<u>HttpServletRequest</u>	
	<p>Como vimos anteriormente los métodos <code>doXXX</code> de la clase <code>HttpServlet</code> reciben por parámetro objetos <code>HttpServletRequest</code>. Este interfaz nos permite obtener la información que envía el usuario al realizar una petición al servlet. Esta información puede ser muy variada, desde encabezados de petición del protocolo HTTP, cookies, o datos de un formulario.</p> <p>Más adelante cuando tratemos las páginas Java Server Pages (JSP) veremos que este interfaz se corresponde con objeto integrado en JSP llamado request.</p> <p>Este interfaz hereda del interfaz general <code>javax.servlet.ServletRequest</code> ofreciendo soporte para el protocolo HTTP.</p> <p>En esta lista faltan los métodos relacionados con los formularios HTML. Esto es así porque los métodos de acceso a la información de los formularios hereda de un interfaz genérico de los servlets, el interfaz <code>javax.servlet.ServletRequest</code>.</p> <p>El interfaz <code>ServletRequest</code> además de aportar los métodos que permiten acceder a la información de los formularios, ofrece otros métodos con variadas funciones, desde los que permiten obtener información acerca del servidor o del protocolo utilizado, hasta los que permiten almacenar y obtener atributos de la petición. Los veremos más adelante.</p>
<code>String getAuthType()</code>	devuelve el tipo de autenticación utilizado para proteger el servlet. Puede ser la cadena BASIC, SSL o null.
<code>String getContextPath()</code>	devuelve la porción de la URL de petición que indica el contexto de la misma. El contexto comienza con una barra “/”.
<code>Cookie[] getCookies()</code>	devuelve un array de objetos cookie que el cliente envió junto con su petición.
<code>long getDateHeader(String nombre)</code>	devuelve el valor de una cabecera de petición como un long que representa un

	objeto Date.
String getHeader(String nombre)	devuelve el valor de la cabecera de petición especificada.
Enumeration getHeaderNames()	devuelve todos los nombres de cabecera que contiene la petición en forma de objeto java.util.Enumeration.
Enumeration getHeaders(String nombre)	devuelve todos los valores de la cabecera de petición especificada.
int getIntHeader(String nombre):	devuelve el valor en forma de int de la cabecera que se indique por parámetro.
String getMethod()	devuelve el nombre del método del protocolo HTTP que se ha utilizado para realizar la petición, por ejemplo, GET, POST o PUT.
String getPathInfo()	devuelve cualquier información de camino extra asociada con la URL que utilizó el cliente para realizar la petición.
String getPathTranslated()	devuelve cualquier información de rutas extra después del nombre del servlet pero antes de la cadena de consulta (QueryString), y lo traduce a una ruta real.
String getQueryString()	devuelve la información de consulta contenida en la URL de petición, después del camino.
String getRemoteUser()	devuelve el login del usuario que ha realizado la petición, si el usuario se ha autenticado o null si no se ha autenticado.
String getRequestedSessionId()	devuelve el identificador de sesión indicado por el cliente.
String getRequestURI()	devuelve una parte de la URL de la petición realizada. Esta parte se corresponde con el contenido de la URL a partir del nombre del servidor y el número de puerto.
String getServletPath()	devuelve la ruta del servlet contenido en la URL de petición.
HttpSession getSession()	devuelve la session actual asociada con la petición, o si la petición no tiene una sesión se crea una.
HttpSession getSession(Boolean crear)	devuelve la session actual relacionada con la petición, si no existe una sesión y el parámetro crear tiene valor true, devolverá una nueva sesión.
Principal getUserPrincipal()	devuelve un objeto java.security.Principal que contiene el nombre del usuario autenticado actualmente.
Boolean isRequestedSessionIdFromCookie()	indica si el identificador de sesión pedido proviene de una cookie.
Boolean isRequestedSessionIdFromURL()	indica si el identificador de sesión pedido proviene de la URL de la petición.
Boolean isRequestedSessionIdFromValid()	indica si el identificador de sesión pedido es todavía válido.
boolean isUserInRole(String perfil)	indica si el usuario autenticado está incluido en el perfil lógico especificado.

Cabeceras de petición del protocolo HTTP

Hemos hablado en varias ocasiones de las cabeceras HTTP, estas cabeceras se utilizan para que el cliente (navegador web) realice una petición determinada de un recurso al servidor web. Estas cabeceras llevan una serie de información, que el servidor puede utilizar para devolver el resultado de la petición.

Podemos considerar que las cabeceras del protocolo HTTP es el lenguaje que utilizan el navegador y el servidor web para comunicarse entre sí.

El cliente o navegador Web envía al servidor Web una serie de cabeceras de petición HTTP para indicarle los recursos que está demandando, cómo puede el cliente aceptar los datos devueltos y donde se encuentran los datos adicionales que acompañan a la petición.

La cabecera que contiene el método completo de petición HTTP, es la primera cabecera de petición que se envía desde el navegador al servidor, también se le denomina línea de petición. Esta es una línea formada por tres elementos separados por espacios. La sintaxis general es la siguiente:

`Método_de_la_peticón URL HTTP_version \n`

El Método_de_la_peticón puede ser GET, POST, HEAD, PUT, DELETE, LINK y UNLINK.

URL: es la dirección o localización del fichero, programa o directorio al que se está intentando acceder.

HTTP_version: indica la versión del protocolo HTTP que puede manejar el navegador.

Al final se indica un salto de línea.

Un ejemplo seria el siguiente:

`GET http://www.eidos.es/index.html HTTP/1.1`

A continuación de esta linea de petición se envían otras cabeceras con otro tipo de información, puede verse más información en el documento pdf localizado en la carpeta “documentacion adicional\JEE” llamado “Tecnología de Servidor Java.pdf”, cuarto capítulo, sección “Cabeceras de petición del protocolo HTTP”.

El servidor recibe toda esa información y comienza a comprobarla y a recopilar la información solicitada y la envía de vuelta al navegador. Esa información va precedida por las correspondientes cabeceras.

Las distintas cabeceras de petición del protocolo HTTP son las siguientes:

Accept	Indica al servidor el tipo de datos que el navegador puede aceptar. Estos tipos se indican mediante el uso de tipos MIME.
Accept-Charset	Indica al servidor que conjunto de caracteres puede utilizar el navegador.
Accept-Encoding	Indica al servidor el tipo de codificación de datos q el navegador acepta.
Accept-Language	Indica al servidor el lenguaje natural que prefiere el navegador.
Authorization	Utilizado por el navegador para autenticarse con el servidor.
Cache-Control	Se utiliza por el cliente para como se almacenarán las páginas en la cache de los servidores proxy.
Connection	Indica si el cliente puede mantener conexiones HTTP persistentes.
Content-Length	Indica el tamaño de los datos transferidos en bytes. Aplicable únicamente a peticiones POST.
Content-Type	Indica el tipo de datos que se esta transfiriendo.
Cookie	Esta cabecera se usa para devolver las cookies a los servidores que previamente las han enviado a los navegadores.
Expect	Esta casi no se usa. Permite al cliente decir al servidor que tipo de comportamiento espera de el.
From	Indica el correo electrónico de la persona responsable de la petición HTTP.
Host	Esta cabecera requerida indica el nombre del servidor y el número de puerto.
If-Match	Esta casi no se usa. Se aplica a peticiones PUT, y permite especificar al cliente si una petición se va a realizar atendiendo a la coincidencia de unas etiquetas.
If-Modified-Since	Indica que el cliente quiere una página solo si ha sido modificada después de una fecha específica.
If-None-Match	Similar a if-Match, except que la operación se realizará solo si las etiquetas no coinciden.
If-Range	Casi no se usa. Permite al cliente pedir partes perdidas de un document parcial.
If-Unmodified-Since	Esta cabecera realize la función contraria a if-Modified-Since
Pragma	Solo puede tener el valor no-cache, con lo que indica que siempre se debe obtener una página aunque se posea una copia local.
Proxy-Authorization	Permite a los clientes autenticarse ante los servidores proxy que lo requieran.
Range	Muy similar a if-Range
Referer	Indica al servidor la URL del enlace que fue utilizado para enviar la cabecera del método de petición al servidor.
Upgrade	Permite especificar al cliente si prefiere utilizar un protocolo preferido distinto del HTTP 1.1
User-Agent	Indica el tipo de navegador que esta utilizando la petición
Via	Esta cabecera es añadida por pasarelas y proxies para mostrar los sitios intermedios por los que ha pasado una petición.
Warning	Raramente utilizado, permite indicar a los clientes errores de cache o de transformación.

Para leer las cabeceras desde el servlet podemos hacerlo de forma muy sencilla usando el método **getHeader() del interfaz HttpServletRequest**, pasándole como parámetro el nombre de la cabecera de la que deseamos obtener su valor. También hay otros métodos especializados para obtener cabeceras de petición HTTP. Por ejemplo, getCookies, getAuthType, getContentType, getHeaderNames, gertHeaders, etc., ya vistos anteriormente.

<u>ServletRequest</u>	Este es el interfaz del que hereda HttpServletRequest. Ofrece una serie de métodos que nos permiten tratar las peticiones realizadas a un servlet. Podemos obtener información de formularios, parámetros, atributos, etc. Aunque siempre utilizaremos un objeto HttpServletRequest y no un objeto ServletRequest, ya que hereda de él.
Object getAttribute(String nombre):	devuelve el valor de un atributo determinado almacenado en la petición, si el atributo no existe devuelve null. Es muy útil para pasar información en forma de objetos entre servlets o entre un servlet y una página JSP.
Enumeration getAttributeNames()	devuelve un objeto Enumeration con los nombres de todos los parámetros disponibles en una petición.
String getCharacterEncoding()	devuelve el nombre del tipo de codificación empleado en el cuerpo de la petición.
int getContentLength()	devuelve la longitud en bytes, del cuerpo de la petición. Si esta no es conocida devuelve -1.
String getContentType()	devuelve el tipo MIME que se corresponde con el cuerpo de la petición, o el valor null si este es desconocido.
ServletInputStream getInputStream()	devuelve el cuerpo de la petición como datos binarios utilizando un objeto ServletInputStream.
Locale getLocale()	devuelve el objeto java.util.Locale preferido para el que el cliente aceptará el contenido, basado en la cabecera Accept-Language.
Enumeration getLocales()	devuelve un objeto Enumeration que contiene objetos Locale, según el orden de preferencia del cliente, y basándose siempre en la cabecera Accept-Language.
String getParameter(String nombre)	devuelve el valor de un parámetro, que se corresponde con un campo de un formulario, como un objeto String. Si el parámetro no existe se devolverá el valor null.
Enumeration getParameterNames()	devuelve en un objeto Enumeration los nombres de todos los parámetros contenidos en una petición determinada.
String[] getParameterValues(String nombre)	devuelve en un array de objetos String todos los valores del parámetro cuyo nombre se indica, si no existen estos valores devuelve null.
String getProtocol()	devuelve el nombre y versión del protocolo que utiliza la petición, de la forma protocolo/version_mayor.version_menor., por ejemplo HTTP/1.1.

BufferedReader getReader()	devuelve el cuerpo de la petición como caracteres utilizando un objeto java.io.BufferedReader.
String getRemoteAddr()	devuelve la dirección IP del cliente que envió la petición.
String getRemoteHost()	devuelve el nombre de la máquina del cliente que realizó la petición, si este nombre no está disponible, devuelve la dirección IP.
RequestDispatcher getRequestDispatcher(String ruta)	devuelve un objeto javax.servlet.RequestDispatcher, que actúa como un envoltorio para un recurso localizado en la ruta dada.
String getScheme()	devuelve el nombre del esquema utilizado para realizar la petición, por ejemplo, http, https o ftp.
String getServerName()	devuelve el nombre del servidor que ha recibido la petición.
int getServerPort()	devuelve el número de puerto en el que se ha recibido la petición.
boolean isSecure()	devuelve un valor booleano indicando si la petición a través de un canal seguro como puede ser el protocolo HTTPS.
void removeAttribute(String nombre)	elimina un atributo de la petición.
void setAttribute(String nombre, Object objeto)	almacena un atributo en la petición.

HttpServletResponse

Este es el interfaz que nos permite enviar información al cliente. Pone a nuestra disposición una serie de métodos para poder trabajar con él. Tal como ocurría con el interfaz HttpServletRequest, este interfaz hereda de una clase más general, que define para los servlets genéricos la forma de enviar información al usuario. Los métodos que contiene:

void addCookie(Cookie cookie)	añade la cookie indicada a la respuesta.
void addDateHeader(String nombre, long fecha)	añade una cabecera de respuesta con el nombre y fecha indicados.
void addHeader(String nombre, String valor)	añade una cabecera de petición con el nombre y valor indicado.
void addIntHeader(String nombre, int valor)	añade una cabecera de petición con el nombre dado y el valor entero indicado.
boolean containsHeader(String nombre)	devuelve una valor booleano indicando si la cabecera de respuesta facilitada ha sido ya establecida.

String encodeRedirectURL(String url)	codifica la URL especificada, devolviendo el resultado como un objeto String.
String encodeURL(String url)	codifica la URL especificada incluyendo el identificador de sesión si es necesario. Devuelve el resultado en un String.
void sendError(int codigoDeEstado)	envía respuesta de error al cliente usando el código indicado.
void sendError(int codigoDeEstado, String mensaje)	envía una respuesta de error al cliente utilizando el código indicado y el mensaje descriptivo indicado.
void sendRedirect(String localizacion)	envia una cabecera de respuesta de redirección al cliente utilizando la localización indicada por parámetro.
void setDateHeader(String nombre, long fecha)	establece una cabecera de respuesta con el nombre indicado y la fecha indicada. La diferencia con el método addDateHeader(), es que addDateHeader() permite añadir nuevos valores a cabeceras existentes, y el método setDateHeader() establece un valor de la cabecera que sobrescribe el existente.
void setHeader(String nombre, String valor)	establece una cabecera de respuesta con el nombre y valor indicados.
void setIntHeader(String nombre, int valor)	establece una cabecera de respuesta con el nombre y valor entero indicados.
void setStatus(int codigoDeEstado)	establece el código de estado para la respuesta correspondiente.
<i>ServletResponse</i>	En nuestros servlets también necesitaremos usar los métodos del interfaz padre, que representa a los servlets genéricos:
void flushBuffer()	fuerza que cualquier contenido del buffer de salida sea enviado al cliente.
int getBufferSize()	devuelve el tamaño actual del buffer de la respuesta.
String getCharacterEncoding()	devuelve el nombre del conjunto de caracteres utilizado para la respuesta que se envía al cliente.
Locale getLocale():	devuelve el identificador local asignado a la respuesta, como un objeto java.util.Locale.
ServletOutputStream getOutputStream():	devuelve un flujo de salida de la clase javax.servlet.ServletOutputStream, utilizado para escribir datos binarios en la respuesta.
PrintWriter getWriter()	devuelve un flujo de salida de tipo java.io.PrintWriter, que

	permite enviar datos en formato de texto al cliente. Es muy utilizado por los servlets, ya que permite construir el documento que se devuelve al cliente.
boolean isCommitted():	devuelve verdadero si la respuesta se ha enviado.
void reset()	elimina cualquier información que se encuentre en el buffer, así como cualquier código de estado o cabecera de respuesta.
void setBufferSize(int tamaño)	establece el tamaño del buffer que se utilizará para enviar el cuerpo de la respuesta. El tamaño del buffer debe ser establecido antes de escribir cualquier contenido en el cuerpo de la respuesta, si se hace después producirá una excepción.
void setContentLength(int longitud)	este método permite especificar la longitud del contenido del cuerpo de la respuesta. En servlets HTTP, este método establece la cabecera del protocolo HTTP llamada Content-Length.
void setContentType(String tipo)	establece el tipo de contenido de la respuesta enviada al cliente, para ello se utilizan los tipos MIME.
void setLocale(Locale local)	establece la configuración local de la respuesta enviada al cliente.

Cabeceras de respuesta HTTP

Tal como ocurría con la petición, en la respuesta también se envían cabeceras y existe también una primera línea de estado de respuesta. Esta línea esta formada por la versión del protocolo, seguida por un código de estado. El formato de una línea de estado de respuesta es el siguiente:

Protocolo/Version código_estado descripción_estado

Por ejemplo:

HTTP/1.1 200 OK

Siguiendo a esta línea van otras donde se especifican cosas como la fecha, el software del servidor, el content-type, content-length, etc.

Enviendo información al cliente

La función principal del interfaz `HttpServletResponse` es enviar información de vuelta al usuario en forma de documento HTML. Antes de comenzar a generar ese documento HTML debe especificar el tipo de contenido que se va a enviar al navegador.

Para ello se utiliza el método `setContentType()`, pasándole por parámetro un objeto de la clase `String` que se corresponde con el tipo MIME del documento en casi todos los casos este será `text/html`. Los tipos MIME mas comunes son Application, Audio (basic, x-aiff y x-wav), Image, Text o Video. Los mas comunes con sus subtipos son: `text/html`, `text/plain`, `image/gif`, `image/jpeg`.

Una vez establecido el tipo de contenido que va a generar nuestro servlet, podemos obtener el objeto `PrintWriter` para enviar la información que forma parte del documento que se va a ir generando. Para ello se usa el **método `getWriter()`** del interfaz `HttpServletResponse`.

El flujo de salida `PrintWriter` utiliza un buffer intermedio al que se va enviando la información, para ello los métodos que se emplean son sobretodo `print()` y `println()`. El contenido del buffer intermedio se va enviando al cliente según se va llenando el buffer. Una vez que se envía algo de contenido al cliente ya no es posible cambiar o añadir cabeceras de respuesta, tampoco se puede redirigir al cliente a otra página o recurso.

El contenido también se puede enviar forzando al buffer a que lo haga usando el método `flushBuffer()`. O bien cuando el servlet ha terminado su ejecución.

Un código de ejemplo de un servlet que envía información al cliente es este:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public Class servletSalida extends HttpServlet {

    public void doGet (HttpServletRequest req, HttpServletResponse res)
        throws IOException, ServletException {

        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><head><title>Pagina de respuesta</title></head><body>");
        out.println("Hola Mundo!!!");
        out.println("</body></html>");
    }
}
```

RequestDispatcher

Este interfaz que pertenece al paquete javax.servlet **representa un recurso localizado en una ruta determinada**, y a través de un objeto RequestDispatcher podemos **redirigir una petición de un cliente a ese recurso o bien incluir la salida de la ejecución de ese recurso dentro del servlet correspondiente**.

Para obtener un objeto RequestDispatcher lanzaremos el método **getRequestDispatcher()** sobre el objeto HttpServletRequest correspondiente. A este método se le pasa un parámetro de tipo String que contiene la ruta del recurso correspondiente, representado mediante su URI.

Como ya hemos dicho, la finalidad del interfaz RequestDispatcher es la de ofrecer una especie de envoltorio sobre un recurso del servidor, para permitirnos realizar dos acciones básicas con este recurso desde nuestro servlet. Estas acciones son, por un lado redirigir la petición del cliente al recurso y por otro lado incluir en el servlet el resultado de la ejecución de dicho recurso.

El interfaz RequestDispatcher ofrece dos únicos métodos, y cada uno cubre la funcionalidad comentada. Estos métodos son:

void forward(ServletRequest request, ServletResponse response)

redirige una petición desde un servlet a otro recurso (servlet, página JSP, o página HTML) dentro del servidor.

void include(ServletRequest request, ServletResponse response)

incluye el contenido de un recurso (servlet, página JSP, o página HTML) en la respuesta.

En un principio se puede pensar que la funcionalidad ofrecida por forward() es muy similar a la ofrecida por el método sendRedirect() del interfaz HttpServletResponse, sin embargo son completamente diferentes.

El método forward() redirige la petición al recurso que representa el objeto RequestDispatcher correspondiente, pero además le pasa la petición inicial a dicho recurso, y además una vez ejecutado el método forward() se vuelve a pasar el control de la ejecución al servlet original que realizó la redirección.

El método forward generará un error si ya hemos enviado algo de información del buffer al cliente.

Un código de ejemplo de un RequestDispatcher dentro de un servlet sería este:

```
RequestDispatcher rd = request.getRequestDispatcher("rutaAOtroServlet");
rd.forward(request, response);
```

El objeto RequestDispatcher se puede obtener también a través del método getRequestDispatcher del interfaz javax.servlet.ServletContext. La única diferencia con el método getRequestDispatcher del interfaz ServletRequest es que **el método getRequestDispatcher del interfaz ServletContext solo admite rutas absolutas como parámetro, mientras que en el interfaz ServletRequest admite también rutas relativas**.

HttpSession

Ya comentamos muy por encima al comienzo de este tema de los servlets, que estos pueden mantener información a lo largo de una sesión del cliente con nuestra aplicación web. Cada cliente tiene su propia sesión y dentro de la sesión se pueden almacenar cualquier tipo de objetos.

El protocolo HTTP es un protocolo sin estado, es decir no puede almacenar información entre diferentes conexiones HTTP. Se deben utilizar otros mecanismos. El interfaz HttpSession y el interfaz ServletContext nos permiten de forma sencilla y directa almacenar información abstrayéndose del uso de cookies y encabezados HTTP.

La información almacenada en objetos HttpSession y ServletContext difieren en el ámbito de la misma, una tendrá ámbito de sesión y la otra tiene un ámbito mas general de tipo aplicación.

El ámbito de sesión es visible para un único usuario y el ámbito de aplicación es visible para todos los usuarios de la aplicación.

Existen tres mecanismos tradicionales para mantener el estado entre diferentes peticiones HTTP, estas soluciones son la cookies, los campos ocultos de los formularios y el mecanismo de reescritura de URLs.

La solución que vamos a utilizar en los servlets para poder mantener la información entre distintas peticiones de un mismo usuario en una sesión con nuestra aplicación web, es el API de alto nivel representado por la interfaz javax.servlet.HttpSession. Este API se basa en los mecanismos de cookies y URL-Rewriting, según la situación. El desarrollador de servlets no tiene que preocuparse de esto ya que se realiza de modo automatico.

El desarrollador tratará directamente con el objeto HttpSession para utilizar y acceder a la sesión de cada usuario.

La forma de diferenciar a cada usuario es a través de la ID de sesión del usuario. Cada usuario tiene un identificador único para su sesión. La sesión se mantiene por un periodo de tiempo especificado entre varias peticiones de un usuario.

Es importante aclarar que las sesiones de un mismo usuario en distintas aplicaciones Web no son visibles entre si.

Los métodos que ofrece la interfaz HttpSession son:

Object getAttribute(String nombreAtributo):	devuelve el objeto almacenado en la sesión actual, y cuya referencia se corresponde con el nombre de atributo indicado por parámetro como un objeto de la clase String. Este método siempre devolverá un objeto de la clase Object y deberemos realizar la transformación correspondiente. Devuelve null si el objeto no existe.
Enumeration.getAttributeNames()	devuelve un objeto Enumeration que contiene los nombres de todos los objetos almacenados en la sesión actual.
long getCreationTime()	devuelve la fecha y hora en la que fue creada la sesión (en unix time).
String getid()	devuelve una cadena que se corresponde con el identificador de sesión.
long getLastAccesedTime()	devuelve en milisegundos la fecha y hora de la ultima vez que se realize una petición asociada a la session actual.

int getMaxInactiveInterval()	devuelve el máximo intervalo de tiempo, en segundos, en el que una sesión permanece activa entre dos peticiones distintas de un mismo cliente. Es decir, es el tiempo de espera máximo en que una sesión permanece activa. El valor por defecto es 30 segundos.
void invalidate():	destruye la sesión de forma explícita, y libera de memoria todos los objetos que contiene.
boolean isNew():	verdadero si la sesión se acaba de crear en la petición actual, o el cliente todavía no ha aceptado la sesión (puede rechazar la cookie). Devolverá falso si la petición ya pertenece a la sesión, es decir si ha sido creada anteriormente en otra petición de la misma sesión.
void removeAttribute(String nombreAtributo)	elimina el objeto almacenado en la sesión cuyo nombre se pasa por parámetro.
void setAttribute(String nombre, Object valor)	almacena un objeto en la sesión utilizando como referencia el nombre indicado como parámetro.
void setMaxInactiveInterval(int interval)	establece, en segundos, el tiempo máximo de inactividad de una sesión antes de ser destruida.

Por defecto los servlets no utilizan sesiones ni pertenecen a una sesión concreta. Si queremos que un servlet cree una sesión que pueda ser utilizada por todos los servlets de nuestra aplicación debemos hacerlo a través de código.

El interfaz HttpServletRequest ofrece un método específico para la creación de una sesión, creándola cuando sea necesaria, es decir cuando todavía no ha sido creada.

Para comprobar si existe una sesión, es decir un objeto HttpSession, se usa el método **getSession()**. Si el servlet no obtiene una cookie o una información de la URL que le indique el identificador de sesión, significa que la sesión aún no ha sido creada. Si al método getSession() le indicamos el parámetro **true**, creará la sesión en caso de que no exista. Y devolverá el objeto HttpSession que representa a la sesión actual.

Si al método getSession(), le indicamos el valor **false**, y la sesión no existe, nos devolverá un valor null, pero si la sesión sí existe devolverá el objeto HttpSession correspondiente.

<u>ServletContext</u>	<p>Ya habíamos visto antes que este interfaz nos permite comunicarnos con el contenedor de servlets, y obtener así información del servidor y también lo hemos utilizado con el RequestDispatcher.</p> <p>Ya hemos comentado también que el ámbito de ServletContext es más general que el de HttpSession ya que HttpSession es particular a una sesión de un usuario y sin embargo ServletContext tiene ámbito de aplicación y por lo tanto es compartida por todos los usuarios de la aplicación.</p> <p>En un objeto ServletContext podemos almacenar cualquier tipo de objeto al igual que lo hacímos a nivel de sesión, pero en este caso se trata de objetos a nivel de aplicación comunes a todos los usuarios de la aplicación web. Al ser comunes podemos tener problemas de concurrencia.</p> <p>Los métodos de esta interfaz son:</p>
Object getAttribute(String nombre)	devuelve el objeto (atributo) almacenado en la aplicación (contexto), que se corresponde con el nombre pasado por parámetro. Si no existe devuelve null.

Enumeration getAttributeNames():	devuelve un objeto Enumeration que contiene todos los nombres de los atributos existentes en la aplicación Web.
ServletContext getContext(String ruta):	devuelve un objeto ServletContext que se corresponde con un recurso del servidor
String getInitParameter(String nombre):	devuelve una cadena que contiene el valor de un parámetro de inicialización del contexto. Si no existe devuelve null.
Enumeration getInitParameters():	devuelve todos los nombres de los parámetros de inicialización en un objeto Enumeration.
int getMajorVersion():	devuelve el numero superior de versión que se corresponde con la versión del API servlet implementada por el contenedor de servlets.
String getMimeType(String fichero):	devuelve en una cadena el tipo MIME del fichero especificado, si el tipo es desconocido devuelve null.
int getMinorVersion()	devuelve el numero de versión inferior que se corresponde con la versión del API servlet implementada por el contenedor de servlets.
RequestDispatcher getNamedDispatcher(String nombre):	devuelve un objeto RequestDispatcher que actuará como un envoltorio que representa al servlet cuyo nombre se pasa por parámetro.
String getRealPath(String ruta):	devuelve un objeto String que contiene la ruta real (física) de una ruta virtual especificada como parámetro.
RequestDispatcher getRequestDispatcher(String ruta)	se usa para obtener una referencia a un recurso en el servidor y poder incluir su ejecución en el servlet actual o redirigir la ejecución a este recurso. Lo vimos anteriormente.
URL getResource(String ruta):	devuelve un objeto URL que se corresponde con el recurso localizado en la ruta indicada
InputStream getResourceAsStream(String ruta):	devuelve el recurso localizado en la ruta indicada como objeto InputStream.
String getServerInfo()	devuelve el nombre y versión del contenedor de servlets en el que el servlet se está ejecutando.
void log(String mensaje)	escribe el mensaje especificado en el fichero de registro de un servlet.
void log(String mensaje, Throwable throwable)	escribe un mensaje descriptivo y un volcado de pila para una excepción determinada en el fichero de registro de un servlet.
void removeAttribute(String nombre)	elimina el atributo indicado de la aplicación web, liberando todos los recursos asociados.
void setAttribute(String nombre, Object objeto)	almacena un objeto determinado en la aplicación, este método y el anterior tienen el mismo significado que los que aparecían en el interfaz HttpSession, pero en esta caso se trata de almacenar, recuperar y eliminar objetos a nivel de aplicación. Los objetos almacenados en la aplicación existirán durante toda la vida de la aplicación Web a no ser que los eliminemos mediante el método removeAttribute(). Por lo tanto no existe un tiempo máximo de inactividad, ni un método para destruir la aplicación, como ocurría con el interfaz HttpSession. Una aplicación se destruirá cuando se detenga el contenedor de servlets que la

	contiene.
Como dijimos antes, los objetos de aplicación están accesibles para todos los usuarios y por lo tanto corren el riesgo de ser accedidos por más de un usuario a la vez, con lo que surge un problema de concurrencia. Para resolver los problemas de concurrencia hay dos soluciones.	
Una de ellas es la implementación por parte del servlet, que va a acceder a los objetos de la aplicación del interfaz javax.servlet.SingleThreadModel. Si el servlet implementa esta interfaz, nos asegura que solo va a servir una implementación cada vez, por lo tanto no existirán múltiples hilos de ejecución que puedan acceder a un mismo objeto de la aplicación al mismo tiempo.	
El interfaz SingleThreadModel no posee ningún método, si queremos que un servlet lo implemente simplemente debemos indicarlo en la cláusula implements de la declaración de la clase del servlet.	
Esta es la solución más sencilla pero también la más drástica ya que eliminamos la interesante característica que nos brindaban los servlets a través de los múltiples hilos de ejecución simultáneos. Además se verá afectado el rendimiento de la aplicación web, ya que cada petición deberá esperar cola hasta que la instancia del servlet quede libre.	
Otro motivo para desechar esta solución es que algunos contenedores, cuando se indica que el servlet implementa SingleThreadModel, crearán un conjunto de instancias de servlets, y por lo tanto no se podrá asegurar el acceso exclusivo a un objeto de la aplicación.	
Debe señalarse que los problemas de concurrencia no afectan exclusivamente a la hora de utilizar los objetos a nivel de aplicación, sino que también podemos tener el problema de los accesos simultáneos a la hora de acceder a los atributos del propio servlet (variables miembro) y a la hora de acceder a métodos estáticos de un servlet.	
La segunda solución que se comentaba, y que resulta ideal, consiste en permitir la ejecución multihilo de los servlets y solucionar los problemas de sincronización utilizando la palabra clave synchronized cuando exista un problema potencial de accesos concurrentes.	
Otra consideración a la hora de utilizar los objetos a nivel de aplicación y que difiere con los objetos a nivel de sesión, se refiere a que la aplicación existe mientras el contenedor de servlets se esté ejecutando, por lo tanto no será necesario crear la aplicación en ningún momento, y la destrucción o eliminación de un objeto almacenado en la aplicación no se notifica al objeto afectado mediante ningún evento.	
Un ejemplo de código que usa synchronized dentro de un método, por ejemplo doGet, en un servlet sería este:	
<pre>ServletContext application = getServletContext(); Synchronized (application) { String nombreAplicacion = (String) application.getAttribute("nombreApp"); If (nombreAplicacion != null) { out.println("El objeto nombreAplicacion ya existe"); } else { application.setAttribute("nombreApp", new String("Aplicación con Servlets")); out.println("Se ha creado el objeto nombreApp"); } }</pre>	
Se puede observar en el código que se ha utilizado la palabra reservada synchronized aplicándola al objeto ServletContext. Solo se debe utilizar synchronized cuando sea absolutamente necesario, y se debe aplicar al objeto adecuado, como ocurre en este caso.	
El objeto ServletContext tiene otras dos funcionalidades, una es permitir el acceso al contenedor de servlets en el que se está ejecutando nuestro Servlet, pudiendo tener así información acerca del producto utilizado y las versiones del API de Java servlet. Por ejemplo a través de los métodos getServerInfo() o getMajorVersion() y getMinorVersion().	

La otra es poder almacenar información del tipo mensajes de alerta o de depuración en los ficheros de registro del contenedor de servlets. Para ello usamos el método log() del interfaz ServletContext. Se puede usar en todos los métodos del ciclo de vida del servlet.

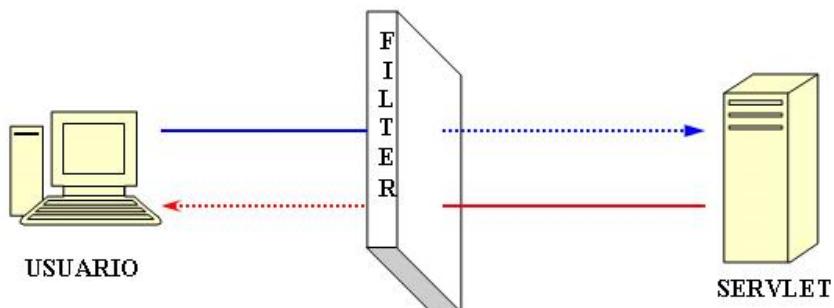
El fichero de registro en el caso del contenedor Jakarta Tomcat está en el directorio \jakarta-tomcat\logs y se llama SERVLET.LOG.

Para más información consultar el PDF llamado Tecnología de Servidor Java.pdf disponible en la carpeta de documentación adicional/jee.

Filtros

Un **filtro** es una porción de código que se ejecuta antes que cualquier otra cosa en una aplicación web. Un Filtro de J2EE es ejecutado por el contenedor siempre que el usuario intenta ingresar una URL que cumpla con un patrón descrito en el archivo **web.xml**.

Los Filtros no son visibles desde los Servlets. La función de estos es de analizar y filtrar los datos. Intercepta la petición y respuesta. Puede haber todos los filtros que se deseen.



Los filtros han de ser descritos y/o declarados en web.xml. La forma de describir un filtro en web.xml es, por ejemplo, la siguiente:

```
<filter>
  <filter-name>SessionFilter</filter-name>
  <filter-class>nombrePaquete.SesionFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>SessionFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Como se puede ver se está indicando un nombre para el filtro y la clase con la que se corresponde, indicando la ruta completa (paquete + nombre de la clase). Además se hace el mapeo de las páginas a las que se va a aplicar dicho filtro.

Los filtros tienen un ciclo de vida que se compone de un inicio representado por el método **init()**, el filtro en sí representado por el método **doFilter()** y una destrucción representada por el método **destroy()**. En el método **doFilter()**, se pasan varios parámetros, uno de ellos es de tipo **FilterChain** que se usa para pasar el flujo al siguiente paso de la petición del usuario que se ejecutara después de que se ejecuta este filtro, el siguiente paso puede ser otro filtro algún servlet, etc.

Para crear un filtro se debe **implementar la clase Filter**.

Un ejemplo de código de filtro es:

```
package com.atrium.filtros;
import java.io.IOException;
import java.util.Enumeration;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;

public class Filtro_Idioma implements Filter {

    private boolean no_elegido = true;
    private String idioma_elegido;
    private boolean depurar;

    public void destroy() {
        if (depurar) {
            System.out.println("TERMINA EL INTERCEPTOR DE IDIOMAS");
        }
    }

    public void doFilter(ServletRequest peticion, ServletResponse respuesta,
                         FilterChain chain) throws IOException, ServletException
    {
        HttpServletRequest mi_request = null;
        if (peticion instanceof HttpServletRequest) {
            mi_request = (HttpServletRequest) peticion;
        }
        String idioma_preferido = mi_request.getHeader("accept-language");
        String lista_idiomas[] = idioma_preferido.split(",");
        for (int i = 0; i < lista_idiomas.length; i++) {
            if (lista_idiomas[i].substring(0, 2).equals("es") &&
no_elegido) {
                idioma_elegido = "es";
                no_elegido = false;
            }
            if (lista_idiomas[i].substring(0, 2).equals("en") &&
no_elegido) {
                idioma_elegido = "en";
                no_elegido = false;
            }
            if (lista_idiomas[i].substring(0, 2).equals("fr") &&
no_elegido) {
                idioma_elegido = "fr";
                no_elegido = false;
            }
        }
    }
}
```

```

        mi_request.getSession()
                    .setAttribute("idioma_elegido", idioma_elegido);
        chain.doFilter(peticion, respuesta);
    }

    public void init(FilterConfig filterConfig) throws ServletException
{
    idioma_elegido = "com/atrium/idiomas/textos_"
                    +
filterConfig.getServletContext().getInitParameter(
                    "idioma_por_defecto");
    // PARAMETRO GENERAL DE APLICACION
    String tipo_ejecucion = filterConfig.getServletContext()
                    .getInitParameter("tipo_ejecucion");
    if (tipo_ejecucion.equals("d")) {
        depurar = true;
    } else {
        depurar = false;
    }
    if (depurar) {
        System.out.println("EMPIEZA EL INTERCEPTOR DE IDIOMAS");
    }
}
}

```

Escuchadores

OBJETOS	ESCUCHADORES	PROPIEDADES	EVENTOS
servletContext	ServletContextListener ServletContextAttributeListener	contextInitialized contextDestroyed AttributeAdded AttributeReplaced AttributeRemoved	servLetContextEvent servLetContextAttributeEvent
HttpSession	HttpSessionListener HttpSessionAttributeListener HttpSessionActivationListener	AttributeAdded AttributeReplaced AttributeRemoved sessionCreated sessionRemoved sessionDidActivate sessionWillPassivate	httpSessionEvent httpSessionBindingEvent httpSessionEvent
HttpServletRequest	ServletRequestListener ServletRequestAttributeListener	RequestInitialized RequestDestroyed AttributeAdded AttributeReplaced AttributeRemoved	servletRequestEvent servletRequestAttributeEvent

Los escuchadores son clases java que se ejecutan cuando se produce un determinado evento, pueden tener ámbito de aplicación y/o de sesión.

La clase del evento debe implementar alguna de las interfaces de Listener como ServletContentListener, HttpSessionListener, HttpSessionAttributeListener, etc:

```
public class Eventos_ServletContext implements
ServletRequestListener
```

Los eventos para los que puede dispararse son los sucesos que pueden ocurrir en una aplicación web, como el inicio o fin de una sesión, o la creación o destrucción de un atributo, que pueden ser escuchados por el código de la aplicación.

La manera de responder a estos eventos se basa en los mismos principios que ya vimos en la parte de J2SE cuando hicimos aplicaciones de escritorio: la utilización de interfaces de escucha.

Hay una amplia colección de interfaces de escucha para poder responder a los eventos, las más importantes son:

ContextRequestListener. Dispone de los métodos de escucha contextInitialized() y contextDestroyed(), mediante los cuales se pueden responder a los sucesos de inicialización y destrucción de la aplicación, respectivamente. Ambos métodos reciben como parámetro un objeto ServletContextEvent que dispone del método getServletContext() para poder obtener una referencia al contexto de aplicación.

ServletContextAttributeListener. Dispone de los métodos attributeAdded(), attributeRemoved() y attributeReplaced(), los cuales son invocados cuando se añade un nuevo atributo de aplicación, se destruye uno existente o se reemplaza el valor existente en un atributo por uno nuevo, respectivamente. Los tres métodos reciben como parámetro un objeto ServletContextAttributeEvent que nos proporciona métodos para acceder al nombre y valor del atributo.

HttpSessionListener. Dispone de los métodos sessionCreated() y sessionDestroyed(), que son invocados cada vez que se crea o se destruye una sesión respectivamente. Los dos métodos reciben como parámetro un objeto HttpSessionEvent que nos permite obtener una referencia a la sesión creada o que va a ser destruida.

HttpSessionAttributeListener. Dispone de los mismos métodos que la interfaz ServletContextAttributeListener, solo que en este caso, los métodos son invocados cuando la acción tiene lugar en un atributo de sesión.

ServletRequestListener. Dispone de los métodos de escucha requestInitialized() y requestDestroyed() para responder a los eventos inicio de petición y fin de petición, respectivamente. El parámetro ServletRequestEvent que reciben ambos métodos, dispone de los métodos getServletRequest() y getServletContext(), que permiten obtener los objetos request y servletContext, respectivamente.

ServletRequestAttributeListener. Dispone de los mismos métodos que la interfaz ServletContextAttributeListener, solo que en este caso, los métodos son invocados cuando la acción tiene lugar en un ámbito de petición.

Para poder responder a cualquiera de los eventos que pueden tener lugar en la aplicación, es necesario realizar dos operaciones:

Crear una clase escuchadora. Se debe crear una clase que implemente la interfaz de escucha correspondiente a los eventos que queremos capturar.

Registrar la clase escuchadora. Para que el contenedor Web conozca la existencia del escuchador y pueda llamar a los métodos implementados, es necesario registrar la clase en el archivo de configuración **web.xml**.

Un ejemplo de código empleado para registrar un escuchador es:

```
<listener>
    <listener-
class>com.atrium.escuchadores.Eventos_ServletContext</listener-
```

```
class>
</listener>
```

Un ejemplo de clase que implementa una interfaz de escuchador, que además hemos usado en uno de los proyectos usados en el master es este:

```
package com.atrium.escuchadores;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

import org.apache.log4j.PropertyConfigurator;

import com.atrium.util.Mantenimiento_Rutas;

public class Eventos_ServletContext implements
ServletContextListener {

    public void contextDestroyed(ServletContextEvent sce) {

    }

    public void contextInitialized(ServletContextEvent sce) {
        // accedemos al parametro de inicio para establecer
        la ruta inicial de
        // los ficheros logs
        String ruta =
sce.getServletContext().getInitParameter("ruta_logs");
        ruta = sce.getServletContext().getRealPath(ruta);
        // comprobamos la existencia de la carpeta y sino
        existe se crea
        Mantenimiento_Rutas man_rut = new
Mantenimiento_Rutas();
        man_rut.comprobar_Carpeta(ruta);

        // ruta para el escribir el fichero de registro
        (bitacora).esta
        // variable es
        // para el log4j y esta en su fichero de
        configuracion
        System.setProperty("ruta_logs", ruta);
        // ruta para que el log4j encuentre su fichero de
        configuracion
        String ruta_logs =
sce.getServletContext().getRealPath(
        "/WEB-
```

```

INF/classes/com/atrium/util/log4j.properties");
PropertyConfigurator.configure(ruta_logs);

    // EN CASO DE UTILIZAR DISCOPARA REGISTRAR LOS
USUARIOS CONECTADOS
    // BORRAREMOS
    // DICHO ARCHIVO PARA INICIAR TODA LA OPERATIVA.
String tipo_proceso =
sce.getServletContext().getInitParameter(
    "tipo_control");

if (tipo_proceso.equals("disco")) {
    Properties fichero_usuarios = new Properties();

    sce.getServletContext().setAttribute("fichero_usuarios_cone
ctados",
        fichero_usuarios);
    FileWriter flujo_escritura;
    try {
        flujo_escritura = new
FileWriter(sce.getServletContext()

        .getRealPath("seguridad/lista_usuarios.properties"));
        fichero_usuarios.store(flujo_escritura,
        "");
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

}
}

```

Instalacion de tomcat con MyEclipse

Durante la instalacion de Tomcat necesitaremos saber la carpeta donde lo vamos a instalar, el puerto (9080), el nombre de usuario (admin) y la contraseña.

Hay un archivo llamado “apache-tomcat-6.0.29.exe” dentro de una carpeta llamada “Tomcat Programas instalación”. Ese es el archivo de instalación para Windows que vamos a usar en el master.

La carpeta de instalacion puede ser la misma que eclipse (opcional). Por defecto instala en “C:\Archivos de programa\Apache Software Foundation\Tomcat 6.0”

El puerto que no sea 8080 si usamos oracle en localhost.

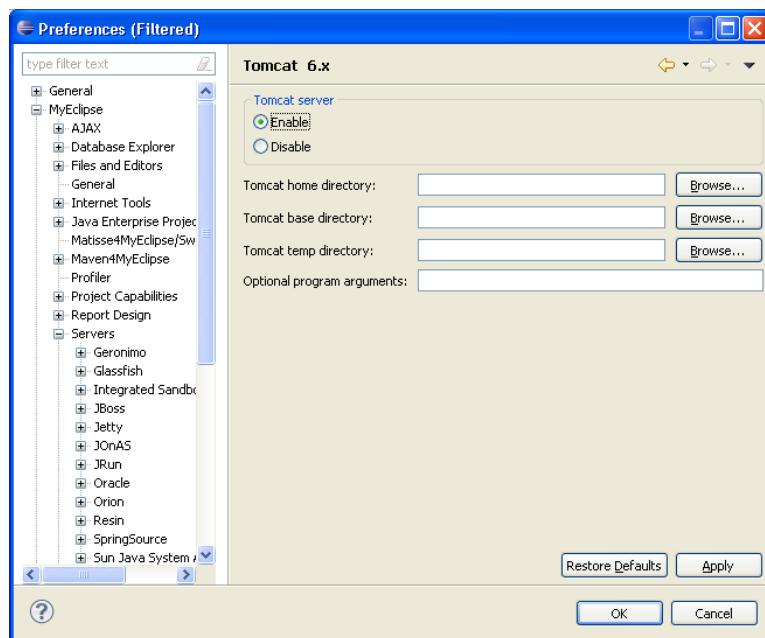
Una vez hecho esto se habrá generado un **servicio** de arranque (Windows - apache Tomcat) con lo cual habrá que ponerlo a **manual** igual que hicimos con oracle. Ya que el servidor lo iniciaremos y detendremos desde MyEclipse.

Despues hay que decirle a MyEclipse donde esta el Tomcat.

Al crear un proyecto nuevo J2EE hay que hacerlo desde la perspectiva de myEclipse Enterprise y luego ir a Archivo – New – Web Project. Se le da nombre y se elije la versión de JDK (5).

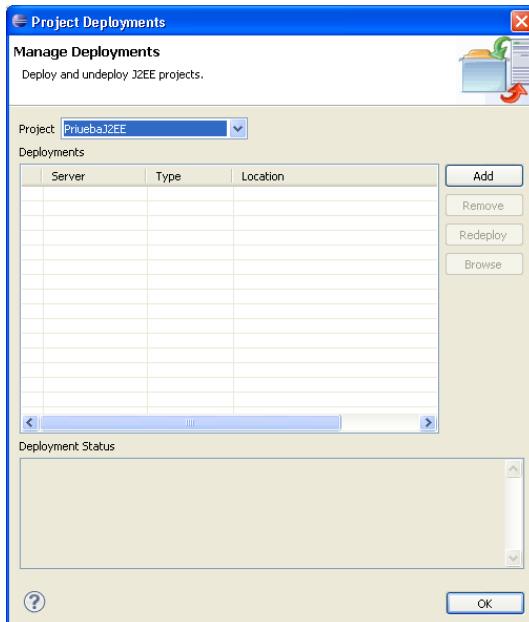
Al hacer esto ya se importan las clases J2EE y toda la estructura de directorios.

Para decirle a MyEclipse donde esta el tomcat se va a MyEclipse – Preferences...



Aquí se despliega MyEclipse en el lado izquierdo, después Servers, Tomcat, se elige la versión que tenemos instalada (6.x). Se marca la casilla Enable dentro de Tomcat server y se busca la ubicación del Tomcat donde lo hayamos instalado en el paso previo.

La primera vez que vamos a probar una aplicación web, siempre hay que implementar la aplicación (deploy), es un ícono a la izquierda del ícono de reinicio de servidores.



En la ventana que se abre al pulsar ese ícono elegimos el proyecto que queremos implementar (deploy), después se pulsa Add y se añade el Tomcat que esta ya instalado y después OK.

JSP

Consultar el PDF llamado Tecnología de Servidor Java.pdf disponible en la carpeta de documentación adicional/jee.

No se escriben clases, sino paginas HTML. Una pagina jsp esta formada por etiquetas HTML (se envian tal y como estan) y etiquetas jsp (generan un servlet y se ejecutan en el servidor).

Las etiquetas jsp en la version 6 no existen. Aunque en la práctica siguen vigentes.

Las etiquetas jsp han ido evolucionado para eliminar al máximo el codigo java, de forma que sean mas faciles de mantener. Es una mala praxis escribir codigo java en la pagina jsp.

Hay tres tipos: Etiquetas, Acciones, Directivas.

Etiquetas:

<!-- Comentario HTML -->

<%-- Cometario JSP--%>

<%! Declaracion %> para inicializar variables u objetos en la pagina. Lo que declare en esa pagina estara disponible en esa pagina. Cuando escribimos una pagina jsp, creamos un servlet.

<%= Expresiones %> se va a resolver esa expresión y el resultado se va a poner en la pagina HTML. El resultado va al cliente. Crea contenido dinamico. El uso mas habitual de una expresión es darle valor a los atributos de cualquier etiqueta sea JSP o no.

<input type = “text” value="<%=_%>">

Las expresiones no acaban en ;

<%= Expresiones %><%=_x=x+3 _%><%=_objeto.metodo() _%>

<% Scriptlets %> trozo de codigo java en medio de la pagina. No es una forma correcta de trabajar.

Muchas de estas etiquetas seran sustituidas por JSLT.

Acciones:

<JSP:forward page=”ruta y jsp//URL Pattern”>

 <JSP:param name=”xxx” value=”<%=_%>”/>

</JSP:forward>

para pasar de un servlet a otro. Acaba generando un ServletDispatcher. Tiene un atributo en el que colocaremos un url pattern o ruta y JSP. Tambien admite otra etiqueta en su cuerpo <JSP:param> que añade parámetros al request. Tiene dos atributos name y

value, que generalmente seran expresiones. En ejecución, primero se resuelven las expresiones JSP

```
<JSP:include page="ruta y JSP" flush="true"/>
    <JSP:param name="xxx" value="<% = %>" />
</JSP:include>
```

Incluye dentro de la página JSP el código HTML que se incluye en otra página. Esto sirve para casi todo.

```
<JSP: useBean id="nombreObjeto" class/type = "paquetes y nombre"
    Scope= "page/request/session/application">
```

Para crear objetos o para crear objetos de los contextos. El atributo ID siempre sera el nombre del objeto depuse class (nuevo objeto) o type (recuperar objeto existente) (nunca los dos), en ambos casos nombre y paquete de la clase. El atributo scope es optativo. Por defecto esta a page (visibilidad de esa pagina), pero si al crear un objeto quiero hacerlo visible para otros contextos lo pongo a request, session o application. Si vamos a recuperar un objeto, scope sera obligatorio.

```
<JSP:setProperty name="nombreObjeto" property="proiedadDeClase"
    value="<% = valor %>">
```

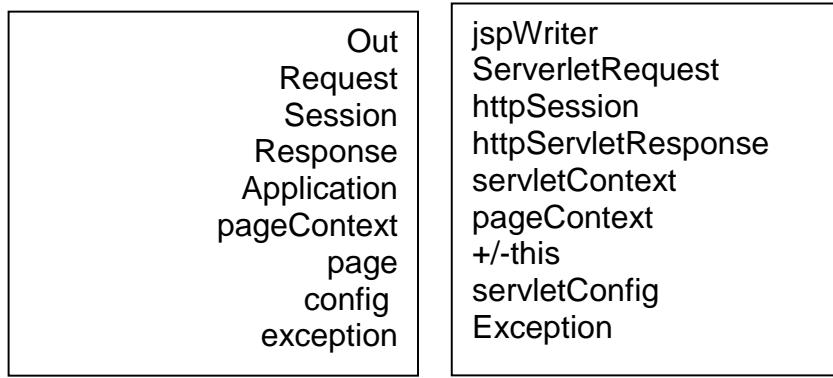
La etiqueta setProperty sera para darle valor a una propiedad de un objeto. Este debe ser un javaBean.

```
<JSP:setProperty name="nombreObjeto" property="proiedadDeClase"
    value="<% = valor %>">
```

Recoge información del objeto y esta información va al cliente en forma de texto.

Directivas: modifican la actuación normal de la pagina.

Objetos implicitos son aquellos que se crean en el servlet y por tanto objetos que yo podre manejar.



```
<%@  

  Language="java"  

  Extends=""  

  Import=""  

  Session="true"  

  Buffer="8kb"  

  AutoFlush="ture"  

  isThreadSafe="true"  

  errorPage=""  

  isErrorPage="false"  

  contextType="text/html"  

  pageEncoding=""  

>
```

Lenguaje vale java por defecto y valdrá eso siempre. Como no se dio soporte a ningún lenguaje mas, es el único valor posible.

Extends solo puede aparecer una vez ya que en java no hay herencia múltiple.

Imports podrán aparecer todos los necesarios y la clase solo fucionará cuando estén todos los que necesita.

Session, si esta a verdadero podemos acceder a la session desde la pagina y viceversa con null.

Buffer, suele venir definido desde el servidor

Autoflush

isThreadSafe, no es recomendable cambiarlo porque se generan errores de concurrencia en las paginas.

errorPage, los errores se pueden resolver a nivel de pagina, de aplicación o de servidor. En **errorPage**, le diremos que pagina queremos devolver en caso de error. Y en esa pagina sera en la que pongamos el **isErrorPage** a verdadero. Este nos permitirá el acceso al objeto **exception**.

contentType y **pageEncoding** suele darles valor el servidor. El **contentType** da la especificación MIME (que está en la cabecera la pagina de texto que se envía/recibe para especificar como debe tratarse). **pageEncoding** es el juego de caracateres, que por defecto se lo dara el servidor (utf-8) y si queremos que sea mas específicamente, pues se lo concretamos.

JSTL

Es un conjunto de etiquetas estándar que usa JSP para generar los componentes de las páginas web y nos dan otras muchas posibilidades.

Para tener toda la información sobre este lenguaje se puede consultar un manual en PDF que está disponible en la carpeta de documentación adicional/jee y se llama JSTL Castellano.pdf.

Etiquetas personalizadas

Se pueden crear etiquetas personalizadas para ser usadas además de las que ofrece el propio JSP.

Una de las carencias de las etiquetas JSP es el control de la lógica. Es decir el uso de if no se puede controlar con JSP. Para solucionar esto se desarrollaron el siguiente conjunto de etiquetas JSTL (Java Standard Tag Library), desarrolladas por la comunidad Apache y adoptado por Sun como proyecto oficial, con lo cual viene en J2EE. De hecho son tan importantes que en la versión 6 (JSF) siguen existiendo. Es un conjunto de ficheros descriptores (extensión TLD) más un conjunto de clases que son llamadas o enlazadas desde estas etiquetas.

Las JSTL están formadas por cuatro categorías distintas, que son **CORE**, para sustituir a las acciones.

La directiva usada para que funcione el paquete CORE es `<%@ taglib URI="http://java.sun.com/jstl/core" prefix="c" %>`

La etiqueta `<c:out value="" default="" />` es una etiqueta JSTL que sustituye a la etiqueta `<%= %>` de JSP.

Existe también un lenguaje de expresiones (**EL** – Expression Language), que se ha creado para sustituir a las expresiones de JSP. Una expresión EL sería por ejemplo `${ ... }` y se pone entre las comillas del value de la etiqueta JSLT, por ejemplo:

`<c:out value="${ ... }" default="..." />`. La expresión default es optativa y se usa para que en caso de que no exista el value, se usa lo que va en default. C:out saca (imprime) el valor de una variable.

La etiqueta **set** amplia la creación de variables de JSP `<c:set value="" var="nombreVariable" scope="" target="" property="" />`, no hace falta poner todas las propiedades del set, solo las que necesitemos. **Value** se usa para dar valor al “nombreVariable”, puede ser una expresión escrita con la forma EL. **Scope**

sirve para indicar en qué contexto se va a almacenar dicha variable, por lo tanto no solo indica el contexto sino la visibilidad y duración, por defecto vale “page” (página, indica que tiene alcance de página, cuando se cierre la página ya no existe), además existen los valores de ámbito habituales que son, **session**, **request** y **application**. Si queremos asignar a la variable un valor de una variable ya existente en otro sitio, usamos **target** y **property**, target es el nombre del objeto y property es el nombre de la propiedad (variable). Por supuesto las clases deben seguir la normativa de JavaBean.

`<c:remove var="nombre" scope="">` elimina la variable, scope por defecto es page (pagina) con lo cual es necesario en el resto de casos.

`<c:if test="%{ ... }> <> </c:if>` se usa como alternativa lógica al if de JSP, test se usa para poner la condición a evaluar y se usa con una expresión EL \${ ... }. No tiene ELSE y si se cumple la condición se ejecuta lo que haya dentro del código que anida. Si no es verdadero se va a la siguiente etiqueta. Dentro de c:if puede usarse un var=”nombre” para guardar el resultado de la expresión del if, y luego se puede usar para preguntar si eso es falso o verdadero. Así mismo puede usarse el atributo scope para indicar el ámbito de esa variable.

`<c:choose>` sirve para aplicar una alternativa multiple, parecido pero más potente que un switch. La sintaxis es:

```
<c:choose>
    <when test="${ ... }">
        Código a ejecutar si se cumple el test
    </when>
    ...
    <otherwise>
        Código a ejecutar por defecto si no se cumple ningún when
    </otherwise>
</c:choose>
```

Dentro de un when puede ir cualquier código.

`<c:ForEach item="${ ... }" var="nombre" varstatus="nombre">` sirve para hacer un bucle parecido al for y al while. El atributo ítem es la tabla, colección o iterador sobre la que vamos a realizar la iteración. A no ser que se indique de otra manera recorrerá la colección completa. El atributo var sirve para indicar el nombre de la variable donde guardaremos el valor obtenido, su alcance es local al ForEach. Como **atributos optativos tenemos begin, end y step**. Begin es la posición donde queremos comenzar, end es la posición donde queremos terminar y step es el salto entre posiciones. Existe un cuarto atributo optativo que es **varstatus** que lleva otro nombre de variable y lo que hace es guardar el objeto en el que se encuentra la iteración. Tiene una serie de valores que pueden ser utilizados, esos valores son, **begin, end, current, index, count, first, last, step**:

Begin, end y step siempre valdrían lo mismo y son iguales que los indicados en el forEach.

Current es un contador de iteraciones que va cambiando, lógicamente a cada iteracion.

Index, si trabajamos con una colección es el elemento dentro de la colección.

First y Last son valores booleanos que serán true o false dependiendo si están en la primera posición o en la última respectivamente.

Varstatus permite una variante para tratar con cadenas de caracteres, lo que hace es separar la cadena en partes por las comas que tenga. Aunque la variable **fortoken** permite usar otro separador distinto de la coma.

```
<c:fortoken    items="${cadena"    a    recorrer}"    var="nombre"
delims="carterSeparador">
    Código HTML a ejecutar
</c:fortoken>
```

Var es donde se guarda el resultado a cada iteración. Delims puede ser un solo carácter o más de uno.

```
<c:import url="" context="/nombreAplicacion" var="" scope="" />
```

Tiene la misma funcionalidad del include de JSP. URL es la url de aquello que estamos pidiendo, puede ser que este en otra aplicación o incluso en otro servidor (contenedor). Context se usa solamente cuando lo que necesitamos está en otra aplicación. Var guarda lo que devuelve el recurso al que se llama. Scope es el ámbito de var.

```
<c:redirect url="..." />
```

redirecciona la página a otro sitio es como el forward de JSP, deteniendo la ejecución de la página en la que está.

```
<c:catch var="nombre"> </c:catch>
```

Captura excepciones. Si se produce una excepción no se detiene la ejecución sino que se guarda en 1 avariable definida en var. Dentro del cuerpo del catch usando un c:if podríamos tratar la excepción.

EL (Expression Language)

Es un lenguaje de expresiones (EL = Expression Language) para usarse en conjunción con JSTL. Se usa la sintaxis `${ ... }` sustituye las expresiones de JSP `<%= %>`

En el cuerpo (dentro de las llaves) pueden ir literales, expresiones, objetos, propiedades, operadores lógicos, operaciones aritméticas.

Aunque puede ir sólo en cualquier página, lo mas habitual es que vaya dentro de un atributo de una etiqueta JSLT.

Las expresiones EL se procesan todas al principio antes de procesar ninguna otra etiqueta de la pagina, con lo cual si estas expresiones dependen de otras operaciones de la página no van a funcionar.

Los operadores lógicos usados son los mismos que los usados en Java. Los operadores aritméticos son tambien los mismos que los usados en Java.

Hay pequeñas diferencias que no existen en Java. Por ejemplo podemos preguntar si un objeto o una colección esta vacía, cosa que no existe en Java. Para hacerlo sería `${objetoA == Empty}`

Mas posibilidades en las expresiones EL serian:

Uso de expresiones ternarias, por ejemplo `${condición ? casoTrue : casoFalse }`

En cuanto objetos, se puede llamar a cualquier objeto y cualquier propiedad de ese objeto siempre que esa clase siga la norma JavaBean. Seria: `${ objeto.propiedad }` Cuando hablamos de objetos incluimos Arrays y Colecciones. En este caso seria `${ obj[numeroEntero o clave en caso de mapas] }` o bien `${ obj.propiedad[numeroEntero o clave en caso de mapas] }`

Las expresiones EL tienen también unos objetos implícitos, que son los siguientes. **RequestScope, SessionScope, ApplicationScope y PageScope**. Sirven para acceder a los objetos según su ámbito y duración. Otro objeto implícito es **param**, sirve para acceder al objeto que se ha enviado en la petición. También existe **paramValues**.

InitParam sirve para acceder (leer) a los objetos de aplicación definidos en los archivos XML de configuración. **Cookie** sirve para acceder a las cookies. **Header** sirve para acceder a las cabeceras HTTP que nos llegaban en la petición.

Con las expresiones EL no pueden llamarse métodos, solo propiedades.

Internacionalización con J2EE

Puede hacerse de tres formas diferentes que además pueden combinarse.

Una primera opción es ofrecer las páginas al usuario en función del idioma establecido en el navegador. Esta información se puede recoger de la cabecera de la petición. Esa información va por pares clave/valor como ya sabemos. La cuestión es saber la clave que nos da el idioma, que es **ACCEPT-LANGUAGE**, su valor puede ser un solo idioma en la normativa correspondiente (idioma_pais), pero también puede ser una lista de idiomas separados por comas. Con lo cual recogiendo ese valor podemos presentar la pagina en el idioma adecuado, se enviaría uno por defecto en caso de no disponer del idioma reflejado en el navegador.. Esto puede montarse en un filtro, que deja en una variable de sesión el idioma en el que vamos a presentar las páginas.

Una segunda opción puede ser almacenando el idioma en un perfil de usuario, es decir en una tabla de la base de datos. Así, una vez que el usuario se ha logeado ya se puede usar el idioma de su elección.

Una tercera opción es que el usuario, dinámicamente pueda cambiar el idioma, a través de hipervínculos en imágenes de banderas, por ejemplo. Esto nos lleva a una idea interesante, es decir si eso queremos que se vea en todas las paginas, es buena idea montar las paginas por partes, creando una cabecera, un cuerpo y un pie de pagina, por ejemplo. De esta forma la cabecera puede incluir el cambio de idioma y se cargaría la misma cabecera en toda la aplicación. Como puede verse es posible e incluso recomendable usar una combinación de las tres.

Independientemente de cómo lo hagamos, por detrás siempre debe haber archivos properties que contienen las traducciones, tal como lo vimos en J2SE.

JSTL tiene un paquete adicional para la internacionalización llamado **i18n** que usa la directiva taglib (librería de etiquetas) `<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt_rt" %>` o bien `<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>`

Para usarlo primero se declara el fichero que se va a utilizar `<fmt:setBundle basename="${sesiónScope.idioma_preferido}" />` después donde vayamos a hacer la sustitución de texto según el idioma elegido se usa

`<fmt:message key="nombreDeLaClaveEnElProperties">`

En los botones, como el texto va en la propiedad value del tag “`<input type="button" value="Sí" />`” debemos guardar en una variable el valor del texto y sustituirlo usando una expresión EL en el value del botón. Para ello hay que definirlo como un parámetro de contexto en el archivo de configuración web.xml, donde el param name es la clase y el param value es el archivo properties, con lo cual hay que poner un parámetro de contexto por idioma. **Es posible que ni siquiera así funcione con lo cual habremos de buscarnos la vida para resolver la internacionalización de botones.**

DISEÑO DE LA WEB

Existen problemas para conseguir el mismo resultado por pantalla de nuestra web dependiendo del navegador con el que la abramos.

Para dar forma al diseño de nuestra página debemos saber HTML, no solo por el diseño o maquetación de la página sino por el extenso uso que haremos de los formularios <form>.

Así mismo la etiqueta <div> es de gran utilidad para poder administrar la posición, tamaño y aspecto de nuestra web. A través de las hojas de estilo CSS. De lo cual se deduce que cuanto mayor sea nuestro conocimiento de CSS mejor resultado obtendremos.

Las hojas de estilo se pueden relacionar con nuestra web de muchas formas, embebiéndolo en etiquetas, embebiéndolo en el <head> de nuestra web o llamándolos como archivos externos con extensión .css. Esta última opción es la mas recomendable por la facilidad de mantenimiento que ofrece, con lo cual es la que usaremos habitualmente.

Las hojas de estylo se llaman desde el <head> de la página con la etiqueta `<link rel="stylesheet" type="text/css" href="rutaAbsoluta" />`

La sintaxis para crear hojas de estilo es la siguiente:

```
Selector {  
    reglaXXX: valor;  
.  
.  
.  
}
```

Donde el selector es la etiqueta a la que se va a aplicar el estilo. reglaXXX es el nombre de la propiedad que vamos a asignar a ese selector, y el valor es como su nombre indica el valor a aplicar a esa propiedad.

HTML

Es un dcoumento xml. Lo tratan por DOM. A la hora de programar en javaScript nos permite interactuar directamente con el navegador.

Un documento HTML empieza y termina con <HTML> y </HTML>. En HTML 4 si hay etiquetas mal escritas se las salta, su sintaxis es muy permisiva. La pagina sale a delante ignorando fallos. No es una buena practica, la sintaxis debe ser estricta.

Una pagina HTML tiene dos partes body y head. Body suele reflejar el contenido de la web. El head no suele reflejarse, es optativo. Nosotros lo usaremos para cargar las librerias de estilo y el javaScript. La unica etiqueta visible en head sera title. Otra etiqueta de head es meta, que tendra informacion para los buscadores o almacenamiento de info. Tanto las librerias de estilo como el javaScript pueden estar embebidos en la pagina o en un fichero externo. Esta segunda sera la practica mas recomendable.

Para cargar estilos utilizaremos `<link href="ruta interna y nombre del fichero css" rel="stylesheet" type="text/css">`.

Para cargar JavaScript utilizamos `<SCRIPT type="text/javascript" src="ruta interna y nombre del fichero js">`.

Type: indicaremos el tipo MIME. Con lo que el navegador sabe como tratarlo.

Para tratar body utilizaremos solo etiquetas que tengan que ver con contenido, para estilo utilizamos solo css.

`
` Salto de linea

`<p>` Salto de linea antes y despues`</p>`

`` texto/imagen(llevaria dentro ``) vinculado `` hipervinculo

`<form action="donde" method="post">` formulario. Solo se puede enviar uno, el del submit pulsado. Tiene dos atributos action para indicar donde se tratará el form y method para indicar con que metodo se enviará.

`<table>` tabla. Es una forma rapida de colocar un formulario, aunque tambien se puede colocar con hojas de estilo. El form tiene dos etiquetas `<tr>` filas `<td>` columnas. Tambien tienen `<thead><tbody>`

```
<HTML>
  <HEAD>
    <LINK href="css" rel="stylesheet" type="Text/css" />
    <SCRIPT type="text/javaScript" src="ruta interna y nombre del
JS">
  </HEAD>
  <BODY>
    <BR/>
    <p></p>
    <table>
      <tr>
        <td>
        </td>
      </tr>
    </table>
  </BODY>
</HTML>
```

El Formulario

```
<form>
    <input type="componentes"      value="valor"
           name="nombre"      id="nombre unico">
</form>
```

Componentes:

Text: etiqueta

textArea: campo texto

password: no muestra caracteres

radio: tiene la propiedad checked

checkBox: tiene la propiedad checked

botones: ver sección botones

valor:

cualquier valor de caracteres, numéricos, etc.

Nombre: este será el nombre utilizado en el servidor. No poner dos iguales.

Nombre único: sirve para manejarlo con JavaScript

Botones:

Submit: Envía directamente una petición. El navegador borra la pantalla la deja en blanco y no repinta hasta recibir la respuesta.

Reset: limpia el formulario

Button: no tiene una funcionalidad definida por defecto.

Lista desplegable

El nombre de este componente lo tiene la etiqueta padre y los valores las etiquetas option. En el select podemos configurar también otras características como por ejemplo que se puedan mostrar varias líneas a la vez con size="" o que se puedan mostrar varias opciones multiple=""

```
<select name="lista desplegable"      size=""      multiple="">
    <option value="1"> xxxx </option>
    <option value="2"> yyyy </option>
    <option value="3"> zzzz </option>
```

Etiquetas div

La forma de aplicar un estilo, tamaño o posición a un componente es colocándolo en una etiqueta div. También se les llama cajas o capas. Al colocar las cajas podemos tener varios supuestos: que no se superpongan, que se superpongan parcialmente o totalmente. Las capas además de tamaño y posición, tienen profundidad. Se pueden colocar unas encima de otras y mostrar la que conviene. Esta práctica de dejar una detrás y mostrarla cuando se quiera sin necesidad de contacto con servidor se llama **patrón de precarga**.

Además de ancho y alto tienen un margen interno y externo de 5 pixels por defecto,

llamado **padding**.

Las capas por defecto no muestran el borde. Usaremos la propiedad border para modificarlo.

Para colocarles estilo tienen un atributo style="\"", pero no debemos usarlo. Los estilos deben ser externos. Es una mala practica.

Para aplicar un estilo hay que crearlos...

Estilos CSS

Los estilos estan formados por reglas (tipo fuente, tamaño, color).

La sintaxis de un estilo es :

```
xxxxx.css  
selector {  
reglas  
}
```

Selector es el atributo que va a decidir a que etiqueta de la pagina se le va a aplicar el estilo.

A un estilo se le pueden aplicar varios estilos. Pero si dos etiquetas le dan un tamaño de fuente a una misma div, esta tomará el valor de la segunda. Por ello hay que intentar que las reglas no sean concurrentes, que no se pisen unos a otros.

Hay documentación adicional sobre este tema en la carpeta de documentacion adicional.

Selectores CSS

Etiqueta

Hay diversos tipos de selectores, por ejemplo selectores de etiqueta, se refiere a las etiquetas HTML como puede ser div, H1, etc. Pueden usarse varios selectores a la vez, separados por comas. Se aplica a todos

```
div, p {  
}
```

Si no se separan por comas sino por espacio lo que indicamos es inclusión, por ejemplo el selector "div p" quiere decir que se aplica el estilo a todos los párrafos que haya dentro de un div, es decir por inclusión.

```
div p{ }
```

Clase

```
.nombre{  
}  
}
```

Las etiquetas pueden tener un atributo class, que puede aplicarse a mas de una etiqueta de la página, se especifica como selector usando el punto (.) antes del nombre de la clase, por ejemplo .nombreClase

ID

```
#nombre {  
}  
}
```

Las etiquetas HTML pueden llevar también un atributo id, en este caso ha de ser único en la página. Se especifica en la hoja de estilo usando el símbolo # antes de su nombre. Por ejemplo para quitar los margenes internos y externos de etiquetas seria:

```
div{  
margin: 0px;  
padding: 0px;  
}  
Border-style: varios tipos de linea  
Border-color:RGB(X,Y,Z)  
Border-width:5px
```

Posicionamiento:

```
# asfasdfasd{  
    Top: 20 px;  
    Left: 50 px;  
    Width: 50 px;  
    Height: xx px;  
    Position: absolute; (este valor es comun a todos los navegadores)  
    Visibility: visible/hidden;  
    z-index:5; (valor de profundidad)  
    overflow: visible/hidden/auto/Scroll; (barra de Scroll, por defecto hidden)  
}
```

Hay que entender que cuando la pantalla se pinta los estilos quedan definidos y no cambiarian hasta que yo modifique los atributos con javaScript.

En caso de conflicto de propiedades, es decir misma propiedad definida en mas de un selector u hoja de estilo, esto se resuelve por herencia, teniendo preferencia los estilos definidos por el autor de la página antes de los que son llamados por enlaces link, y si son todas llamadas prevalece el orden en que han sido llamadas. Lo deseable es evitar conflictos de estilos.

JavaScript

No es Java, no es orientado a objetos, no se compila, es muy aceptado y utilizado para manejar páginas web en el aspecto sobretodo de diseño. Aunque se puede embeder en la web igual que las hojas de estilo lo ideal es crear archivos con extensión .js y llamarlos desde la web con la etiqueta `<script type="text/javascript" src="rutaAbsoluta" />` una vez llamadas se guardan en cache con lo cual cualquier cambio en el archivo js para ver los cambios se ha de reiniciar el navegador. Para hacer debug necesitaremos sacar mensajes por pantalla.

Todas las palabras reservadas de JavaScript van en minúsculas.

Las funciones se declaran con la palabra reservada function seguido del nombre de la función. Entre llaves, el cuerpo. Los parámetros entre paréntesis a continuación del nombre. La devolución se hace usando return dentro del cuerpo de la función.

```
function nombreFuncion(x, y){  
    nombreVariable = valor;  
    x="15";  
    x="5";  
    x=null;  
    x=true;  
    y="7";  
}
```

La declaración de variables no exige declarar el tipo. Hay 5 tipos (número, cadenas de caracteres, null, y booleano). Un tipo también puede ser un objeto del navegador (que son tablas de mapas), un ejemplo de objeto es `x= new Date();` Opcionalmente puede usarse la palabra reservada var para hacer ver que lo que sigue es el nombre de una variable. La visibilidad de las variables es únicamente dentro de la función.

Utiliza los mismos iteradores y operadores de Java. Existe algún operador más que en Java, por ejemplo `==` (estrictamente igual) que es una comparación donde debe coincidir, no solo el valor sino también el tipo.

Las estructuras de control son iguales que en Java.

Las colecciones son tablas, es decir pares clave/valor. Que además son dinámicas, no hay que declarar tamaño. Se declaran usando corchetes, `nombreTabla[][][]...;` Al no haber tipos no se puede restringir el uso de distintos tipos en la misma tabla.

```
Var tabla= new Array();  
Tabla[5]= "x";
```

Con la propiedad `Tabla.length` y el for tendremos la estructura para recorrer tablas.

JavaScript funciona en respuesta a eventos, por lo tanto es en respuesta a eventos cuando se llaman a las funciones de JavaScript.

El código de JavaScript puede embeberse en el código HTML sin mas. Por ejemplo `<body onload="nombreFuncion();">` ejecutaria la función nombreFuncion al cargarse el cuerpo de la página.

Formularios:

En los formularios el evento onSubmit puede ser de gran utilidad antes de enviar el request a un servlet Java. También el evento onClick dentro de etiquetas <form> puede sernos muy útil para no sobrecargar al servidor con formularios mal llenados por parte del cliente, etc.

Otros eventos son:

onBlur = cuando el componente pierde el foco

onFocus = cuando el componente gana el foco

onMouseOver = cuando el raton pasa por encima del componente

onMouseOut = el cursor del raton sale del componente

onChange = cuando cambia el contenido de un componente

onSelect = cuando un componente tipo radioButton o combo cambia la selección

FORM Form		EVENTO onReset onSubmit
text textArea password		onBlur onFocus onMouseOver onMouseOut onChange onSelect
Button Reset Submit Radio checkBox select		onBlur onFocus onMouseOver onMouseOut onChange onSelect
Body		onLoad onUnload

Deben tenerse en cuenta las reglas que aplica cada navegador por lo cual es mas recomendable comprobar las paginas JavaScript en varios navegadores.

A través de JavaScript también podemos interactuar con la pagina. Los navegadores interpretan el HTML con el objeto DOM creando objetos (nodos) y disponiéndolos en forma de árbol. Con JavaScript podemos interactuar con estos objetos. El DOM ofrece métodos para acceder a sus nodos (elementos contenidos unos en otros), es decir al contenido de la página. Existen una serie de métodos que son compartidos por todos los navegadores y además cada navegador tiene sus particularidades.

La estructura (DOM) básica de objetos del navegador es la siguiente:

```

Navigator
  Window
    History
    Location
    Frames
    Document
      Anchors
      Applets
      Embeds

```

Para acceder a un elemento de un formulario se usaría un código como este `window.Document.form[0].Element[3]` si solo hubiera una ventana abierta se puede omitir el objeto window.

Como normalmente se trabaja con IDs entonces los objetos de la página tienen su nombre reflejado en esa ID. En este caso se puede acceder a sus elementos por su id con el método DOM **getElementById** por ejemplo así:

```
var x = Document.getElementById("IDdelElemento");
```

De esa forma podemos acceder a las propiedades del elemento a través de la variable a la que se le ha pasado el elemento, así:

```
x.value = "xxx";
```

o así:

```
var x = Document.getElementById("IDdelElemento").value;
```

el objeto document tiene además otros métodos interesantes como:

`x[] = Document.GetElementsByName("xxx");` → Obtenemos un array de elementos con el nombre indicado.

`X[] = Document.getElementsByTagName("tag");` → Devuelve un array de elementos del tipo determinado por el parámetro, puede ser div, p o cualquier etiqueta (tag) de HTML (div, p). Ordenados por la posición en la página.

A partir de la obtención de un elemento podemos usar el método **innerHTML** para insertar código HTML, como texto o cualquier otra cosa a la página de forma dinámica.

Objeto window

El objeto window tiene las siguientes propiedades:

Name – nombre de la ventana

Closed - Indica (Booleano) si la ventana ha sido cerrada

Length - Número de ventanas hijas (frames)

Self - El nombre del marco en el que estamos (focus)

Parent - El nombre del marco que contiene al marco en el que estamos (padre)

Top - El nombre de la ventana principal que contiene todos los marcos

Opener- La ventana (objeto window) desde la que se ha abierto nuestra ventana

Status - Referencia la barra de Status (abajo a la izquierda)

DefaultStatus – Texto a mostrar en la barra de Status cuando este no contiene nada.

El objeto Window tiene los siguientes métodos:

Alert("mensaje");	- muestra una ventana de diálogo con el mensaje pasado.
Confirm();	- muestra una ventana de diálogo
Prompt();	- muestra una ventana de mensaje
WriteLn("-");	- Escribe el texto indicado
setTimeOut("función()", milisegundos);	- Ejecuta una función cada x milisegundos.
clearTimeOut("función()");	- cierra el setTimeOut de la función dada.
Blur()	- Quita el foco a la ventana
Focus()	- Da el foco a la ventana
Close()	- Cierra la ventana
Scroll(x,y) Y	- Posiciona la pagina dentro del Navegador con la sposiciones X e Y
Open(url, nombre, características)	- Permite abrir una nueva ventana de navegador
Características (Yes/No):	
ToolBar	- Barra de Herramientas (Yes/No)
Location	- Barra de URL
Directories	- Bara del Historial y adelante/atras
Status	- Barra de status
MenuBar	- Barra de Menu
ScrollBars	- Barras de desplazamiento
Resizable	- Modifiable
Width	- ancho en pixeles
Height	- Altura en pixeles
Top	- Posicion respeto a la parte de arriba en pixeles
Left	- Posicion respect al lado izquierdo en pixeles

Objeto Location

Es la barra de Navegacion. Se refiere a la petición que nos ha traido a la página donde estamos.

Propiedades:

Hash
Port
Hostname
Host
Protocol
PathName
Search
Href

En una dirección o URL del siguiente tipo <http://localhost:8080/Pedidos/pedido.jsp?var1=valor1> http seria el protocol, localhost es el hostname, 8080 es el port, localhost:8080 es el host y Pedidos/pedido.jsp seria pathName, ?var1=valor1 es el search.

En caso de que estemos pidiendo un anchor (enlace interno) del tipo url#nombreanchor, el Hash seria ese anchor.

Href seria la URL completa.

Metodos:

Reload() - Recarga la página
Replace(url) - Envía al navegador a la url indicada, haciendo desaparecer del historial a la pagina desde donde se ha hecho el replace.

Objeto History

Métodos:

Back() - Una pagina hacia atras
Forward() - Una página hacia delante
Go(int)- Ir a la pagina indicada del historico

Objeto Navigator

Propiedades:

AppCodeName - Devuelve una cadena que indica el código de Navegador (no útil)
AppName - Nombre del navegador
AppVersion - Versión del Navegador
UserAgent - AppName + AppVersion
JavaEnabled() - Si se admiten applets (devuelve Boolean)
Plugins - Devuelve un array de plugins instalados en el navegador (no en IE)
MimeTypes - Array de tipos Mime soportados por el navegador

Librerias externas de JavaScript

Existen docenas de frameworks externos gratuitos y comerciales para ayudar en la implementación de javascript. Un ejemplo puede ser Prototype, Script.aculo (basada en prototype), jQuery, Motoools, etc. Hay también otras librerías que combinan javaScript y Java como DWR, DOJO, GWT, etc.

Prototype

Esta librería no trata con efectos visuales sino asuntos de programación.

AJAX

Es un tipo de petición al servidor asíncrona. Es decir sigue siendo una petición aunque no queda registrada en el navegador, con lo cual el usuario puede seguir trabajando con la página mientras llega la respuesta que esperamos (datos, no páginas) y que cambiara el contenido de la página.

Lo que cambia respecto a una petición normal es la forma como se hacen estas peticiones AJAX.

La desventaja es que todo el proceso ha de crearse a mano con JavaScript.

El objeto principal de este procedimiento es **XMLHttpRequest**. Por lo tanto es lo primero que hay que crear. La forma de hacer esto depende del navegador con lo cual hay que hacer una distinción del navegador para usar un código u otro dependiendo de eso. Lo normal es hacer un script (.js) aparte y usarlo para la conexión llamándolo desde la web cuando lo necesitemos.

Un ejemplo de clase XMLHttpRequest seria este:

```
var petición;
If (window.XMLHttpRequest) {
    petición = new XMLHttpRequest;
} else if(window.ActiveXObject) { // Navegador IE
    petición = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Con la segunda parte del if pedimos el httpRquest al sistema mediante el ActiveXObject con el nombre de la librería. El problema es que según la versión de IExplorer, el nombre de la librería cambia.

Una vez creada la conexión (objeto XMLHttpRequest), hay que hacer la petición. Se usa primero la función **open("tipo", URL, Boolean)**, siendo el tipo GET o POST, PUT, DELETE, URL es una dirección absoluta completa y bien formada que llama al servlet o script que se va a usar. El tercer parámetro booleano indica el sincronismo, si es true será asíncrona y si es false no lo será, con lo cual **True**.

Una petición get lleva los parámetros en la url. La url la calcularemos con dos funciones, una que calcule la ruta del servidor y la segunda que me calcule la petición tipo get, es decir con parámetros.

?=xx=xx&xx=xx&xx=xx

```
Petición.open("GET", URL, true);
Petición.send(null);
```

Si es con post, los parámetros irán en el cuerpo y tendré que construirlo.

```
Petición.open("GET", URL, true);  
Petición.send(cuerpo de petición);
```

Quien atiende la respuesta cuando vuelve es el XMLHttpRequest. Esto lo hace el navegador con el timer, preguntando si ha llegado cada segundo hasta que la recibe.

Después indicamos quién se va a hacer cargo de la petición cuando llegue. Se hace usando la técnica de eventos usando el método **onreadystatechange**. Algo como `petición.onreadystatechange = función;`, el nombre de la función sin paréntesis. Es importante colocarlo delante del **send**.

Por último se usa el método **send** para enviar la petición. El método send depende de si la petición es GET o POST. Si es GET el parámetro será null, si es POST el parámetro es el cuerpo de la petición POST.

```
petición.open("tipo", "url", true);  
petición.onreadystatechange = nombreFunction;  
petición.send("xxx");
```

La función a la que estamos llamando es la que se encarga de buscar la información que se está pidiendo por AJAX. Esta función tiene funciones y propiedades, una de esas propiedades es **readyState** con valores numéricos del 1 al 4, dependiendo de ese número sabremos el estado de la petición y así sabremos si ya ha llegado. Siendo:

- 0 – no se ha lanzado la petición
- 1 – se ha lanzado la petición
- 2 – no ha llegado la respuesta
- 3 – está llegando la respuesta pero no está completa
- 4 – Ha llegado la respuesta completa

1, 2 y 3 dependen del navegador. No en todos significan lo mismo

Cuando tenemos un código 4 sabemos que ha llegado la respuesta completa pero aún no sabemos si es la que esperamos con lo cual se hace necesario conocer el **código de status** de la respuesta http, 200 es la que queremos la cual refleja que todo ha sido OK.

Una vez tenemos los códigos 4 y 200 podemos usar el método **responseText**, **responseJSON** o **responseXML** para obtener los datos que han llegado.

Esto iría en un if anidado parecido a esto:

```
If (petición.readyState == 4) {  
    If (petición.status == 200) {  
        funciónProcesarRespuesta(petición.responseText);  
        Código...  
    }  
}
```

}

Ejemplo:

```
/** VARIABLE GLOBAL PARA LAS PETICIONES AJAX */
var peticion_http;
/**
 * FUNCION QUE REALIZA LAS PETICIONES DE LA CABECERA.
 */
function actualizar_cli_ajax() {
    cerrar_mensaje(false);
    if (window.XMLHttpRequest) {
        peticion_http = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        peticion_http = new ActiveXObject("Microsoft.XMLHTTP");
    }
    // Preparar la funcion de respuesta
    peticion_http.onreadystatechange = respuesta_cliente;
    // Realizar peticion HTTP
    var codigo = document.getElementById("cod_cliente").value;
    var nombre = document.getElementById("nombre_cliente").value;
    var url_peticion = crear_URL(1);
    peticion_http.open('GET', url_peticion + '&cod_cli=' + codigo +
    "&nom_cli="
        + nombre, true);
    peticion_http.send(null);
}
```

JSON (JavaScript Object Notation)

Es un formato standard de texto, util para serializar objetos. Su sitio oficial es <http://www.json.org>

Como ventaja adicional, la comunidad de desarrollo ha preparado las clases adecuadas para trabajar con este formato en todos los lenguajes de programación incluido Java. Con lo cual tenemos la mayor parte del trabajo hecho. Podemos encontrarlo en la web arriba mencionada.

Este formato es ideal para las comunicaciones via http.

Un objeto descrito en formato JSON empieza y termina con llaves {}. Cada uno de los valores y propiedades que tenga tendra un nombre de propiedad, dos puntos (:) y el valor de dicha propiedad, separados por comas puede haber tantos como se quiera. Si el valor es texto debe ir entre comillas dobles o simples, El valor puede ser un numero, entero o decimal, puede ser también true o false, admite también null. Tambien puede ser un valor múltiple como una tabla, en ese caso los valores van entre corchetes. Además permite anidarse, es decir el valor de una propiedad puede ser otro JSON, con sus llaves de inicio y fin y sus pares clave/valor separados por comas.

```
{“nombre propiedad”=valor ; masPropiedades = valores}, {otro objeto}, {otro objeto}  
Valor:  
“ ”  
-9.9  
Traq/false  
Null  
[]  
{---}
```

Las clases que se emplean para traducir el JSON de objeto a texto y de texto a objeto ya están hechas por la comunidad de desarrollo.

Si miramos la web oficial de JSON veremos que en el apartado de Java hay muchas opciones disponibles para usar estas clases. La utilizada por el profesor del master es **Json-lib**.

Desde el entorno de JavaScript las ventajas que nos ofrece JSON es que en JavaScript se puede simular la programación orientada a objetos. Con lo cual se pueden definir constructores, los objetos pueden tener herencia, métodos, propiedades, etc. De forma que, si serializamos un objeto desde Java, en JavaScript podemos hacer el proceso inverso convirtiendo el texto de JSON a un “objeto javascript” usando la función **eval(“objJSON”)**. Eval requiere que el parámetro que va entre paréntesis a su vez vaya también entre paréntesis y además el nombre de la propiedad entre comillas simples, por ejemplo: `eval('(` + variable + ')');`

Por ejemplo:

```
Funcion (objJSON) {
    var objJavaScript = eval('(' + objJSON + ')');
    // accederíamos a las propiedades usando el punto
    objJavaScript.propiedad;
}
```

Esto conlleva problemas de seguridad ya que si lo que se pasa al cliente es un código javascript maligno puede crear problemas de tipo virus en entornos windows.

Como hemos dicho, con JavaScript se puede simular el trabajo con objetos.

Para definir un objeto en JavaScript hay que definir un constructor para ello creamos una función y en ella definimos propiedades, se usa this para referirnos a esa función (clase). Las propiedades pueden tener o no un valor. Además las propiedades pueden definir otras funciones llamandolas por su nombre. Con lo cual un método (función) que defina las propiedades de un objeto seria un constructor.

Para instanciar una clase de JavaScript se usa la misma sintaxis que en java, usando la palabra reservada **new** que lo que hace es reservar memoria y además el navegador crea un objeto padre del cual copian todos los hijos, esa propiedad es la propiedad **prototype** y es accesible y modificable. Si modificamos la propiedad prototype de una clase modificaremos por lo tanto todas sus instancias (hijos). Con lo cual podemos tener control absoluto de todo lo que queramos incluido el navegador.

Como se puede ver, los objetos de JavaScript son simplemente arrays asociativos, también conocidos como mapas (pares clave/valor).

```
function xxxx() {
    this.propiedad1;
    this.propiedad2 = xxx;
    this.propiedad3 = nombreFuncion;

    var obj = new nombreFuncion();
    obj.nombrePropiedadMetodo();
}
```

Con JavaScript podemos tener en las funciones un numero variable de argumentos. Para hacer eso podríamos crear una función sin argumentos y cuando instanciamos un objeto de esa clase, además de crearse el prototype, también se crea otra propiedad dentro de ese objeto que es la que recibe todos los valores que se mandan cuando se crea el objeto. Esa es la propiedad **arguments** que es un array.

```
Function xxx() {
    // obtiene el valor
    var x = obj.arguments.length;
    // añadimos un Nuevo argumento a una clase
```

```

        obj.arguments[x + 1] = xxx;
    }
}

```

Como en JavaScript no existe el concepto de visibilidad tenemos control total, no necesitamos métodos accesores para acceder a las propiedades de un objeto y da igual si están en una carpeta u otra, basta con saber donde está.

Cuando se trabaja con objetos en JavaScript, para crear una clase se sigue este patrón:

```

var xx = {
    This.x:xxx;
    This.y:functionXXX() {
        Codigo javascript
        .
        .
    }
}

```

Para trabajar con JSON en MyEclipse hay que importar las librerías necesarias al proyecto. Las librerías necesarias están explicadas en la web de Json-lib (por ejemplo <http://json-lib.sourceforge.net/>) enlace getting started (<http://json-lib.sourceforge.net/usage.html>). Nosotros tenemos esas librerías en la carpeta JSON dentro de la documentación adicional.

Ejemplo:

```

/**
 * Recoge la respuesta cuando llega y lanza el procesamiento de la
peticion de los clientes.
 */
function respuesta_cliente() {
    if (peticion_http.readyState == 1) {
        document.getElementById("imagen_ajax_articulos").src =
"./imagenes/espera_ajax.gif";

        document.getElementById("imagen_ajax_articulos").style.visibility =
"visible";
    }
    if (peticion_http.readyState == 4) {
        if (peticion_http.status == 200) {
            document.getElementById("imagen_ajax_articulos").src =
"";

            document.getElementById("imagen_ajax_articulos").style.visibility =
"hidden";
            cargar_clienteJson(peticion_http.responseText);
        }
    }
}

```

Cuidado al serializar con JSON, porque los objetos de Hibernate no satisfacen la carga vacía, por tanto tendremos que poner las propiedades a null para poder serializarlo sin

que de un error.

Ejemplo:

```
public String convertir_Json(Clientes_Pedido cliente, String
mensaje) {
    String texto_salida = null;
    JSON objeto_json = null;
    if (cliente != null) {
        cliente.setVencimientos(null);
        cliente.setPedidos(null);
        cliente.setFormasPagoClientes(null);
        objeto_json = JSONSerializer.toJSON(cliente);
        texto_salida = objeto_json.toString();
        if (mensaje != null) {
            texto_salida = texto_salida.substring(0,
                texto_salida.length() - 1)
            + ", 'error_cliente':'" + mensaje +
        "}";
    } else {
        texto_salida = "{ 'error_cliente':'" + mensaje + "'}";
    }
    return texto_salida;
}
```

Struts

J2EE es una arquitectura de trabajo, da libertad total para implementarla a nuestro gusto. Con esto nació la necesidad de diseñar marcos de trabajo funcionales. Nosotros vamos a ver dos de ellos; Struts y JSF. Son independientes entre si.

Struts Clásico (versión 1) es la versión que vamos a ver en el master, no confundir con Struts 2 que lo único que tiene de parecido a Struts es el nombre.

Struts está basado en JSP con lo cual está quedándose obsoleto a partir de la versión 6 de J2EE. La versión 6 de J2EE no usa JSP sino JSF.

Struts solo se puede aplicar a proyectos J2EE.

Hasta ahora hemos trabajado con el patrón MVC. Struts lo que propugna es ampliar el modelo MVC haciendo que el controlador sea estrictamente eso sin contener nada de la lógica del negocio con lo cual se implementa una nueva capa para la lógica, esta capa estaría entre el controlador y el modelo y la vista.

Struts clásico solo trabaja con la vista y el controlador, no con el modelo. El modelo se deja para que lo manejen otras herramientas como Hibernate.

Struts nos da un controlador ya hecho que es un servlet, es la clase **ActionServlet**, que controla el flujo del programa de acuerdo a lo que le expliquemos en un archivo descriptor, evidentemente.

Lo que nosotros tenemos que programar es la lógica, esas clases de lógica para poder ser utilizadas por el ActionServlet han de heredar del ActionServlet (clase Action).

Struts 1 usa objetos de dominio a través de la clase **FormBean**, a partir de esta clase de dominio podemos usar las dos tareas mas clásicas que son **validaciones y conversiones** además de internacionalización.

Para la capa de presentación nos da un conjunto de etiquetas personalizadas para interactuar con todos esos recursos.

Para controlar, definir y construir todos esos recursos nos da un fichero de recursos habitualmente llamado **strut-config.xml**. Que además, se declara también en el web.xml.

En el web.xml definimos el ActionServlet como cualquier otro servlet, es decir con la etiqueta <servlet>.

```
<servlet>
  <servlet-name>xxxx</servlet-name>
  <servlet-class>org.strut.....ActionServlet</servlet-class>
```

```

</servlet>

<servlet-mapping>
    <servlet-name>mismonombre de arriba xxxx</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

```

A la misma altura del árbol de directorios que web.xml definimos el archivo strut-config.xml, gran parte de la funcionalidad de strut está en este fichero no en ninguna clase. Este fichero no solo declara los componentes sino también la forma como se relacionan entre ellos.

Este fichero descriptor lleva una etiqueta raíz <strut-config> y dentro va la definición de todos los elementos, muchos de estos son acciones.

```

<strut-config>

<action-mappings>
    <action path="/nombre de la acción" type="paquete y clase"
        name="nombre form-bean" input="ruta y nombre pagina error"
        validate="true/false">
        <forward name="nombrePagina" path="ruta + nombre pagina">
            <forward-message parameter="" type="">
        </action>
    </action-mappings>

    <form-beans>
        <form-bean name="mismo nombre del action path" type="paquete y
        clase" />
    </form-beans>

</strut-config>

```

Las etiquetas **action** definen las acciones y llevan atributos, unos son obligatorios y otros optativos.

Path → obligatorio y es el nombre de la acción, no la ruta

Type → obligatorio y es el nombre del paquete y la clase

Name → Si queremos cargar un objeto de dominio se lo indicamos con esta etiqueta dándole el nombre del objeto de dominio (form-bean).

Input → Si vamos a validar la información y se produce un error de validación podemos indicarle en esta etiqueta la página que devolverá de forma automática. Trayectoria y nombre de página.

Validate → Para hacer esas validaciones se usa esta etiqueta usando true como valor

Dentro de la etiqueta action se indica la/las pagina a donde ir después de ejecutarse la acción. Para eso se usan las etiquetas **forward**. En esta etiqueta **forward** el name y el path definen cosas al contrario de como se ha hecho en la etiqueta action es decir:

Name → nombre que le damos a la página

Path → trayectoria + nombre de la página

Los forwards también pueden declararse globales con la etiqueta <global-forwards>:

```
<global-forwards>
    <forward name="incorrecto" path="/jsp/login.jsp">
        <description>PAGINA INICIAL DE LOGIN. TAMBIEEN USADA EN LA
ACCION /login
        </description>
    </forward>
    <forward name="correcto" path="/jsp/menu.jsp" />
</global-forwards>
```

También se definen los form-beans como mapeo de clases de apoyo a las acciones, son optativos. Pueden estar compartidos por varias acciones, es decir, no son excluyentes.

Los **form-beans** son clases publicas que extienden la clase ActionForm, deben seguir las normas de JavaBean. Los métodos accesores **solo llevan parámetros de tipo String**.

```
<form-beans>
    <form-bean name="nombre formbean" type="paquetes y clase"/>

<form-beans>
```

La validación de formularios se realiza en dos fases, una primera llamada validación simple donde se comprueba que los datos escritos en el formulario siguen la forma correcta, por ejemplo que el código postal contenga 5 numeros, o que no haya campos vacíos, etc. Esto se hace en un FormBean, dentro del método **validate()**. Cuando se producen errores muestra la pagina especificada en el atributo input de la etiqueta action asociada con el form-bean.

Como hemos dicho, hay que sobreescibir el método **validate()** que hereda de ActionForm, es un método publico que devuelve objetos **ActionError** (clase de ActionForm) y recibe un parámetro ActionMapping y otro parámetro HttpServletRequest.

Si el ActionError esta vacio se entiende que se ha superado la validación si contiene algo se entiende que hay un error y no se ejecuta la acción sino que se ejecuta el **input**, es decir se redirige el flujo a la web que maneja el error.

En **error.add** debemos añadir una serie de parámetros que son:

El primero es un texto que identifica el texto del error que se ha producido.

El segundo es la clave de un archivo properties que mapea el texto de error que se mostrara al usuario.

```

public class xxx extends ActionForm {

    Private xx;

Metodos accesores con parametros de tipo String

public ActionErrors validate(ActionMapping, HTTPServletRequest)
{
    ActionErrors error = new ActionErrors();
    If(logica a validar){
        error.add("texto que identifica el error", new
ActionMessage("clave del properties de internacionalizacion que
contiene el mensaje de error"));
    }
    return error;
}
}

```

En la página JSP la etiqueta para mostrar el error sería algo así:

```
<html:errors property="nombreCliente"
bundle="${sessionScope.idioma_elegido}" />
```

Donde property contiene el nombre del elemento de formulario y bundle el idioma elegido para la internacionalización. El nombre del elemento del formulario va en un atributo property, por ejemplo para una caja de texto que case con el ejemplo anterior sería:

```
<html:text property="nombreCliente"></html:text>
```

Cuando se realiza validación en una clase de tipo Action, no en un FormBean, después de haber cargado el objeto ActionErrors con los errores detectados, y antes de hacer el return, hay que usar el método **addError()** para cargar en la petición (request) o la sesión (no conveniente) y que así este disponible y se puedan mostrar los errores. Este método addError necesita dos parámetros, uno es el request y el otro el objeto ActionError. Esto se hace de forma **diferente a cuando se hace en un FormBean**.

Después de que se realiza esta validación simple se pasa a realizar una validación más compleja donde se verifican cosas como que el usuario no está ya logeado desde otra máquina o que el usuario existe. Esta es una validación más compleja que requiere usar la lógica de negocio y usar la capa de modelo.

Es entonces cuando se pasan los datos al sistema para que sean procesados. Este procesamiento normalmente se conoce como lógica de negocio y consiste en tres tareas principales; **una segunda validación compleja, transformación de datos y Navegación**.

Es decir **después de** ejecutarse correctamente el método sobreescrito **validate** del

ActionForm, struts ejecuta el método **execute** de la clase Action especificada en el archivo struts-config.xml, dentro del atributo name de la etiqueta action se escribe el nombre de la clase ActionForm (form-bean) con la que esta enlazada esa acción.

La clase que define esta segunda validación debe ser una clase pública que se crea mediante la herencia de la clase Action. Cuando la clase ActionServlet llama a la clase Action se ejecuta el método **execute** (antiguamente perform). Este es un método público que **devuelve** un objeto **ActionForward** y que **recibe cuatro parámetros**; un objeto **ActionMapping**, un objeto **ActionForm**, un objeto **HTTPServletRequest** y un objeto **HTTPServletResponse**.

En el ActionMapping de struts-config.xml se recogen todos los sitios a donde puede llevar la acción.

El **ActionForm** es el objeto de dominio que puede ser null si no se ha definido. Hay que castearlo al objeto de nuestro interés.

El objeto que se devuelve **ActionForward** debe contener la información necesaria para que el ActionServlet sepa a qué página ir una vez concluida la acción. Usando el método **findForward()**.

El **ActionForm** tiene otros métodos como **reset()**, que se corresponden con las etiquetas de struts html:submit, html:reset y html:cancel. Las tres etiquetas se corresponden con el botón de formulario de tipo submit. La etiqueta **html:reset** se corresponde con el método **reset()** del ActionForm, y lo que hace es limpiar la información de los campos del formulario, que está en el servidor dentro del formbean. El **botón Cancel** también es un botón de tipo submit, es decir envía la orden al servidor para que no llame al objeto de dominio, ni valide la información. La lógica del botón Cancel la escribimos nosotros dentro del método **validate()** del formbean. Esto se detecta dentro del request (**getParameter()**), que es uno de los parámetros que se pasan al método validate().

Las etiquetas de Struts las veremos más adelante.

Además estas clases Action carecen de estado, es decir NO llevan variables de instancia, sí pueden llevarlas dentro de una función.

```
public class xxxx extends Action {  
  
    public ActionForward execute (ActionMapping mapping,  
        ActionForm x, HttpServletRequest, HttpServletResponse) {  
  
        miclase obj = (miclase)x;  
        ActionForward salida = new ActionForward();  
        .  
        .  
        .  
        salida = mapping.findForward("nombre forward donde queremos  
        dirigirlo");  
    }  
}
```

```

        Return salida;
    }

}

```

Como dijimos antes, estas clases Action representan la lógica de negocio y es una capa intermedia entre el controlador y el modelo. Es donde se lleva a cabo la segunda fase de la validación que tiene tres pasos que explicamos más detalladamente ahora:

Validación compleja: Consiste en comprobar que el identificador de usuario existe.

Transformación de Datos: Consiste en guardar el identificador del usuario y la contraseña en la base de datos.

Navegación: Consiste en visualizar la página de registro si el registro fue correcto o el formulario con los mensajes de error si el identificador ya existe.

Struts solo trabaja con cadenas de caracteres.

La clase **Action** tiene algunas subclases que pueden ser muy útiles para realizar algunas implementaciones. Por ejemplo la subclase **LookupDispatchAction**, que es adecuada para que dentro de una misma clase de tipo Action quisieramos realizar varias acciones diferentes, como podría ser el escenario típico de altas, bajas, modificaciones, consultas, etc. Esta clase tiene un método llamado **getKeyMethodMap** que devuelve un Map, debe tener visibilidad **protected**. Esta clase lo que hace es que, dependiendo de la cadena de texto que le llegue, usará uno de los elementos del Map, ejecutando de esa manera un método distinto del execute para tratar la petición que haya llegado. Un ejemplo de código sería:

```

public class xxxx extends LookupDispatchAction{
    protected Map getKeyMethodMap() {
        Map <String, String> yyyy = new HashMap<String,
String>();
        yyyy.put("nombreboton", "nombreMetodo");
        yyyy.put(".....");
        Etc.
        return yyyy;
    }

    public ActionForward nombreMetodo (ActionMapping x,
ActionForm y, HttpServletRequest Req, HttpServletResponse Res) {

        //Código de la lógica del método
    }
}

```

El método **nombreMetodo** sería definido dentro de la misma clase para poder ser ejecutado según el botón que haya sido pulsado en el formulario de la página JSP de donde venga la petición. Esos métodos han de tener la misma firma que tendría un

método execute de una clase tipo Action normal y recibir los mismos parámetros es decir, han de ser public, devolver un Actionforward y recibir un ActionMapping, un ActionForm, HttpServletRequest y un HttpServletResponse. La única diferencia con un método execute sería precisamente el nombre.

En el fichero strut-config.xml la etiqueta action, además de las etiquetas ya conocidas como path, type, name, input o validate tiene otro parámetro llamado **parameter** que contendrá la cadena de texto que el Map del método **getKeyMethodMap** puede recibir para saber que método debe ejecutar. Este valor de parameter será el nombre del botón submit del formulario en la pagina jsp. Con lo cual podemos así diferenciar si se trata de un alta, o una baja, etc. El problema esta que en la internacionalización tendremos que usar como nombre del botón la clave del fichero properties para que esto no varíe en función del idioma usado por el usuario.

Otra subclase de Action muy útil además de LookupdispatchAction es **MappingDispatcherAction**, se emplea para resolver el escenario distinto, es decir que **desde varios formularios diferentes queramos usar la misma Accion**. En esta clase tenemos que crear todos los métodos públicos que queramos que devuelvan ActionForward y reciben ActionMapping, Actionform, Request, Response. Igual que los métodos Execute de Action.

Para diferenciar qué método hay que emplear dependiendo del formulario empleado, en el strut-config.xml dentro de la etiqueta action, además de los parámetros habituales usamos también parameter, pero en este caso, es el nombre del método a ejecutar.

Un ejemplo de código seria:

```
Public class xxx extends MappingDispatcherAction {  
    Public Actionforward xxxx (ActionMapping, Actionform,  
    Request, Response) {  
  
    }  
}
```

Un ejemplo de caso en que esta subclase seria útil es cuando tenemos una opcion de imprimir en los formularios de nuestra aplicación.

Internacionalización con Struts

Struts es muy útil para internacionalizar la aplicación, utiliza la estrategia de internacionalización que ya vimos, usando ficheros properties, tantos como queramos.

En el archivo struts-config.xml se define en la etiqueta message-resources, por ejemplo:

```
<message-resources parameter="com.atrium.idiomas.textos_es"  
key="es" />  
<message-resources parameter="com.atrium.idiomas.textos_en"  
key="en" />  
<message-resources parameter="com.atrium.idiomas.textos_fr"  
key="fr" />
```

La clave esta en la terminación del nombre del fichero properties. Si no queremos internacionalizar y por lo tanto usamos un solo archivo property no necesita llevar la terminación del idioma que seria el idioma por defecto. Si usamos varios idiomas hay que decirle cual ha de usar. **El nombre se indica en el atributo key del properties.**

Etiquetas Personalizadas de Struts

Struts utiliza sus propias etiquetas personalizadas para crear la vista. Han de ser declaradas en un taglib al igual que cualquier otra etiqueta usando JSP `<%@ taglib uri="" prefix="" %>` My Eclipse nos incluirá este taglib automáticamente. Al igual que otras etiquetas de JSP usa sus archivos TLD.

Una de las etiquetas mas usadas es HTML, contiene las etiquetas que tienen que ver con los formularios. Algunas de las mas importantes y usadas son:

<html:...

Formularios:

Form

Para textos:

Text
Password
Textarea
Hidden

Selectores:

Checkbox
Multibox (multiples Checkboxes relacionados)
Radio (varios radios con el mismo nombre son excluyentes, selección única)
Select (es un listbox, sus elementos son los option/options)
Option/OptionCollections (se usa OptionCollections cuando se generan a partir de una consulta a una BD, tiene un atributo value que es el atributo value de la etiqueta option del select en el código HTML, que va con otros dos label y property y que contienen la clave y el valor de dicha consulta. La clave es el valor que se envía para poder ser procesado y el valor es lo que se muestra en el select.)

Botones:

Submit
Reset
Cancel
Button

Archivos:

File
Image

Otros:

Errors

Base. Esta etiqueta dà la trayectoria hasta la página, no hasta el proyecto, como se hacia

en HTML. Para hacerlo hasta el proyecto hay que definirla así:

```
<html:base ref="site" />
```

Img

Link: es para crear hipervínculos

No hay que olvidar que son etiquetas de struts y no de HTML, por lo tanto siguen las reglas del TLD y no otras reglas. Las etiquetas de struts no **tienen tienen ni name ni ID, lo que tienen es propertie**, lo cual es lo que da valor en el código HTML al name y al ID. Que además se corresponderá con el nombre de la propiedad en el formBean.

Las clases formBean no las escribe MyEclipse, con lo cual o las escribimos a mano o **usamos el truco** de usar la clase de persistencia creada por hibernate modificándola para que extienda Actionform y sobreescribiendo el método validate().

Las etiquetas tienen una serie de atributos dependiendo de la etiqueta pero todas tienen unos atributos comunes que son:

Property (sustituye al name y al ID)

On... (eventos de JavaScript que llaman una función de JavaScript como en HTML)

Style: aplica un estilo como se hace en HTML con la etiqueta style

StyleClass: Le da nombre a la etiqueta class de un estilo

StyleID: Le da nombre a la etiqueta ID para ser reconocida como tal por una CSS y por JavaScript.

Bundle: El atributo key del message-resourcer que hayamos especificado en el strut-config se refiere al archivo del que va a leer para la internacionalización.

El bundle seria algo como “es” o “en”, etc, se puede usar una expresión EL.

Otros atributos no comunes entre las etiquetas de Struts son:

Action. Se usa en la etiqueta para un formulario <html:form ...>. Llama a una clase de tipo Action, es lo que hayamos puesto en el atributo path, de la etiqueta action en el strut-config, sin la barra y sin la terminación .do.

Link. Esta etiqueta de HTML es para crear hipervínculos y lleva los atributos action para llamar acciones y forward para llamar a un forward según lo hayamos definido en el strut-config. Tambien puede usarse con el atributo href como se hace en HTML.

A parte del paquete HTML de las etiquetas de struts existe el paquete **bean**, de esta usaremos la etiqueta message para sacar textos.

```
<bean:message key="clave del property" bundle="..." />
```

Existe tambien el paquete **logic** que sirve para incluir bucles tipo for o while, etc. Pero es mas adecuado usar JSTL. La que si se usa es la etiqueta forward, que llama al mapeado que se corresponde con un archivo al que desviaremos el flujo, según lo hayamos definido en el strut-config. Sirve para levantar el contexto, es decir para definir

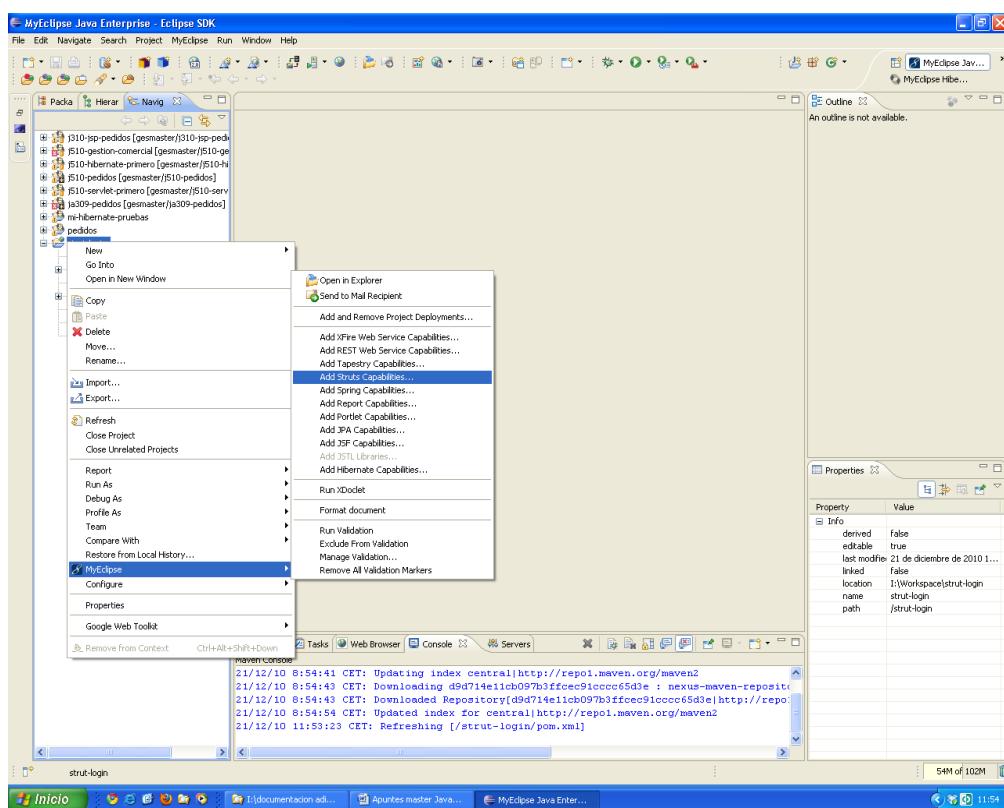
el ActionServlet.

```
<logic:forward name="nombre forward" />
```

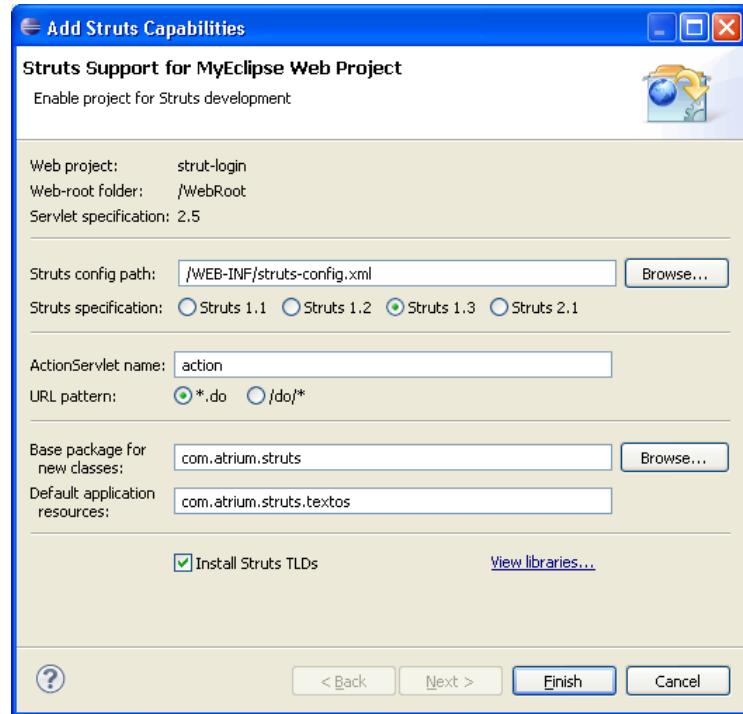
La versión de struts que monta Eclipse Galileo es la 1.3.10.

Instalacion de Struts en MyEclipse

A partir de un proyecto web que tengamos creado damos botón derecho en el proyecto – MyEclipse – Add Struts Capabilities



Aparece esta ventana:

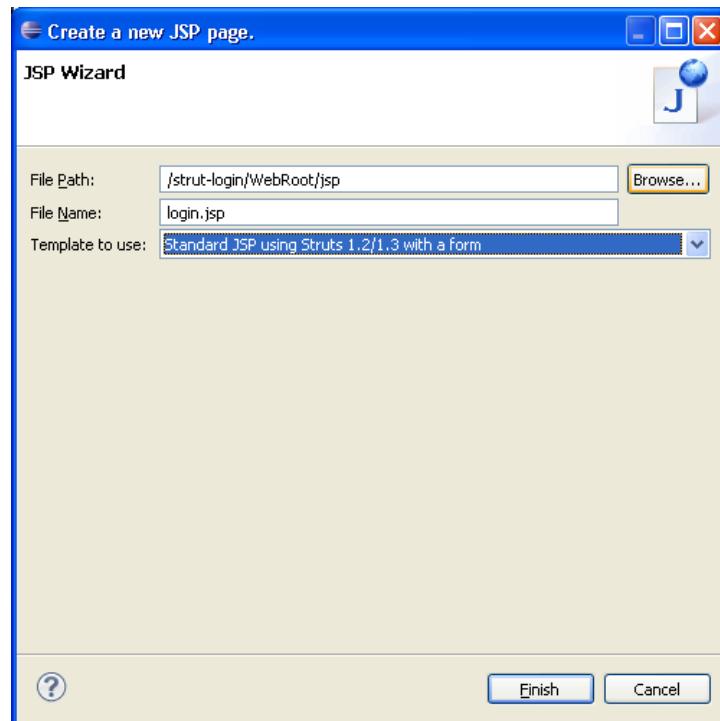


Indicamos la versión de struts, el nombre del ActionServlet, el patrón de la extensión de las URLs, el paquete donde iran las nuevas clases y donde iran los archivos properties y su nombre. Se marca la casilla “Install Struts TLDs”.

Pulsamos Finish. Al hacer esto MyEclipse añade los paquetes de Struts, y otros paquetes necesarios para las clases de struts.

Se crea también un fichero struts-config.xml con las etiquetas vacias a la espera de que las completemos nosotros. Lleva una pestaña de diseño que va dibujando el grafico de la navegación del sitio. Tambien mapea el archivo web.xml para incluir la estructura de struts y asi asegurarnos de que la configuración esta bien mapeada ya que puede ser un poco liosa..

Mas adelante cuando vaya a crear por ejemplo una pagina de login podemos usar uno de los modelos que ya nos ofrece para la capa de presentación (vista), creariamos una carpeta jsp en el webroot, y despues dando con botón derecho. Sobre ella elegimos New – JSP (Advanced templates) y sale esta ventana:



Aquí le damos nombre y como plantilla elegimos la que necesitemos que probablemente será la elegida en la imagen de arriba.

Esta web usa etiquetas de Struts en su código importadas con taglibs. En esas etiquetas la propiedad property es la que se usa como enlace con las etiquetas del form-bean.

Como la web arranca en index.jsp esta hay que redirigirla usando una etiqueta de struts usando el taglib `<% taglib uri="http://struts.apache.org/tags-logic/ prefix="logic" %>` y después la etiqueta es `<logic:forward name="nombre forward definido en sruts-config.xml" />` el forward debe estar definido exclusivamente para esto o un forward global que sirva para muchas acciones. Con lo cual en la etiqueta global-forwards del archivo xml struts-config la definiríamos así:

```
<global-forward>
  <forward name="nombre pagina" path="ruta y nombre pagina de inicio">
    <description>descripción del forward, opcional</description>
  </global-forward>
```

Tiles

Se encarga de la maquetación de las páginas Web, dentro de Struts. Es un marco de trabajo que se encarga solo de eso. Puede usarse con etiquetas dentro de la página o bien con ficheros de configuración XML, que nos da alguna opción mas.

Fue desarrollado originalmente por un freaky francés, mas tarde paso a ser un proyecto de Apache, abierto y comunitario.

Se puede usar sin Struts, aunque se considera como conocimiento avanzado de Struts. Struts ya lo incluye, con lo cual al incluir Struts en Eclipse como vimos en la sección anterior ya va incluida una versión de Tiles que suele ser adecuada. Solo si quisieramos cambiar la versión de tiles a usar es cuando deberíamos hacer una importación manual descargando la versión de Tiles que deseemos desde la web de Apache.

Para montarlo a través de páginas JSP, requiere la realización de una serie de páginas, dentro de las cuales usaremos etiquetas personalizadas de Tiles.

Hay que hacer una primera página JSP que será una plantilla, algunas plantillas ya están creadas. En esta plantilla hay que definir las zonas de la página, no se define el contenido, solo las zonas de la misma. Tiles solo se encarga de establecer las zonas que va a tener y como se van a llamar. Para esto se usan etiquetas div y estilos.

Después de definir la plantilla se definen otras páginas JSP que serán el contenido de las zonas definidas en la plantilla. Una plantilla puede tener tantos contenidos como nos interese. Esos contenidos se definen con mas etiquetas personalizadas de Tiles.

Para ponerlo en marcha, a nivel de procesamiento a lo que se llama es al contenido, no a la plantilla. **El contenido es el que llama a la plantilla que va a llenar**. Se usan las etiquetas **forward** de Struts para esta llamada.

Dentro de la plantilla para definir zonas se usa la etiqueta `<tiles:insert ... />` y lleva el atributo **name** que es el nombre que le damos a esa zona. Debe ser un nombre único para esa plantilla.

```
<div>
<tiles:insert name="nombreZona" />
</div>
```

Para definir el contenido usamos también la etiqueta `<tiles:insert ... />` pero esta vez con el atributo **template** que es el nombre de la plantilla que vamos a llenar de contenido. Se incluye la ruta y el nombre de la página JSP a través de la etiqueta `<tiles:put ...>` con los atributos **name** y **value**. Para indicar el nombre de la zona y su contenido.

```
<tiles:insert template="ruta_y_nombrePlantilla.jsp" >
```

```
<tiles:put name="nombreZona" value="ruta y contenido.jsp">  
</tiles:insert>
```

Evidentemente hay que definir el contenido para cada página. En la etiqueta value de los put podemos usar expresiones para ir variando dinámicamente el contenido.

En nuestro proyecto del master, dentro de WebRoot/jsp podemos crear nuevas carpetas para las plantillas y los contenidos separadamente. Las plantillas, como ya hemos dicho son páginas JSP desde las que se llaman a los estilos y se establecen los divs de la estructura de la página, según nos convenga, dentro de esos divs van las etiquetas <tiles:insert ... /> que vayamos a necesitar. Y eso es todo de lo que consta una plantilla.

Las hojas de estilo especifican las posiciones y tamaños de las zonas, además de las otras características propias del estilo que vayamos a aplicar a nuestras páginas.

Las páginas de contenido para contenidos dinámicos lo que tienen son las etiquetas de tiles:insert y tiles:put donde, en los atributos value de los put se escribe el código de las llamadas dinámicas al contenido real. Este código normalmente será una expresión EL y lo que hace básicamente es leer variables de sesión (sessionScope) que serán diferentes dependiendo del enlace que ha sido pulsado para ser llamado al contenido. Ver en el proyecto que hicimos en el master llamado j510-strut-login.

Para usar Tiles mediante ficheros XML se usa la misma lógica inicial de uso de plantillas y contenidos, pero el contenido parte de un fichero XML, con lo cual llamamos a un componente llamado “definition” contenido en uno o más ficheros de tipo XML. Con lo cual en la etiqueta tiles:insert usaríamos un atributo definition. Algo así:

```
<tiles:insert definition="archivo.def">
```

El archivo definition es un archivo XML que lleva una etiqueta <tiles-definitions> y dentro una etiqueta <definition> con los atributos **name** y **path**, además dentro de las etiquetas definition van etiquetas <put> con atributos **name** y **value** para la sección de la plantilla y su pagina jsp de contenido.

Para leer estos archivos XML hay que usar una clase de Java que puede ser añadida al entorno de Struts como un plug-in añadido. Para ello hay que modificar el archivo struts-config para añadir este plug-in al final del strut-config:

```
<plug-in classname="org.apache.struts.tiles.TilesPlugin">  
  <set-property name="definitions-config" value="/xml/definitions.xml" />  
</plug-in>
```

Cuando hay mas de un archivo XML se separan por comas en el value de la etiqueta set-property. En nuestro proyecto j510-strut-login hemos montado la parte de pedido usando estos archivos XML.

JSF

Es un marco de trabajo para aplicaciones J2EE. Es obligatoria a partir de J2EE 6, ya que JSP queda obsoleto a partir de esa versión. La versión que vamos a ver en el curso es la 1.2.

JSF de principio tiene una filosofía de desarrollo que es la contraria a JSP. JSP se desarrolló al inicio de J2EE cuando había pocos desarrolladores web y muchos diseñadores web, con lo cual se inspiró en la idea de acercar Java al mundo Web. Por el contrario JSF, como se ha visto que eso no ha funcionado, decide escribir páginas JSP lo más parecido posible a Java. Por ejemplo, un recurso de JSF son los eventos de usuario, como en J2SE. JSF es tecnología de servidor, pero permite generar código JavaScript al compilarse.

JSF pretende desarrollar aplicaciones Web de forma parecida a como se construyen aplicaciones locales con Java Swing, AWT, SWT o cualquier otra API similar. En definitiva se trata de hacer aplicaciones Java en las que el cliente no es una ventana de la clase JFrame sino una página HTML.

JSF constituye un marco de trabajo de interfaces de usuario del lado servidor para aplicaciones Web basadas en tecnología Java y en el patrón MVC.

JSF lo que hace es controlarlo todo desde el lado servidor, incluida la vista, ya que genera objetos por cada etiqueta JSF que se incluya en el código de las páginas (componente).

JSF combina un tratamiento de las peticiones junto con un sistema de vistas y de objetos de dominio. Añade cosas que no son consideradas por otros marcos de trabajo como Struts. Por ejemplo realiza conversiones de tipos automáticamente. Es decir cubre el ciclo de vida completo. Es un framework más completo y rico que la mayoría. Tiene un punto de extensión (hot spot) que permite desarrollar componentes personalizados. Con lo cual ya existen muchos componentes de terceros como, MyFaces, Trinidad Tobago, tomaHawk, estos de Apache. JBoss tiene RichFaces. Todas ellas gratuitas. Una muy interesante es la de **IceFaces**, que tiene una librería de componentes llamada IceFaces. Hay otras opciones comerciales de pago.

Muchos de estos componentes controlan totalmente la parte de cliente incluyendo las conexiones ajax para realizar componentes rich, y los estilos. IceFaces hace todo eso. A parte de que la integración con Eclipse es total.

JSF es muy flexible, permite por ejemplo, crear nuestros propios componentes, y/o crear nuestros propios renderizadores para pintar los componentes en la forma que más nos convenga.

Una de las grandes ventajas de la tecnología JavaServer Faces es que ofrece una clara

separación entre el comportamiento y la presentación. Puede, por ejemplo mapear una petición HTTP con un manejo de eventos específicos de los componentes o manejar elementos UI como objetos con estado en el servidor.

JSF incluye una librería de etiquetas JSP personalizadas para presentar componentes en una página JSP, las APIs de JavaServer Faces se han creado directamente sobre la API JavaServer, esto permite crear nuestros propios componentes personalizados directamente desde las clases de componentes y generar salida para diferentes dispositivos cliente, entre otras.

En definitiva, la tecnología JSF proporciona una rica arquitectura para manejar el estado de los componentes, procesar los datos, validar la entrada del usuario, y manejar eventos.

JEE5 → JSF 1.2 → 1.8x
JEE6 → JSF 2.02 → 2.01

Ciclo de Vida JSF

Las peticiones llegan a partir de una página web. Cuando se solicita una página que puede ser jsp o no (jspx o xhtml), JSF genera la página html que le enviará al cliente, pero además genera un árbol de objetos que representa esa vista (la página html que recibe el usuario), este es un proceso automático de JSF y se almacena a la espera de que llegue una petición.

Existen tres posibles escenarios que se pueden dar, estos son:

Escenario 1: Una petición No-Faces genera una respuesta Faces. Desde una pagina HTML se pulsa un enlace a una página con componentes JSF.

Escenario 2: Una petición desde una pagina JSF que genera una pagina No-Faces.

Escenario 3: Una petición Faces genera una respuesta Faces.

Cuando llega una petición del tipo del escenario 3 es cuando se inicia el ciclo de vida explicado a continuación:

1. **Reconstrucción del árbol de componentes:** Se recomponen el árbol de objetos de la vista, si está en memoria se usa y si no lo está se vuelve a generar. También conecta los manejadores de eventos y los validadores y graba el estado del FacesContext. (1º CREAR / RECUPERAR VISTA)
2. **Aplicar valores de la petición:** JSF recoge los valores que llegan en la petición y los carga en los componentes. Es decir en los objetos que representan a esos

elementos (formulario) que existen en la vista. En este paso pueden generarse eventos que se tratan cerrando el ciclo de vida y devolviendo la página inicial al cliente. Como cuando se producen errores en una validación de formulario. Si no se producen eventos se pasa a la tercera fase. (2º CARGAR PARÁMETROS EN COMPONENTES)

3. **Conversión de Datos:** Los valores recibidos que son siempre strings se tratan convirtiéndolos primero en el tipo adecuado si es necesario, de forma optativa. La conversión se realiza a partir de lo que nosotros hayamos programado. En esta fase se pueden dar eventos también, en cuyo caso se devuelve la pagina original al usuario. Si no hay eventos en la conversión se pasa a la fase 4. (3º CONVERTIR)
4. **Validación de Datos:** En esta fase se valida optativamente la información usando las reglas de validación que hayamos escrito en nuestro código. También pueden producirse eventos, en cuyo paso se vuelve a enviar la página de origen. Sigue siendo una validación sencilla de los datos del formulario. (4º VALIDAR)
5. **Actualización de Valores del Modelo:** En esta fase se carga el objeto de dominio que va a contener la información que nos ha llegado en la petición. La información se carga en los ManageBean, que son los objetos JSF que contienen la información y la lógica del tratamiento. JSF junta los datos y la lógica, al contrario que hacia Struts. Esta carga de datos también puede producir errores que se cargan en eventos y en ese caso se sale enviando la página de origen. (5º CARGAR MANAGEDBEAN)
6. **Generación de la Respuesta:** En la sexta fase es donde se ejecuta la lógica, el proceso que nosotros queramos aplicar a la petición. Si existieran eventos de cliente se tratan entre la fase 5 y la 6. Aunque eso puede cambiarse para que esos eventos de cliente se procesen antes de la fase 2. Una petición puede ser procesada atendiendo antes a los datos que a la lógica o viceversa. Una vez que se ha ejecutado la lógica se genera una nueva página y con ella un nuevo árbol de objetos y el ciclo de vida vuelve a comenzar. (6º EJECUTAR LÓGICA)

La diferencia fundamental entre lógica (acción) y eventos es que los eventos siempre devuelven la página de origen y la lógica tiene navegación para permitir presentar una nueva página después de ser procesada. El tratamiento de eventos, se realiza tras el paso 5 y antes del 6.

Estos seis pasos son los que componen el ciclo de vida de JSF y debe ser comprendido y memorizado completamente para poder trabajar sin problemas con JSF.

En principio el código que nosotros escribimos son páginas jsp y los ManageBean con la lógica de los procesos. Además debe tener un fichero descriptor con los componentes que vamos a utilizar este fichero normalmente se llama **faces-config.xml**. En este fichero solo se declaran los componentes no se establecen sus relaciones, como se hacia en Struts. Con lo cual permite hacer un uso mas libre y mantener un acoplamiento lo

más bajo posible.

faces-config.xml

Como hemos dicho, en este archivo definimos todo lo que necesitamos. Básicamente lo que vamos a definir son los ManagedBean, que son clases que incluyen datos y acciones.

Los ManagedBean se definen con la etiqueta **<managed-bean>** y habrá tantos como componentes de este tipo vayamos a necesitar. En el cuerpo de estas etiquetas tenemos otras tres etiquetas que son, **<managed-bean-name>**, **<managed-bean-class>** y **<managed-bean-scope>**, que representan el nombre del ManagedBean, el nombre de la clase donde está el ManagedBean, y el contexto (visibilidad) en el que se va a guardar y por lo tanto su duración, respectivamente (request, session, application).

Otro componente importante que se define aquí son las reglas de navegación (navigation rule), es decir una vez que se maneja un ManagedBean, a donde vamos a ir. Se usa la etiqueta **<navigation-rule>** y va referido a cada página, no a cada componente.

Navigation-rule lleva las etiquetas **<form-view-id>** que lleva la ruta y el nombre de la pagina jsp a la que aplicamos estas reglas de navegacion, es la **ruta interna a partir del webroot**.

A continuación lleva la etiqueta **<navigation-case>** para indicar las páginas a las que nos dirigiremos, que como pueden ser mas de una, dependiendo del caso, pues habrá tantas etiquetas navigation-case como páginas a las que nos podemos dirigir. Esta lleva las etiquetas **<from-outcome>**, que es el texto que devuelve el método de la lógica y **<to-view-id>** para especificar la ruta y el nombre de la página a la que iremos dependiendo del from-outcome recibido.

Más adelante se explica, con más detalle, la navegación en las páginas JSF.

Tanto Managed-bean como Navigation-rule pueden llevar una etiqueta opcional **description** para añadir comentarios que describan el asunto. Por ejemplo:

```
<managed-bean>
    <description>Breve descripción sobre este
ManagedBean</description>
    <managed-bean-name>NombreBean</managed-bean-name>
    <managed-bean-class>claseDelBean</managed-bean-class>
    <managed-bean-scope>ambito</managed-bean-scope>
</managed-bean>

<navigation-rule>
```

```

<description>Descripcion de reglas de Navegacion de la
pagina</description>
<from-view-id>paginaOrigen</from-view-id>
<navigation-case>
  <from-outcome>salidaMetodo</from-outcome>
  <to-view-id>paginaDestino</to-view-id>
</navigation-case>
</navigation-rule>

```

Gracias a este fichero se genera el árbol de objetos a partir de las peticiones del cliente que recibe JSF.

web.xml

Además, en el **web.xml**, hay que definir el **FacesServlet** como cualquier otro servlet, con las etiquetas servlet (servlet-name, servlet-class, load-on-startup) y servlet-mapping (servlet-name y url-pattern). Por ejemplo:

```

<servlet>
  <servlet-name>nombre Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>mismo nombre de arriba Faces Servlet</servlet-
name>
  <url-pattern>*.faces</url-pattern>
<servlet-mapping>

```

Por supuesto se pueden aplicar filtros o escuchadores.

ManagedBeans

Son las clases que incluyen los datos y la lógica que vamos a aplicar a una determinada petición. Son clases normales, es decir `public class xxx`.

La forma de enlazar esta clase con las páginas .jsp, se hará mediante el UEL(una evolución de EL que emplea \$ en vez de # y que tiene como ventaja, que me permite llamar a métodos).

Siguiendo las normas de JavaBean, debe llevar propiedades privadas y métodos accesores, además debe llevar otros métodos para la lógica y si es necesario los métodos para la lógica de los eventos.

Los **métodos de acción**, para la lógica, han de seguir estas normas: Métodos públicos, han de devolver un String, y no pueden recibir ningún argumento. Es decir `public String nombreMetodo() {}`

Los ManagedBean también incluyen los **métodos de los eventos**. Que serán públicos, devuelven nada (void) y reciben el escuchador como por ejemplo ActionEvent. Es decir `public void nombreEvento(ActionEvent evento) {}`

El método que hay que ejecutar se indica en la página jsp. Es decir, desde los formularios de las páginas se llama a los métodos de los componentes en el ManagedBean.

Cuando desde una página se llama a una propiedad de un ManagedBean desde una expresión, por ejemplo `#{{usuario.nombre}}`, JSF lo que hace es ejecutar el método **accesor getNombre()** para recibir el valor de esa propiedad. No accede directamente a la propiedad.

Los ManagedBean han de ser declarados, como ya explicamos en la sección del faces-config.xml. Para poder así ser reconocidos por nuestra aplicación JSF. En esa declaración una de las cosas que se especifica es el ámbito, concretamente en la etiqueta `<managed-bean-scope>`. Hay cuatro ámbitos que pueden ser declarados, que son, **session, application, request, o none**.

Además de lo visto en la sección de faces-config.xml, **también podemos configurar las propiedades del ManagedBean** en el fichero descriptor usando la etiqueta `<managed-property>`, anidada dentro de la etiqueta `<managed-bean>` para declarar e iniciar propiedades del ManagedBean. El nombre del atributo se inicializa con la directiva `<property-name>`, y para los valores está el elemento `<value>` o `<null-value>` para los valores nulos. Un ejemplo sería:

```
<managed-bean>
  <description>Breve descripción sobre este
```

```

ManagedBean</description>
<managed-bean-name>NombreBean</managed-bean-name>
<managed-bean-class>claseDelBean</managed-bean-class>
<managed-bean-scope>ambito</managed-bean-scope>
<managed-property>
    <property-name>nombre</property-name>
    <value>Juan</value>
</managed-property>
</managed-bean>

```

Si tuviéramos un ManagedBean con atributos de tipo array, lista o mapa se puede usar la etiqueta **<list-entries>** o **<map-entries>** para los mapas, anidada dentro de la etiqueta **<managed-property>**. Por defecto los valores son Strings pero puede indicarse el tipo usando la etiqueta **<value-class>** Algo así:

```

<managed-property>
    <property-name>nombres</property-name>
    <list-entries>
        <value>Juan</value>
        <value>Pepe</value>
        <value>Luis</value>
        <value>Jose</value>
    </list-entries>
</managed-property>

```

Los mapas son algo más complejos ya que tienen clave y valor. Los mapas se identifican con la etiqueta **<map-entries>**, que lleva una etiqueta **<map-entry>** por cada par clave valor que anida una clave **<key>** y un valor **<value>** (o **<null-value>**). Se puede especificar el tipo de la key con **<key-class>**, y el tipo del valor con **<value-class>** siendo siempre el tipo por defecto un String. Algo así:

```

<managed-property>
    <property-name>nombres</property-name>
    <map-entries>
        <key-class>java.lang.Integer</key-class>
        <map-entry>
            <key>1</key>
            <value>Juan</value>
        </map-entry>
        <map-entry>
            <key>2</key>
            <value>Pepe</value>
        </map-entry>
        <map-entry>
            <key>3</key>
            <value>Luis</value>
        </map-entry>
    </map-entries>
</managed-property>

```

Los ManagedBean también pueden anidarse.

Como se puede ver, a través del fichero descriptor del ManagedBean podemos describir y controlar completamente el ManagedBean.

Habrá ocasiones en que nos interesaría ir quitando de la sesión (contexto) las cosas que no vamos a ir necesitando, por ejemplo, en una aplicación que usa un ManagedBean para aceptar logins de clientes, una vez logeados correctamente ese ManagedBean que estará en el ámbito de sesión ya no lo necesita, con lo cual, en la siguiente clase que usa una vez logeados, en el constructor de la misma podemos eliminar ese ManagedBean, para ello accedemos al contexto de sesión y usaremos `removeAttribute("nombreAtributo")`; para eliminarla de la sesión y así no acumular datos en memoria que no necesitamos. Podemos ver un ejemplo de esto en la clase ClientesBean del proyecto creado durante el master y llamado j510-jsf-primer.

Una forma de controlar los atributos de sesión que no necesitamos o los que sí necesitamos es definir en el web.xml aquellos atributos de sesión de los que podemos prescindir o por el contrario aquellos que si vamos a necesitar durante toda la vida de la sesión y usar un escuchador que controle cuando se añade un atributo de sesión. Nosotros, en el proyecto j510-jsf-primer, tenemos una clase llamada `Eventos_AtributoSesion` dentro del paquete `com.atrium.escuchadores` que se encarga de hacer eso mismo.

En el web.xml hemos declarado los atributos de sesión que no necesitamos de la siguiente forma:

```
<!--
      se declaran aquellos managedbean que se quieran
      mantener en la sesión
      y por lo tanto preservarlos del proceso de limpiado.
Se pondrán los
      nombres definidos en el faces-config separados por
      comas
  -->
<context-param>
    <param-name>lista_managedbean_necesarios</param-name>
    <param-value>idioma,usuario</param-value>
</context-param>
```

Por supuesto también se declara el listener como se hace habitualmente:

```
<listener>
  <listener-
  class>com.atrium.escuchadores.Eventos_AtributosSesion</listener-
  class>
</listener>
```

Cuando queremos actualizar un formulario después de hacer una modificación hay que actualizar, no solo las propiedades del managedBean sino también los objetos

correspondientes de los componentes afectados.

Eso se hace primero casteando al tipo de componente, por ejemplo, en el caso de un inputText seria UIInput. A continuación buscando al componente usando los métodos de Facescontext

**getCurrentInstance().getViewRoot().findComponent("contenedorPadre:contenido
rPadre:nombrecomponente(id)").** A continuación hay que usar los métodos **resetValue()** y **setValue()** del componente afectado.

Por ejemplo:

```
UIInput codigo = (UIInput) FacesContext.getCurrentInstance()  
  
        .getViewRoot().findComponent("formu_administracion:panel_op  
ciones:codigo_rol");  
codigo.resetValue();  
codigo.setValue(getCodigo_rol());
```

En las paginas jsf todas las etiquetas del mismo tienen que estar contenidas en una etiqueta view. Solo puede haber una etiqueta view. Por ejemplo si hacemos un formulario ira dentro de esta etiqueta.

COPiar LOGIN.JSP DE LA aplicación J511-jsf-login

Si quiero que un componente responda a un evento tendré que asignarle el escuchador como en swing.

Todo componente al que no le ponga nombre, le sera autoasignado por jsf. Lo que sera el name en HTML. Aunque no vayamos a trabajar con JavaScript es muy recomendable que asignemos un id a cada componente. No tienen atributo action, ni method; ya que siempre se envia la petición por post.

Para hacer un form colocábamos una tabla. Ahora tenemos que recuperar el concepto de contenedor de posicionamiento (panelGrid). Al utilizar este contenedor, nos olvidamos de crear una tabla y él solo la creará. Para hacer una validación de obligación de llenar el campo, en vez de definir un validator, colocaremos el atributo required y como mensaje un attribute message. Para que salga el mensaje en pantalla, después de colocar el string del sms, hay que colocar una etiqueta message con el atributo del nombre del componente que genera el sms y lo muestra. Jsf tiene una gestión más eficiente de mensajes, con un objeto actionMessage, que tiene los atributos sumario y detalle, que poniendo a true showDetail y showSummary mostrarán ambos. También se puede formatear el mensaje en función del nivel de gravedad.

Navegacion

Las aplicaciones JSF usan las reglas de navegación para controlar la navegación entre páginas. En la arquitectura MVC, la navegación de la página es una responsabilidad del controlador. Estas reglas de navegación están contenidas en el archivo descriptor faces-config.xml.

Hay dos tipos de navegación, la estática y la dinámica.

La navegación estática es cuando una página siempre va a llevar a una misma página irremediablemente, por ejemplo cuando desde una página con un formulario que tiene un botón que al ser pulsado envía datos al servidor, el cual analiza los datos y al final muestra una página que será siempre la misma aunque su contenido varía en función de los datos enviados.

En ese caso al botón del formulario se le da un valor para su atributo **action**, por ejemplo:

```
<h:commandButton label="Aceptar" action="login" />
```

Esta acción desencadenante debe coincidir con la etiqueta **outcome** del fichero **faces-config.xml**, dentro de sus reglas de navegación. En esa regla de navegación se especificará que tras la acción **login**, se navegará a la página hola.jsp. No hay que olvidar la barra al inicio de las líneas de las etiquetas from-view.id y to-view-id.

La Navegación Dinámica, que es la más usada, el flujo de la página no depende de qué botón se pulsa, sino que depende de los datos que el cliente introduce en el formulario. Por ejemplo, una página de entrada a un sistema puede tener dos resultados posibles, éxito o fracaso.

El resultado depende de una computación realizada, por ejemplo si el nombre de usuario está o no registrado. El botón de Aceptar debe tener un método de referencia, por ejemplo:

```
<h:commandButton label="Aceptar"
action="#{loginControlador.verificarUsuario}" />
```

En este caso, **loginControlador**, hace referencia a un ManagedBean, y este debe tener un método **verificarUsuario**.

Un método de referencia, en un atributo de acción (action), no tiene parámetros de entrada y devuelve una cadena de caracteres (String), que será usada para activar una regla de Navegación, por ejemplo, el método **verificarUsuario**, antes mencionado, debería ser algo así:

```

public String verificarUsuario() {
    if(---){
        return "éxito";
    } else {
        return "fracaso";
    }
}

```

El método devuelve un String “éxito” o “fracaso”. El manejador de navegación usa el String devuelto para buscar una regla de navegación que coincida con eso que el método ha devuelto. De tal manera que las reglas de navegación declaradas en el archivo descriptor faces-config.xml deberían ser algo así:

```

<navigation-rule>
    <from-view-id>/jsp/login.jsp</from-view-id>
    <navigation-case>
        <from-outcome>correcto</from-outcome>
        <to-view-id>/jsp/administracion.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>incorrecto</from-outcome>
        <to-view-id>/jsp/login.jsp</to-view-id>
    </navigation-case>
</navigation-rule>

```

Si se añade una etiqueta **<redirect>**, después de **<to-view-id>**, el contenedor de JSP termina la petición actual y envía una redirección HTTP al cliente. Esto quiere decir que el usuario ve el URL de la página que él actualmente mira, en contra de la dirección de la página previa.

Pueden usarse **comodines (asteriscos *)** en las reglas de navegación, en la etiqueta **<from-view.id>**, por ejemplo: **<from-view-id>/application/*</from-view-id>**

Haciendo eso la regla se aplicara a todas las páginas contenidas en el directorio application, solo puede usarse un asterisco y debe estar al final de la cadena del from-view-id.

Junto a la etiqueta **from-outcome**, está la etiqueta **from-action**. Puede ser útil si se tienen dos acciones separadas con la misma cadena de acción, o **dos métodos de acción que devuelven la misma cadena**. Por ejemplo, supongamos que tenemos dos métodos, accionRespuesta y nuevoExamen, y ambos devuelven la misma cadena “repetir”, pues para diferenciar ambos casos de navegación, se usa el elemento from-action:

```

<navigation-case>
    <from-action>#{examen.accionRespuesta}</from-action>
    <from-outcome>repetir</from-outcome>
    <to-view-id>/repetir.jsp</to-view-id>
</navigation-case>

```

```
<navigation-case>
    <from-action>#{examen.nuevoExamen}</from-action>
    <from-outcome>repetir</from-outcome>
    <to-view-id>/index.jsp</to-view-id>
</navigation-case>
```

Eventos

Son tres los eventos que se pueden encontrar en JSF, los **actionListener** eventos de acción como los de J2SE, los eventos de cambio de contenido (**valueChangeListener**) y **phaseListener** que son eventos de fase que son lanzados por el servidor.

Los **eventos de acción** (actionListener) son disparados por componentes de comando, como **h:commandButton** y **h:commandLink**, cuando el botón o enlace es pulsado o activado.

Los **eventos de cambio de valor** (valueChangeListener) los disparan los componentes de entrada, como **h:inputText**, **h:selectOneRadio** y **h:selectMenuMany**, cuando el valor del componente cambia y el formulario es tramitado.

Los **eventos de fase** (phaseListener) son **disparados por el ciclo de vida de JSF**.

Los eventos se tratan cuando la petición llega al servidor no en el momento en que se producen en el cliente. Entre la fases 5 y 6. Se diferencian de los métodos de lógica en que los eventos siempre devuelven la página de origen y los de acción llevan navegación.

Los métodos de evento deben ser public, no devuelven nada (void), tienen cualquier nombre y reciben el evento, por ejemplo ActionEvent (el de JSF no el de AWT).

```
public void comprobar_NIF(ActionEvent evento) {  
}
```

Los métodos de los eventos se implementan también en las ManagedBean.

Para cargar un evento en un componente se usan las etiquetas de evento con el nombre del método que va a tener la acción que va a disparar el evento. Por ejemplo en un botón (commandButton) podemos poner un **atributo actionListener** que lleva una expresión UL donde le especificamos el método de la clase ManagedBean que contiene el código que se va a ejecutar cuando se pulse ese botón. Por ejemplo:

```
<h:commandButton actionListener="#{bean.metodoDelEvento} />
```

El parámetro que recibe el método del evento dependerá del tipo de escuchador que estemos aplicando al componente de la página que lo va a disparar, por ejemplo, al escuchador **actionListener** le corresponderá el parámetro **ActionEvent**, al escuchador **valueChangeListener** le corresponde el evento **ValueChangeEvent**.

Eventos de Cambio de Valor

Normalmente puede interesar que estos eventos sean tratados cuando se produce el cambio de valor y no después de que se haya pulsado el botón de envío del formulario en la fase del ciclo de vida correspondiente. Para ello, en la etiqueta del componente que dispara este evento podemos apoyarnos en una **etiqueta de JavaScript** **onChange="submit()"** para invocar una función que llame al bean en ese momento en que se produce el cambio del valor del componente. También podemos usar **ICEfaces** (lo veremos más adelante) que **trabaja con peticiones AJAX** y lo va tratando según se producen los eventos.

La clase `ValueChangeEvent` es la que trata este tipo de eventos y extiende de `FacesEvent`, ambas residen en el paquete `javax.faces.event`. Los métodos más usados por este evento son:

En `javax.faces.event.ValueChangeEvent`:

`UIComponent getComponent()`: Devuelve el componente de entrada que disparó el evento.

`Object getNewValue()`: Devuelve el nuevo valor del componente, después de validado y convertido.

`Object getOldValue()`: Devuelve el valor previo del componente.

En `javax.faces.event.FacesEvent`:

`void queue()`: Cola de eventos a soltar al acabar la fase actual del ciclo de vida

`PhaseId getPhaseId()`: Devuelve el identificador de fase correspondiente a la fase durante el evento que es soltado.

`Void setPhaseId(Phase id)`: Conjunto de identificadores de fase correspondientes a la fase durante los eventos que son soltados.

Eventos de Fase

Estos eventos se producen antes y después de cada fase del ciclo de vida. Los manejan escuchadores de fase que deben ser declarados en el archivo de configuración `faces-config.xml`, tal que así:

```
<faces-config>
  <lifecycle>
    <phase-listener>eventoFase</phase-listener>
  </lifecycle>
</faces-config>
```

Pueden especificarse tantos como sean necesarios. Los escuchadores se invocan en el orden en que han sido especificados en el fichero de configuración.

Los escuchadores de Fase se implementan por medio de la interfaz **PhaseListener** del paquete `javax.faces.event`. Esta interfaz define tres métodos:

```
PhaseId getPhaseId()
void afterPhase()
void beforePhase()
```

getPhaseId dice a la implementación JSF cuando debe entregar los eventos de fase al escuchador; por ejemplo, `getPhased()` podría devolver **PhaseId.APPLY_REQUEST_VALUES**. En ese caso, `beforePhase()` y `afterPhase()` serían llamados una vez por el ciclo de vida: antes y después de la fase aplicación de valores de petición. También podría especificar **PhaseId.ANY_PHASE**, lo cual realmente significa todas las fases y en ese caso los métodos de escuchadores `beforePhase()` y `afterPhase()` serían llamados seis veces por ciclo de vida, una vez por cada fase. Los oyentes de fase son útiles para la depuración y corrección de errores.

Paginas Web JSF

Las páginas JSF son páginas JSP con etiquetas personalizadas, es decir lleva sus TLDs, etc.

Las páginas JSF tienen dos paquetes de etiquetas personalizadas (dos TLDs), el que contiene los componentes UI básicos de JSF y el que contiene los componentes que controlan otras acciones importantes como validadores y manejadores de eventos. Por lo tanto podría haber dos etiquetas taglib:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h">
```

Contiene componentes UI como componentes de formulario.

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f">
```

Se utiliza siempre. Se usa para registrar eventos, validadores, conversores y otras acciones de componentes.

Los componentes UI de la página están representados en el servidor como objetos con estado, esto permite manipular el estado del componente y conectar los eventos generados por el cliente a código en el lado del servidor.

Los componentes UI pueden ser simples como por ejemplo, un botón, o pueden ser compuestos como por ejemplo una tabla, que puede estar compuesta por varios componentes.

La tecnología JSF proporciona una arquitectura de componentes rica y flexible, que incluye:

- Un conjunto de clases `UIComponent` para especificar el estado y

comportamiento.

- Un modelo de renderizado que define como renderizar los componentes.
- Un modelo de eventos y escuchadores que define como manejar los eventos de los componentes.
- Un modelo de conversión que define como conectar conversores de datos a un componente.
- Un modelo de validación que define como registrar validadores con un componente.

Las clases UIComponent son extensibles, con lo cual podemos extenderlas para crear nuestros propios componentes personalizados.

Todas las clases de componentes UI descienden de la clase **UIComponentBase**, que define el estado y el comportamiento por defecto de un UIComponent.

El conjunto de Clases UI incluida en la versión de JSF (1.2) que estamos estudiando en nuestro master es:

UICommand: Representa un control que dispara actions cuando se activa. Botones y links.

UIForm: Encapsula un grupo de controles que envía datos de la aplicación. Es análogo a la etiqueta form de HTML.

UIGraphic: Muestra una imagen.

UIInput: Toma datos de entrada del usuario. Es subclase de UIOutput.

UIOutput: Muestra la salida de datos en la página.

UIPanel: Muestra una tabla.

UISelectItem: Representa un solo ítem de un conjunto de ítems.

UISelectItems: Representa un conjunto completo de ítems.

UISelectBoolean: Permite a un usuario seleccionar un valor booleano en un control, seleccionándolo o deseleccionándolo. Esta es subclase de UIInput.

UISelectMany: Permite al usuario seleccionar varios ítems de un grupo de ítems. Esta es subclase de UIInput.

UISelectOne: Permite al usuario seleccionar un ítem de un grupo de ítems. Es subclase UIInput.

Normalmente no tendremos que usar estas clases directamente, sino que usaremos las etiquetas correspondientes al componente. La mayoría de estos componentes se pueden renderizar de formas diferentes, por ejemplo, un UICommand se puede renderizar como un botón o como un hiperenlace.

Las etiquetas de JSF representan componentes que a su vez son clases (objetos) que pueden ser manipulados programáticamente, es decir sus atributos (name, value, etc) pueden ser cargados, cambiados, etc, en las clases ManagedBeans.

La implementación de referencia de JavaServer Faces proporciona una librería de etiquetas personalizadas para renderizar componentes en HTML.

Etiquetas Estándar de JSF

Como ya dijimos hay dos paquetes de etiquetas, las **core** (prefijo **f**) y las **HTML** (prefijo **h**). En total forman 42 etiquetas.

Las etiquetas **core** manejan eventos, atributos, conversiones, validadores, recursos y definición de la página. La librería de etiquetas **html_basic** usadas básicamente para construcción de formularios y otros elementos de la interfaz de usuario.

Etiquetas Core

view	- Crea una vista
subview	- Crea una subvista
facet	- Añade una faceta a un componente, contenido adicional.
attribute	- Añade un atributo (clave/valor) a un componente
param	- Añade un parámetro a un componente
actionListener	- Añade una acción a un componente
valueChangeListener componente	- Añade un nuevo valor a un escuchador de un componente
convertDateTime	- Añade fecha a un componente
convertNumber	- Añade una conversión de número a un componente
validator	- Añade un validador a un componente
validateDoubleRange componentes	- Valida un rango de tipo Double para valores de componentes
validateLength	- Valida la longitud del valor de un componente
validateLongRange	- Valida un rango de tipo Long para valores de componentes
loadBundle	- Carga el origen de un elemento Bundle
selectItems componentes	- Especifica elementos (collection) para llenar
selectItem	- Especifica un elemento para seleccionar un elemento
verbatim	- Añade una marca a una página JSF

Etiquetas HTML

form	- Formulario HTML (método POST únicamente)
inputText	- Simple línea de texto de entrada
inputTextarea	- Multiples líneas de texto de entrada
inputSecret	- Contraseña de entrada
inputHidden	- Campo oculto
outputLabel	- Etiqueta para otro componente
outputLink	- Enlace HTML
outputFormat	- Formatea un texto de salida
outputText	- Simple linea de texto de salida
commandButton	- Botón: submit, reset o pushbutton. Hace petición (action).

commandLink	- Enlace (link) asociado a un boton pushbutton. Hace Peticion.
message	- Muestra el mensaje más reciente para un componente
messages	- Muestra todos los mensajes
graphicImage	- Muestra una imagen
selectOneListbox	- Seleccion simple para una lista desplegable. Usa selectItem.
selectOneMenu	- Seleccion simple para menú. Usa selectItem. Boolean.
selectOneRadio	- Conjunto de botones radio. Usa selectItem para cada radio.
selectBooleanCheckbox	- CheckBox (solo uno). Boolean.
selectManyCheckbox	- Conjunto de CheckBoxes.
selectManyListbox	- Seleccion multiple de lista desplegable
selectManyMenu	- Selección multiple de menu
panelGrid	- Tabla HTML. Va con columns.
panelGroup	- Agrupa dos o mas componentes para aparecer juntos.
dataTable	- Dos ó + componentes mostrados como uno dinámicamente.
column	- Columna de un dataTable. Puede llevar encabezado y pie (facet).

Atributos y Propiedades de las Etiquetas:

En el caso de las etiquetas de selección múltiple como listas, menús o radiobotonnes, cuando las opciones se leen de una base de datos siendo diferentes opciones dependiendo del cliente que sea, usaremos el atributo **selectedItems** que apuntará a una colección, la cual será una propiedad del ManagedBean que contendrá todos los valores que se van a mostrar en ese componente.

Para que este componente tenga datos cuando se carga la página por primera vez, hay que escribir la lógica que cargue los datos en el constructor del ManagedBean correspondiente a esa página.

El **dataTable** carga datos dinámicamente, se usa cuando se desconoce la cantidad de datos, o elementos que hay en una colección de objetos:

```
<h:DataTable value="#{obj.propiedadColeccion}"
var="nombreIteracion">
    <h:column id="nombreColumna">
        <h:outputText value="#{nombreIteracion.propiedad}" />
    </h:column>
    <h:column id="nombreColumna">
        <h:outputText value="#{nombreIteracion.propiedad}" />
    </h:column>
</h:DataTable>
```

facet: Lleva **name** (nombre del contenido añadido) y **value**. Para especificar un

encabezado de un column el name ha de ser **header** y si va a ser un pie de un column el name ha de ser **footer**.

```
<h:column id="nombreColumna">
    <f:facet name="header">
        <h:outputText id="nombrecolumna" value="Texto
        Cabecera" />
    </f:facet>
    <h:outputText value="#{nombreIteracion.propiedad}" />
</h:column>
```

Hay un ejemplo ya hecho en el proyecto j510-jsf-primero del master, en la pagina clientes.jsp, donde se puede ver como se hace un dataTable.

A tener en cuenta, muy importante:

Siempre que necesitemos incluir algo de lógica en una pagina JSF, como puede ser un bucle o un condicional if, etc., podemos recurrir a las etiquetas JSTL, incluyendo el taglib correspondiente en una directiva.

Todo componente de JSF debe estar dentro de la etiqueta view `<f:view>` y solo puede haber una etiqueta view por pagina. Lo normal es meter la etiqueta f:view dentro del body, para evitar problemas en este sentido.

Los **formularios** llevan la etiqueta `<h:form>` y **solo soporta peticiones POST** (esto cambia en la versión 2 que sí soporta peticiones GET).

Los formularios (<form>) no tienen action, se usan eventos como en J2SE, en los botones del formulario.

Los formularios tienen un atributo **ID** que normalmente es el único que se utiliza. Este atributo no es igual que en HTML, sino que **representa al atributo name** que se usaría **en HTML**. Con lo cual no puede haber dos IDs iguales en la misma página.

Todos los componentes deberían llevar su ID o este será asignado por el FaceServlet y no sabremos como se llama.

El ID que nosotros le pongamos a un componente luego se transforma añadiéndose por delante el ID que tenga el formulario (padre) separados por dos puntos (:). En caso de que el formulario estuviera dentro de otro componente el ID de ese componente iría antes del ID del formulario seguido de dos puntos (:).

JSF tiene sus propios contenedores para evitar usar tablas, de la misma forma que los había en J2SE. Por ejemplo `<h:panelGrid id="nombre" columns="2">` que generará una tabla con dos columnas en este caso.

Los formularios también usan la etiqueta `<h:outputtext ID="nombre" value="texto a mostrar">` para incluir texto.

`<h:inputtext ID="" value="#{obj.propiedad}">` son cajas de texto, usan los mismos parámetros que etiquetas usadas en HTML. El nombre de la caja de texto va en el parámetro value y puede usar expresiones, pero no expresiones EL sino expresiones UL. Que no comienzan por el signo dólar \$ sino por almohadilla #.

A diferencia de EL, en UL la expresión no se ejecuta hasta que el flujo del programa llega a la etiqueta.

UL sirve tanto para lectura como para escritura. Además no solo se pueden llamar propiedades sino métodos también, aunque sin pasarles argumentos. En JSF 2 ya sí se pueden pasar argumentos.

Los **botones** usan la etiqueta `<h:commandbutton ID="" value="texto mostrado" action="metodo a ejecutar en el ManagedBean con expresión UL">`

El atributo action del botón indica con una expresión UL el nombre del método del ManagedBean que tratará los datos enviados. Algo como esto:

`#{nombreClase.nombreMetodo}`

Para sacar los **mensajes de error** se usa la etiqueta `<h:message for="nombrecocomponente" />` como puede verse, aquí solo se indica el componente, pero el mensaje propiamente dicho se escribe en la clase ManagedBean correspondiente dentro del método que trate la lógica de esta petición. Para eso se usa un objeto de la clase **FaceMessage**, un ejemplo de código sería:

```
FacesContext.getCurrentInstance().addMessage("id del componente", new FacesMessage("nivel del error", "se ha producido un error", "texto del detalle"));
```

Los mensajes de error pueden llevar, nivel, sumario y detalle, separados por comas.

La clase **FaceContext** es la que nos permite acceder a los contextos de J2EE, solo tiene un método static que es **getCurrentInstance** el cual devuelve el contexto de JSF asociado al ManagedBean desde el que se llama. A su vez ese contexto tiene un método **addMessage** que nos permite añadir el mensaje de error al componente a través de un objeto **FacesMessage**.

Cuando el contenido de los atributos sea desconocido, porque se deben cargar dinámicamente en tiempo de ejecución, pueden accederse desde las clases ManagedBean, ya que todos los componentes son objetos que pueden ser accedidos programáticamente y tienen métodos para acceder a sus propiedades.

Conversiones y Validaciones en JSF

Cuando un usuario desde un formulario de una pagina web envía datos a un servidor al pulsar el botón de envío, estos datos se llaman “**valor demandado**”, los valores demandados se cargan en objetos (cada etiqueta del formulario tiene su correspondiente objeto componente). El valor que se carga en el objeto se llama “**valor tramitado**”.

Todos los valores demandados son cadenas de caracteres (Strings). Por otro lado la aplicación se ocupa de tipos arbitrarios, como enteros (int), fechas (date) o tipos mas sofisticados. Un proceso de conversión transforma esas cadenas en esos tipos. Los valores convertidos no son tratados inmediatamente en el bean, sino que son primeramente cargados en objetos de valores locales. Despues de la conversión, los valores locales son validados. En la pagina web pueden establecerse condiciones de validación, por ejemplo, que ciertos valores deban tener una longitud máxima o mínima. Después de que todos los valores locales hayan sido validados, comienza la fase de actualización de los valores del modelo, y los valores locales son cargados en el bean.

Para conservar la integridad del modelo, JSF efectua dos pasos. Los usuarios suelen equivocarse al introducir información en los formularios, si los valores se hubieran actualizado antes de haberse detectado el primer error del usuario el modelo podría entrar en un estado inconsciente y seria tedioso regresarlo al estado anterior. Por esa razón JSF primero convierte y valida todos los valores de entrada del usuario. Si se encuentran errores, la página es recargada, para que el usuario lo intente de nuevo, la fase de actualización del modelo comienza cuando todas las validaciones han ocurrido sin problemas.

Conversiones y Validaciones son parte del ciclo de vida de JSF. Tenemos la posibilidad de personalizar estas tareas y también podremos usar conversiones o validaciones ya establecidas.

Las conversiones y validaciones no son excluyentes, podemos usar una u otra o las dos. Primero se convierte y después se valida.

JSF es capaz de detectar las modificaciones en los valores entre peticiones, con lo cual si entre peticiones no han cambiado algunos valores que ya estaban validados o convertidos no realizara esa operación de nuevo.

CONVERSIONES

Las conversiones ya establecidas son las más clásicas, es decir, de texto a numero y de numero a texto. Tambien existen conversiones de fechas.

Cuando tenemos una etiqueta de tipo <h:inputtext> la conversión se realiza en las dos direcciones, desde la pagina al servidor y viceversa.

Los conversores y validadores pertenecen al paquete “F”. Para realizar una conversión, dentro del cuerpo del h:inputtext se puede usar una etiqueta de conversión como la etiqueta <f:convertNumber> lo cual realiza la conversión de texto a numero cuando va hacia el servidor y de numero a texto cuando vuelve a la pagina. Esa etiqueta convertNumber tiene atributos para indicar características de la conversión, por ejemplo:

minIntegerDigits: numero mínimo de dígitos antes de la coma

maxIntegerDigits: numero máximo de dígitos antes de la coma

minFractionDigits: numero mínimo de dígitos decimales

maxFractionDigits: numero máximo de dígitos decimales

integerOnly: Boolean: Se convierte a integer o no? Por defecto false.

groupingUsed: Booleano para indicar si se usa grupo de separadores. Por defecto false.

currencyCode: ISO 4217 – Código usado para conversión a valores monetarios.

currencySymbol: Símbolo de moneda usado.

type = number, percent o currency.: tipo de moneda. Ver indicación final.

pattern = “.....” : Personalización

Este tipo de conversiones se realizan usando un objeto **NumberFormat** que actuará dependiendo de la configuracion local para la moneda, decimales etc. La etiqueta es <f:convertNumber>

```
<h:inputText id="cantidad" value="#{pago.cantidad}">
  <f:convertNumber minFractionDigits="2" />
</h:inputText>
```

La conversión a fecha/hora se realiza usando un objeto **DateFormat**. La etiqueta es <f:convertDatetime> Lleva algunos atributos

type= time, date o both (por defecto date)

pattern= Configuración personalizada usando la configuración ya conocida de date

timeZone= Zona horaria (java.util)

dateStyle = default, sort, medium, long o full: Estilo si es fecha

timeStyle = default, sort, medium, long o full: Estilo si es hora

```
<h:inputText id="fecha" value="#{pago.fecha}">
  <f:convertDateTime pattern="MM/yyyy" />
</h:inputText>
```

La fecha y hora también se establecen usando la configuración local del servidor en el que se esté utilizando. Estos detalles han de ser tenidos en cuenta cuando se internacionaliza.

Las conversiones se pueden personalizar para ello habría que hacer un nuevo componente y declararlo en el faces-config.xml usando una etiqueta <converter>. Esta etiqueta lleva a la vez otras dos que son <converter-id> que es el nombre que le queramos dar al conversor y <converter-class> que es la ruta y nombre de la clase que se usa para la conversión. Por ejemplo

```
<converter>
    <converter-id>nombre conversor</ converter-id>
    <converter-class>ruta + nombre clase<converter-class>
</converter>
```

En la pagina jsp el código seria:

```
<h:inputtext .....>
    <f:converter converterId="id especificado en el faces-
config.xml" />
</ h:inputtext>
```

La clase que va a ser un **conversor** personalizado debe ser una clase publica que **implementa la interfaz converter**, y los **métodos** que se implementan son dos. Uno hace la conversión de texto a lo que nosotros queramos y el otro hace la conversión en el sentido contrario. Esos métodos son: **getAsObject()** que será publico devuelve Object y recibe tres parámetros FacesContext, UIComponent y String. El otro método es **getAsString** que es publico devuelve un String y recibe tres parámetros FacesContext, UIComponent y Object.

```
public class xxxx implements converter{

    Public Object getAsObject(FacesContext, UIComponent,
String){
        // Este es el que trata la conversion de entrada, hacia el
servidor
        If (...){
            Throw new ConverterException(new FacesMessage("texto
del error"));
        }
    }

    public String getAsString(FacesContext, UIComponent,
Object){
        // Este metodo trata la conversion de salida, hacia el
cliente
    }
}
```

Facescontext es el **contexto general de JSF**. A través de él accedemos a los atributos que nos interesen del contexto.

El objeto **UIComponent** que reciben ambos métodos representa a los componentes que forman el árbol de objetos que se crea con la petición, todas esas clases tienen una herencia común que es la clase **UIComponent** (User Interface Component), por lo tanto todos esos componentes se pueden castear a la clase del componente con el que vamos a trabajar en la conversión. Ese argumento es **el componente que vamos a convertir**.

Muchas veces se dará el caso de que la conversión no es posible debido, por ejemplo, a errores del usuario en el texto introducido en el formulario. Estos errores generan excepciones que cortan el proceso y el flujo vuelve a mostrar la página de origen al usuario. Este tratamiento de errores lo tenemos que hacer nosotros por código.

Para este tratamiento de errores se lanza la excepción usando la palabra reservada **Throw new ConverterException** que llevará el mensaje (**FaceMessage**) del error.

En las conversiones a tipos de moneda (currency), siempre se dará un error con el símbolo del euro €, es porque el símbolo del euro no está en el ISO-8859-1, es decir es un error de los navegadores, la solución es cambiar el ISO-8859-1 al ISO-8859-15.

Podemos crear tantos conversores personalizados como sean necesarios.

VALIDACIONES

Validar significa que vamos a comprobar que el valor, que ya está correctamente convertido al tipo que nos interesa, encaja en los parámetros de validez que vamos a utilizar. Se trata de validación sencilla, no implica el uso del modelo (base de datos).

Un ejemplo de validación sería el cálculo de la letra del NIF para ver si se ha escrito correctamente.

Los validadores son solo de entrada, es decir solo hay que validar la información que nos llega no hay que validar la salida al cliente.

Existen validaciones ya establecidas y también validaciones personalizadas

En el caso de validaciones predefinidas, en el cuerpo de la página se indicaría usando etiquetas F con estas opciones:

validateDoubleRange: Atributos maximum y minimum. Valida si está en un rango de valores double

validateLongRange: Atributos maximum y minimum. Valida si está en un rango de valores long

validateLength: Atributos maximum y minimum. Comprueba el número de caracteres

del valor.

Las tres etiquetas tienen atributos **minimum** y **maximum**, puede usarse uno solo o los dos. Si se usan los dos se comprueba un rango y si es uno solo será una cantidad o valor mínimo o máximo.

De esta forma:

```
<h:inputtext required="true">
    <f:validateLength minimum="9" /> // numero mínimo de
    caracteres 9
<h:inputtext>
```

Para hacer que un **campo** sea **obligatorio** se usa en el h:inputtext el atributo **required**, que **por defecto** es **false**.

Las **validaciones personalizadas**, al igual que las conversiones personalizadas, previamente se declaran en el faces-config.xml con la etiquetas:

```
<validator>
    <validator-id>nombre del validador</ validator-id>
    <validator-class>ruta + nombre clase</ validator-class>
<validator>
```

La clase de validación **implementa** la interfaz **validator**. Con un solo método a implementar llamado **validate**, que es un método público que devuelve void y recibe tres parámetros, **FaceContext**, **UIComponent** y **un Object**.

```
public class xxxx implements validator{

    public void validate(FaceContext, UIComponent, Object) {
        // código de la lógica de validacion
        If (.....) {
            Throw new ValidatorException(new FaceMessage("Texto del
error"));
        }
    }
}
```

Las excepciones se tratan lanzando la **excepción ValidatorException** que lanza un mensaje (**FaceMessage**) con el texto del error.

Existen ficheros ya preparados para cuatro idiomas en el paquete **javax.faces**, dentro de la librería **Java EE 5 Libraries – jsf.impl.jar** en realidad esos ficheros contienen las claves para tratar los mensajes de error pero la traducción no está bien hecha ni está completa. Se pueden copiar y ponerlos en un paquete creado por nosotros para la internacionalización.

Es faces-config.xml usamos una etiqueta **<application>** para modificar algunos ajustes del comportamiento del JSF, por ejemplo podemos decirle que use un fichero u otro para los recursos de los mensajes de error.

En concreto para los mensajes de error, dentro de las etiquetas **<application>** usaríamos la etiqueta **<message-bundle>** donde le indicamos trayectoria y nombre del fichero sin la extensión .properties. por ejemplo así:

```
<application>
    <message-
bundle>com.atrium.textos.textos_errores_es</message-bundle>
</application>
```

El throw que ponemos en el método adecuado de las clases de validación o de conversión lanza esos mensajes ya a partir de la clave.

También puede usarse una etiqueta **<h:validatorMessage>** o bien **<h:converterMessage>** con el texto personalizado como parametro, sin usar los ficheros properties del sistema.

Mensajes de Error:

Es importante que el usuario pueda ver los mensajes de error cuando se producen en la conversión, y/o validación. Para ello se añaden etiquetas <h:message> o <h:messages> cuando se usan conversores y/o validadores.

Lo lógico es mostrar mensajes de error junto a los componentes que causan dichos errores. Por lo tanto se debe asignar valor ID y referenciarlo con la etiqueta h:message, por ejemplo:

```
<h:inputText id="cantidad" value="#{pago.cantidad}">
    <h:message for="cantidad" />
```

Un mensaje de error posee dos partes, sumario y detalle. Por defecto la etiqueta h:message muestra el detalle y esconde el sumario, pero eso se puede controlar con los atributos booleanos **showSummary** y **showDetail**, así:

```
<h:message for="cantidad" showSummary="true" showDetail="false">
```

Tambien se pueden usar los atributos styleClass y style para cambiar el estilo de los mensajes, por ejemplo:

```
<h:message styleClass="mensajeError" />
ó
<h:message style="color:red" />
```

Por defecto los mensajes de error usan un archivo properties estándar, los cuales usan textos como “*Conversion error occurred*”. Para cambiar los mensajes de error estándar hay que modificar los archivos que usa cambiando el valor de la clave javax.faces.component.UIInput.CONVERSION, por el texto que deseemos.

Por ultimo hay que establecer el nombre de la base que recoge los mensajes en el fichero de configuración (faces-config.xml), en el caso de que a ese fichero lo hayamos llamado mensajes.properties seria algo como:

```
<faces-config>
    <application>
        <message-bundle>mensajes</message-bundle>
    </application>
</faces-config>
```

Internacionalizacion con JSF

Se procede a través de ficheros properties, como siempre.

En JSF se pueden leer ficheros .properties de dos formas, una de ellas seria usando la etiqueta **<f:loadBundle var="" basename="" />** que se usa para indicar la clase ManagedBean que trata la internacionalización, lleva dos atributos, uno es **basename** para indicar la clase + la propiedad que se va a usar para obtener el idioma, debe tenerse en cuenta que el nombre de la clase es la que se le ha dado en el el web.xml dentro del atributo del parámetro de contexto, en realidad lo que se obtiene en el base name (como su nombre indica) será la ultima parte del nombre del archivo properties. El nombre de ese archivo se crea a partir de una primera parte que es igual en todos los archivos de internacionalización mas una segunda parte que es la que distingue el idioma, será algo como “es” o “en”, etc. El otro atributo (**var**) se va a usar después para poderse referirse a este loadBundle. El loadBundle tiene ámbito de petición. Por ejemplo:

```
<f:loadBundle basename="mensajes" var="msjs">
.....
<h:outputText value="#{msjs.saludo}">
```

Ese codigo leeria el valor que contenga la clave saludo del archivo properties mapeado con el nombre mensajes dentro del fichero descriptor web.xml y la sacaría por pantalla.

Esta etiqueta loadBundle debería ser la primera dentro de la etiqueta **<f:view>**. Lo que hace es declarar esa clase para ser usada después. Esta es la forma correcta para internacionalización. Tambien pueden usarse archivos .properties solo para escribir el texto estático que queremos incluir en nuestras páginas sin necesidad de internacionalizar.

En el faces-config.xml se declara este nuevo ManagedBean, usando las etiquetas **<managed-bean>** que ya conocemos con sus etiquetas hijo.

La clase ManagedBean que trata este asunto de la internacionalización es una clase normal pública con propiedades privadas, métodos accesores, en el constructor podemos establecer el idioma elegido por defecto. Como eso esta en un parámetro de contexto que es el que lee la configuración de web.xml, se usa la buena practica de crear una clase aparte para leer del contexto de JSF, dentro de un paquete de utilidades. Esa clase tendría un método static que devuelve HttpSession, no recibe nada y tiene una línea de código como este para obtener la sesión:

```
public static HttpSession xxxx() {
Return(HttpSession)FacesContext.getCurrentInstance().getExternal
Context.getSession(false);
}
```

O como este para obtener la petición:

```

public static HttpServletRequest xxxx() {
    Return(HttpServletRequest) FacesContext.getCurrentInstance().getExternalContext.getRequest();
}

```

Como decíamos, el constructor de la clase para manejar los idiomas usaria esos métodos estáticos para obtener el idioma y así indicar la ruta y nombre del archivo a leer dependiendo del idioma del usuario.

Como en otros casos el archivo de internacionalización depende de la terminación de su nombre que coincide con el idioma, el cual puede recogerse como hemos hecho en otras ocasiones dentro de J2EE, usando un filtro, dando una opción por defecto, dejando que el usuario elija su idioma, etc. En el archivo web.xml podemos usar un <context-param> para establecer el idioma por defecto.

```

<context-param>
    <param-name>idioma_por_defecto</param-name>
    <param-value>en</param-value>
</context-param>

```

En la etiqueta sobre la que queremos internacionalizar, en el atributo value usamos una expresión EL donde escribimos la clave del properties de la que leer el valor, así: `#{{idioma['clave.properties.para.leer']}}` donde idioma sería la variable (var) usada en el <f:loadbundle>.

Integracion de jsf con Spring

Para decirle a jsf que Spring se encargara de crear los manageBean, tenemos que indicárselo en el faces-config.xml.

```

<application>
    <variable-resolver> org.springframework.web.jsf.</variable-resolver>
</application>

```

hay que quitar la definición q sobre en este fichero

ahora hay que declarar en el applicationContext, en este caso lo haremos en un nuevo fichero de configuración para no cargar mucho el que ya tenemos creamos un fichero descriptor de spring managebean.xml y en él describimos el bean
COPIAR FICHERO MANAGEDBEAN.XML

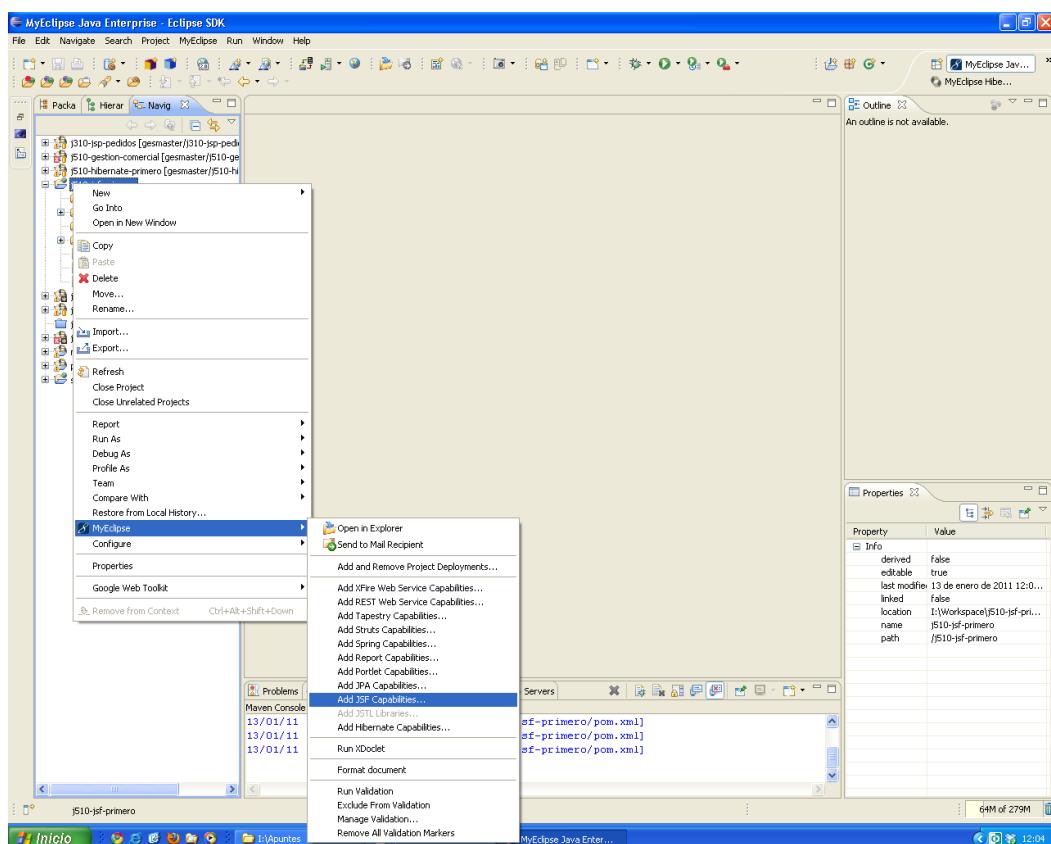
Hacerlo así tiene un pequeño inconveniente, el asistente de la página no reconoce ese nuevo managedbeans.

Tambien hay que decirle a web.xml que hay dos ficheros descriptores, para que lo reconozca.

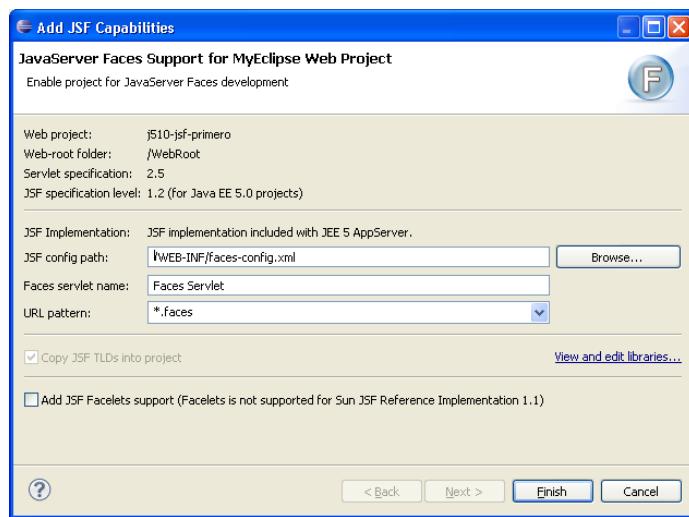
Haciendolo de esta forma en la accion no necesitaremos instanciar las fachadas por que spring las injecta. No podemos olvidar que tendremos que dotar a la clase de metodos accesores.

Implementación de proyecto JSF en MyEclipse

A partir de un nuevo proyecto web. Al pulsar con botón derecho sobre el nombre del proyecto vamos a MyEclipse – Add JSF Capabilities



Esto abre una ventana donde escribimos el nombre y ruta para el faces-config y el FacesServlet así como el URL pattern. Podemos aceptarlo tal como viene. Y pulsamos Finish.



ICEfaces

Es una extensión de JSF y solo se puede instalar en los proyectos JSF. Forma un conjunto nuevo de componentes y también incorpora una serie de temas como AJAX para hacer las peticiones al servidor de forma automática con esta tecnología. Nosotros vamos a ver la **versión 1.8.x** y NO la nueva versión que acaba de salir, la 2.0, (11 de Enero 2011) y que se refiere a JSF 2.

ICEfaces controla de forma automática toda la generación de **JavaScript a través de AJAX**.

Además de las peticiones, también controla de forma automática las actualizaciones de la página que llegan via AJAX. Para ello incorpora tres servlets, uno se encarga de las peticiones normales, otro se encarga de las peticiones AJAX y otro se encarga de la seguridad.

Otra característica que toca ICEfaces es el **diseño de la Web**, incorpora un conjunto de **estilos CSS** para dar una apariencia vistosa a las páginas que genera. Incluyendo imágenes preestablecidas e iconos. Si quisieramos crear nuestro propio estilo o modificar alguno, podemos abrir la hoja de estilo CSS de ICEfaces y cambiarla para adaptarla a nuestro gusto, no es recomendable hacer una hoja de estilo nueva, sino usar las de ICEfaces aunque las vayamos a cambiar completamente.

Para el **JavaScript** se apoya en **script.aculos** (Prototype), y lo genera automáticamente a partir de clases Java.

ICEfaces también controla **AJAX-PUSH**, que son eventos de AJAX en el servidor. AJAX-PUSH se usa por ejemplo en el caso de una tienda online, cuando un cliente hace una compra de un artículo, se puede avisar a los otros clientes en tiempo real para que sepan que se ha vendido un nuevo producto como el que están viendo y se pueden acabar, otro ejemplo puede ser una web de subastas, cuando alguien hace una puja se informa a los demás que están siguiendo la subasta de un artículo determinado. Es decir AJAX-PUSH envía respuestas automáticas a otras sesiones dependiendo de lo que pasa en una de las sesiones. No todos los servidores lo soportan y otros necesitan añadir algún plugin para poderlo utilizar.

ICEfaces también funciona con facelets igual que JSF.

ICEfaces tiene una versión gratuita y otra de pago. ICEfaces está muy bien documentado, en su página web, tiene tutoriales muy completos, con ejemplos que funcionan a la primera. Es una empresa canadiense. <http://www.icefaces.org/>

ICEfaces modifica el paquete de etiquetas H de JSF, no el F. Es decir sustituye los componentes de JSF respetando su nombre y añade muchos más. Particularmente

ICEfaces añade atributos a los componentes de JSF además de crear componentes que no existen en JSF. ICEfaces sigue usando JSF por lo tanto el ciclo de vida es el mismo.

Ejemplos de atributos de IceFaces:

visible – hace visible o funcional (lanza eventos de AJAX o no) el componente
disabled – hace invisible o no funcional (no lanza eventos de AJAX) el componente
partialsubmit – Es un booleano y hace que cuando se cambia el contenido de la pagina parcialmente, como que cambie la selección de un radio, etc. Envia solo la información que ha cambiado y se recibe respuesta a ese cambio para que se actualice solo esa parte. Algunos componentes lo tienen por defecto en true y otros en false.

Existe un componente en **ICEfaces** que **genera menús** en páginas web de forma rápida y sencilla. Un menú comienza por un contenedor específico para menús que es **menubar**, así:

```
<ice:menubar orientation="horizontal/vertical">
    <ice:menuitem id="x" value="texto" icon="icono" ActionListener="#{obj.metodo}" (tb
Action)>
        <ice:menuitem ... />
        <ice:menuitem ... />
        <ice:menuitem ... />
    </ice:menuitem>
</ice:menubar>
```

Tiene que estar dentro de un formulario ya que los menus realizan peticiones. Tiene atributos para la orientación horizontal o vertical, el atributo es **orientation**, la posición se realiza por estilo, es decir debemos ponerlo en un div.

Cada ítem del menú se trata con **menuitem** anidados según tenga submenús o no. Muneuitem lleva los atributos **value** para el texto a mostrar y **icon** para el icono de forma opcional. Además lleva un **evento** como **ActionListener** que mediante una etiqueta UL le indicamos el método que va a ejecutar. Tambien puede tener atributo Action lo que implicara navegación.

Para saber que elemento del menú ha sido pulsado, en el método del evento que recibe el parámetro ActionEvent (`evento.getComponent().getId()`) podemos usar ese ActionEvent para saber cual es la ID del elemento que ha disparado este evento y así poder escribir la lógica para esa opción de menú concretamente.

Para añadir dinamismo al menú y así cambiar los elementos programáticamente, podemos usar la etiqueta menuitems `<ice:menuitems>` que recibe la colección de elementos, en su atributo value del menú; así podemos, mediante código crear los elementos que conformaran el menú.

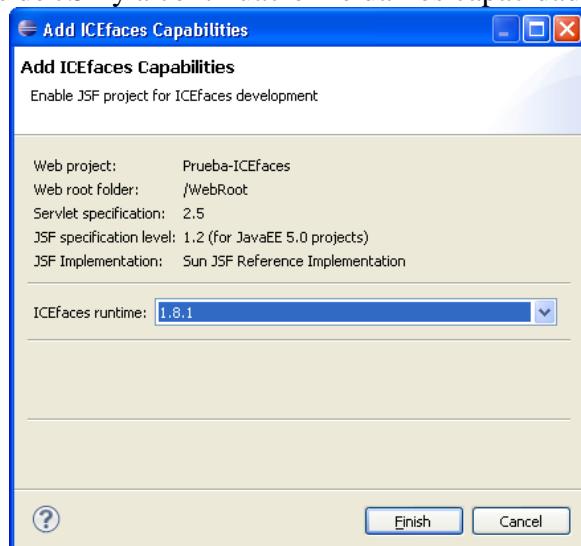
El código para **crear menús con contenido dinámico** se hacen dentro de un **ManagedBean** que tiene que implementar la interfaz **ActionListener** (la de JSF no

la de AWT). En la lógica del método que implementa el código para crear las opciones del menú basándose por ejemplo, en los roles o tareas que un usuario tiene asignadas en la base de datos, al añadir el actionListener como atributo del elemento de menú programáticamente usando el método addActionlistener, lo que hacemos es crear un escuchador, el método que tenemos que implementar por usar la interfaz ActionListener se llama **processAction** y es el que ejecuta el código asociado al evento de ese elemento del menú.

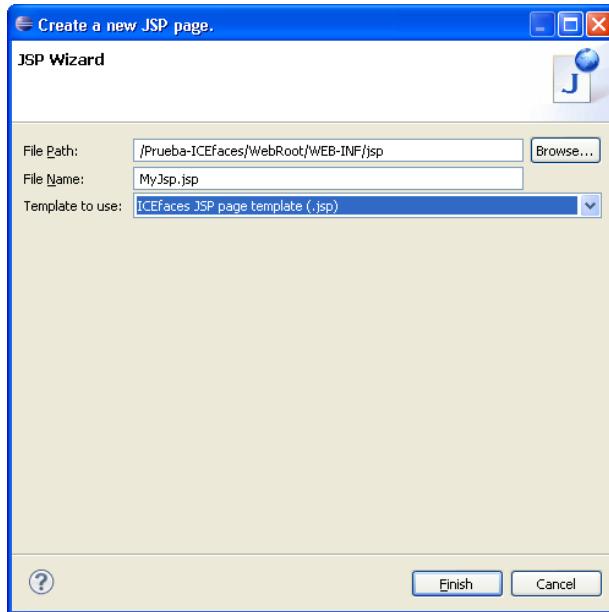
En el **proyecto j510-icefaces-menu** tenemos la clase (ManagedBean) **Menu_Dinamico_Java** que sirve de ejemplo para la creación de un menú dinámico que incluye la inclusión de elementos de submenú y un método que trata los eventos lanzados por ese menú dinámico.

Creacion Proyecto ICEfaces en MyEclipse

En **MyEclipse**, para hacer un proyecto ICEfaces crearemos un proyecto web, entonces le damos capacidades de JSF y a continuación le damos capacidades de ICEfaces.



Para hacer páginas se hace como con JSF pero en el wizard elegiremos la opción ICEfaces JSP page template (.jsp).



Para **añadir hibernate** en MyEclipse se hace como en cualquier otro proyecto web. Creamos un paquete para hibernate, le damos capacidades de hibernate al proyecto y hacemos la ingeniería inversa seleccionando las tablas que nos interesan.

Esto hará cambios en web.xml mapeando los nuevos servlets de ICEfaces mas el de JSF. Además incluirá los servlet-mapping que son necesarios y no hay que tocarlo ya que la mayoría de peticiones AJAX irán por ahí.

En el web.xml también encontraremos algunos parámetros de contexto ya creados <context-param>, hay uno que es el **DEFAULT_SUFFIX**, esto significa que las páginas JSP van a ser documentos XML, con lo que el tratamiento y la sintaxis van a ser diferentes. Como **nosotros no vamos a usar páginas JSPX borraremos ese parámetro de contexto o no funcionara nuestro proyecto.**

En el archivo web.xml también encontraremos un escuchador muy interesante y se usa para evitar las repetidas peticiones por parte del usuario (síndrome del dedo nervioso...). De forma que se evita el envío masivo de peticiones AJAX al servidor.

En la carpeta webroot encontraremos una carpeta xmlhttp que contiene varias carpetas, una de ellas (css) incluye tres estilos para la decoración de nuestra web.

DataTable en ICEfaces

El componente **DataTable** de ICEfaces es muy interesante por sus añadidos con respecto al datatable de JSF. Por lo tanto, además de lo que nos ofrece el de JSF, el DataTable de ICEfaces nos proporciona, por ejemplo, una barra de navegación para paginar los resultados, el cual es un componente que se llama **DataPaginator** que debe estar asociado a un DataTable y va realizando las peticiones AJAX al servidor para ir mostrando las filas que nos interese que muestre. Además el DataPaginator tiene una labor informativa hacia el usuario.

```
<ice:datapaginator for="Id del DataTable" paginator ="true">
    <f:facet name="first">
        <img.../> ó bien
        <ice:iceGraphicImage url="...">
    </f:facet>
</ice:datapaginator>
```

El numero de líneas a mostrar se define en el datatable con el atributo rows. El atributo **for** se refiere al ID del DataTable con el que esta relacionado, el atributo **paginator** es un booleano para decirle si queremos que muestre las características de navegación (true) o bien las de información (false).

En el caso de la opción de navegación, para indicar el numero de botones que se van a usar en la paginación se usan **facets** con unos nombres concretos (name), **first** para el primer registro, **last** para el ultimo, **next** para el siguiente y **forward** para el número de registros de salto, dentro del facet ira el **icono** que queremos que se muestre opcionalmente, que se puede mostrar con la etiqueta **img** de HTML o bien con **<ice:graphicImage url="...">**.

Para mostrar la información se pone a false el atributo paginator y se usan unos atributos adicionales:

rowcountvar – numero total de filas
displayrowcountvar – numero de filas que esta mostrando
firstrowcountvar – la primera fila que esta mostrando
lastrowcountvar – la ultima fila que esta mostrando
pagecountvar – numero total de paginas
pageindexvar – la pagina que esta mostrando

El mensaje se compone usando un texto con formato, por formato entendemos un texto con variables que serán sustituidas por los valores de los atributos anteriores. Por ejemplo:

```
<ice:datapaginator for="IdDataTable" paginator="false"
    rowCountvar="nom1" displayRowCountvar="nom2">
    <ice:outputformat value="Texto literal{0} de {1} a {4}...">
        <f:param value="#{nom1}">
```

```

<f:param value="#{nom2}">
</ice:datapaginator>

```

El contenido de las variables va en las etiquetas **<f:param>**, donde con un atributo **value** indicamos el nombre de la variable que aplica y escrita en la etiqueta **datapaginator** y el orden en el que se colocan indican el numero en el texto con formato, comenzando por cero.

Una tabla tiene una serie de funcionalidades interesantes como poder ordenar los resultados por columna, o bien poder seleccionar un ítem de la tabla.

Para poder seleccionar filas se usa un componente llamado **rowselector** **<ice:rowselector>** se pone dentro de un column del datatable y solo se coloca uno. Por ejemplo:

```

<ice:datatable>
  <ice:column>
    <ice:rowselector value="#{obj.prop}" multiple="true/false"
selectionListener="#{obj.prop}" inmediate="true/false" />

```

Lo que necesitamos es que el objeto que contiene la colección de filas que se estén mostrando tenga un campo booleano para poder indicar qué filas están seleccionadas. Puede permitirse selección multiple o sencilla. Los atributos del rowselector son:

value – La propiedad booleana donde se colocara el valor para indicar si esta seleccionada, usando una expresion.

multiple – booleano para indicar si se permite o no la selección multiple.

selectionListener – Escuchador de eventos, como valor recibe en una expresión el objeto.metodo que contiene la lógica para este evento. Como ICEfaces funciona por AJAX si queremos que se maneje de forma inmediata y no según el ciclo de vida de JSP deberemos poner un atributo **inmediate** a true.

inmediate – booleano para indicar si queremos o no que el evento **selectionListener** se atienda de forma inmediata, es decir a través de AJAX en el momento en que se produce la acción que dispara el evento que en este caso es la selección o deselección de una fila.

Para ordenar las filas por columnas tenemos que hacer el código nosotros, el componente lo único que hace es poner el icono de la forma como se ha ordenado en la columna adecuada y pintar las filas en el orden que nosotros le hayamos programado.

La ordenación se indica en las columnas con el atributo **CommandSortHeader** y se pueden poner en tantas columnas como queremos permitir la ordenación, es decir va dentro de la etiqueta column del datatable.

Los atributos de CommandSortHeader son:

columnName – Para indicar la propiedad del managebean por la que queremos ordenar la columna, será una expresión UL indicando objeto y propiedad.

arrow – booleano para indicar si queremos un icono de una flecha que indique la

ordenación en la cabecera de la columna.

Además, en la etiqueta del dataTable pondremos los atributos **sortColumn** y **sortAscending**, para establecer cual es la columna de ordenación y cual es el sentido de la ordenación respectivamente.

```
<ice:datatable id="xx" rows="x" sortColumn="#{obj.prop}"  
sortAscending="#{obj.prop}">  
  <ice:column>  
    <ice:CommandSortHeader columnName="#{obj.prop}"  
    arrow="true/false">
```

En el managedBean hemos de escribir el código de ordenación. En el proyecto llamado demo_ICEFACES_DATATABLE1 que tenemos en nuestro workspace del master Java, tenemos una clase que sirve de ejemplo para ver un método de ordenación y podemos ver como funciona este dataTable de ICEfaces. El managedBean que contiene el código se llama **sorttableTable** y el método se llama **sort()**, en este ejemplo el ManagedBean se apoya en la clase **SortableList** para hacer el orden ascendente o descendente de los datos. El proyecto completo vale para ser usado en cualquier caso que podamos necesitar y sus clases y páginas son de interés para ver como se programa una ordenación de este tipo, en concreto el método **sort()** es el que deberemos estudiar para adecuarlo a nuestro caso concreto ya que hace conversiones de tipos y usa nombres de tablas que no tienen porque coincidir con nuestras necesidades.

Contenedores de ICEfaces

ICEfaces dispone de una amplia variedad de contenedores, uno de ellos es un contenedor que muestra pestañas y se llama **panelTabSet**, cada pestaña será un **panelTab**. Los panelTab van dentro del paneltabSet y admite iconos, etiquetas etc.

Dentro de cada **panelTab** deberíamos poner un contenedor para poder colocar el contenido de esa pestaña, por ejemplo podemos usar **panelGrid** el cual generará un table.

```
<ice:form>
    <ice:panelTabSet id="panelOpciones" label="usuarios">
        <ice:panelTab id="panelUsuarios" label="USUARIOS">
        // vacio
        </ice:panelTab>

        <ice:panelTab id="panelRoles" label="ROLES">
            <ice:panelGrid id="contRoles" columns="3">
                <ice:outputText id="eti_cod" value="Codigo Rol:" />
                <ice:inputText id="codigo_rol" value="#{bean.codigoRol}" />
            </ice:panelGrid>
            <ice:message id="" for="codigo_rol" />
            <ice:outputText id="eti_descripcion" value="Descripcion Rol:" />
            <ice:inputText id="descripcion_rol" value="#{bean.descripcion}" />
            <ice:message id="" for=" descripcion_rol" />

            <ice:outputText id="eti_fechaAlta" value="Fecha de Alta:" />
            <ice:selectInputDate renderMonthAsDropDown="true"
                renderYearAsDropDown="true"
                renderAsPopUp="true" id="fecha_Alta" value="#{bean.fecha}" />
            <ice:message id="" for=" fecha_Alta" />
        </ice:panelGrid>
        <ice:panelGrid id="botonesRoles" columnas="4">
            <ice:commandButton id="botonAlta" value="ALTA"
                actionListener="#{bean.rol}"/>
            <ice:commandButton id="botonBaja" value="BAJA"
                actionListener="#{bean.rol}"/>
            <ice:commandButton id="botonConsulta" value="CONSULTA"
                actionListener="#{bean.rol}"/>
            <ice:commandButton id="botonModificar" value="MODIFICAR"
                actionListener="#{bean.rol}"/>
        </ice:panelGrid>
    </ice:panelTab>
</ice:panelTabSet>
</ice:form>
```

Usamos **selectInputDate** para la fecha ya que es un componente que nos incorpora un calendario. Usa **renderMonthAsDropDown** para desplegar los meses y es un booleano, tambien **renderYearAsDropDown**. Tambien **renderAsPopUp** que muestra o no una caja de texto para introducir la fecha directamente.

En el ManagedBean, habrá un método que trate este evento y diferenciará el botón que ha sido pulsado por la id del botón o también, en el caso de que en vez de botones usemos **commandLink** podemos poner un parámetro **<f:param id="" name="" value="">** si en vez de botones usamos **commandLink** y en la lógica del método en el ManagedBean diferenciamos por este parámetro. Logicamente todo los param tendrán el mismo name y un valor diferente que es lo que se obtiene en el método que trata este evento.

Podemos ver un ejemplo del managedBean que trata esto en el proyecto del master llamado **j510-icefaces-menu**, el ManagedBean se llama **Roles_beans.java** y el método se llama **gestión_Rol(ActionEvent evento)**.

CONTENEDOR PANELPOPUP

A veces puede interesarnos **mostrar mensajes de salida al usuario**, por ejemplo cuando damos un alta de usuario y entonces indicamos si el proceso de alta ha ido bien o no. El **panelPopup es modal** y puede aparecer o desaparecer cuando nos interese, por lo tanto es ideal para ese propósito de mostrar menajes en tiempo real al cliente.

Se juega con la propiedad **visible** (booleano) y la propiedad **modal** (booleano), tambien tenemos la propiedad **autocenter** para centrar en la pantalla. Tambien hay que darle valor al z-index Ademas se le pueden añadir algunos efectos.

Este panel tiene dos partes que se definen con **facets**, **header** y **body**. En estos facets podemos poner lo que queramos pero se puede usar **header para el título** y en el body podemos poner el contenido y un botón para cerrar el panel.

```
<ice:panelPopup visible="" modal="" autoCenter="">
    <ice:facet name="header">
        <ice:outputText value="titulo de la ventana modal" />
    </ice:facet>
    <ice:facet name="body">
        <ice:panelGrid columns="1" id="panel_mensajes" >
            <ice:outputText value="#{bean.propiedad}" />
            <ice:commandButton value="Cerrar mensaje"
actionListener="#{bean.metodo}"/>
        </ice:panelGrid>
    </ice:facet>
</ice:panelPopup>
```

Por supuesto ira dentro de un formulario.

Facelets

En la versión 1.2 de JSF es un añadido adicional, que podemos usar con JSF y con ICEfaces. Es parte de ello en la versión 2.0 de JSF y además es la forma de uso recomendada.

Facelets se encarga de dos cosas, por un lado se encarga del **maquetado de la página** y además ayuda en la **creación de componentes**.

Además añade más tareas y etiquetas. Cuando se usa facelets dejamos de usar páginas JSP y comenzamos a usar documentos XML, que pueden usar las extensiones JSPX o XHTML, por lo tanto nuestro mundo se transforma notablemente.

Esto requiere una transformacion de XML a JSP que hace Facelets automáticamente, es decir Facelets, traduce y convierte el XML a JSP, además corrige algunos errores que podamos haber cometido como poner mas de un view en el código, etc. Luego lo pasa todo al servlet para su conversión a HTML. Facelets, para esta conversión usa XSLT y XPATH. En esta fase de la conversión es donde se hace posible la maquetación. Esto posibilita la transformacion a otros tipos de navegadores, por ejemplo para ser vista en un móvil, sin tener que multiplicar el esfuerzo de construir varias vistas. Esto se hace programáticamente detectando el tipo de navegador que hace la petición.

Facelets es un proyecto de la comunidad libre. Para trabajar tanto en JSF como en ICEfaces hay que añadirlo manualmente adecuándose a la versión de JSF o ICEfaces que estemos usando. Hay que tener cuidado con esto porque equivocarse de versión no arroja un error, pero sin embargo el proyecto no funcionará bien.

En MyEclipse primero daremos capacidades JSF, luego capacidades ICEfaces y por ultimo capacidades Facelet eligiendo la versión adecuada.

La transformación que hace Facelets se hace añadiendo un componente al fichero faces-config.xml, que es el que se encargara de traducir de XML a JSP, para ello en la etiqueta **<application>** podemos declarar componentes para el funcionamiento general de JSF. El componente con la etiqueta **<view-handler>** por ejemplo para usar Facelets con ICEfaces el código en el archivo faces-config.xml es este:

```
<application>
    <view-handler>
        com.icesoft.faces.facelets.D2DFaceletsViewHandler
    </view-handler>
</application>
```

MyEclipse se encarga de poner este código automáticamente.

Además, en el fichero web.xml hay que dejar, o si no está hay que ponerlo, la declaración que quitábamos con ICEfaces que era la del **DEFAULT_SUFFIX** en el

<context-param>.

Para la **maquetación** Facelets usaremos un documento que será la plantilla a modificar y otro que contiene las especificaciones de esa plantilla, como hacíamos con Tiles para Struts.

La plantilla, ya hemos dicho que es un documento XML por lo tanto lleva su encabezado, su DTD, etc. **MyEclipse ya tiene una plantilla llamada icefaces.xml** que tiene todo eso y nosotros solo tendremos que modificarlo para que se ajuste a nuestras necesidades.

Hay que seguir la sintaxis tan estricta de este tipo de documentos XML, teniendo en cuenta que hay que cerrar todas las etiquetas y seguir la jerarquía, se pueden usar etiquetas de JSF, de ICEfaces, de HTML, más propias etc.

En la plantilla, lo que se definen son zonas que pueden tratarse con <div>, las etiquetas de icefaces que se usan llevan el prefijo UI, como es un archivo JSPX no lleva directivas para definir los taglibs lo que usa son espacios de nombres. El space name se especifica en el encabezado del documento con **xmlns:**. Para insertar un documento se usa la etiqueta <ui:insert name="nombreDocumento.jspx">

El contenido que rellene estas plantillas será un documento XML donde vendrán las etiquetas necesarias para que en el proceso de montaje se pueda añadir el contenido a esas zonas. La plantilla debe llevar una etiqueta <f:view>. Para definir el contenido se usa una etiqueta <ui:composition> con el atributo template para indicar la plantilla a la que vamos a dar contenido, especificando la ruta y el nombre de la plantilla. Además, por cada insert que lleve la plantilla especificaremos dentro del cuerpo de los ui:composition una etiqueta <ui:define> con un atributo name que se corresponde con el ui:insert al que se refiere. El contenido puede ser estatico o dinamico.

```
<f:view>
  <ui:composition template="ruta + nombre_plantilla">
    <ui:define name="nombre_insert">
      <ui:include src="#{obj.propiedad}">
    </ui:define>
  </f:view>
```

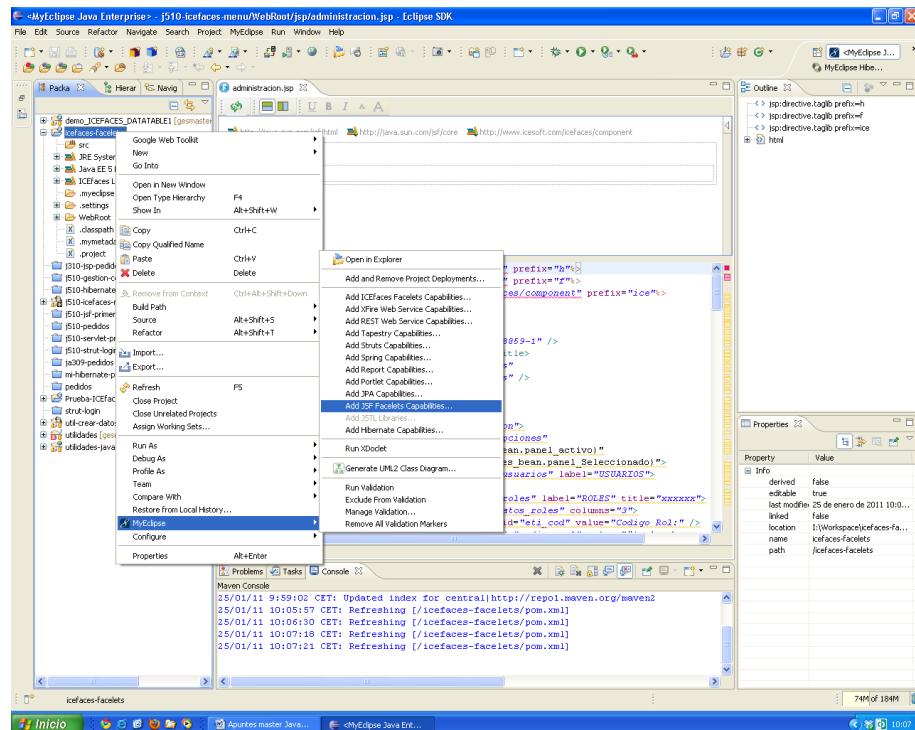
Cuando añadimos contenido dinámico podemos usar el atributo **src** en el include para obtener el valor de una propiedad que puede ser lo que haga que cargemos una pagina u otra. Por ejemplo basándose en una opcion de un menú. Como podemos ver en el proyecto j510-icefaces-facelest.

Tenemos un proyecto llamado j510-icefaces-facelest, ese proyecto esta hecho con anotaciones. Podemos usarlo para ver con detalle este tema de facelets.

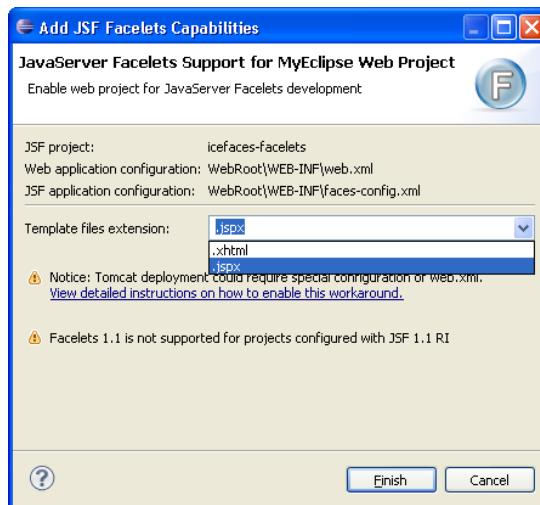
Creación de un proyecto facelets en MyEclipse

Para crear un proyecto que use Facelets con ICEfaces y JSF en MyEclipse seguimos estos pasos:

Primero creamos un proyecto web nuevo como siempre y le añadimos capacidades de JSF y de ICEfaces como hemos hecho hasta ahora, después le añadimos capacidades Facelets:



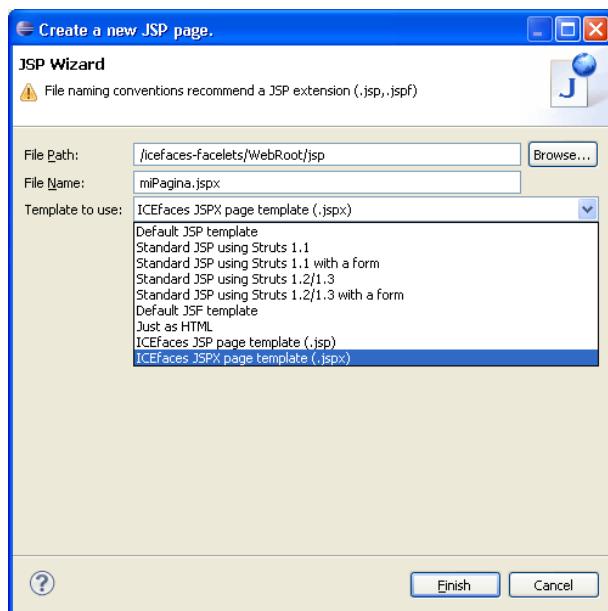
Nos preguntara si queremos que la extensión sea .jspx o .xhtml, elegimos la que queramos:



Pulsamos finish y se nos importan las librerías necesarias y se nos modifican el web.xml y faces-config.xml adecuadamente.

Despues, si lo necesitamos, podemos darle capacidades de Hibernate al proyecto también como hemos venido haciendo habitualmente.

Cuando vayamos a **crear una pagina nueva** hemos de elegir la opción de New – JSP (advanced templates) y en la siguiente ventana escogemos la opción ICEfaces JSPX page template (.jspx). Hay que acordarse de cambiar el nombre y la extensión (.jspx o .xhtml) en File Name:



Habra dos ejemplos de ejercicio. Uno con un menú dinámico y otro estático. j511-icefaces-menu

En cuanto a IceFaces , ver ejercicio j511-iceface-facelets

Para idiomatizar con Facelets. El idioma debe estar definido en un managedBean, al igual que antes estaba en un filtro o en un Servicios

Hasta ahora hemos trabajado con el concepto de aplicación, pero hay algo mas. Es habitual que las empresas necesiten algo mas, J2EE utiliza lo que se llaman servicios, que es una capa diferente.

La definición de servicio es; un programa, sin estado, que está a la espera de una petición para satisfacerla. No necesita tener interfaz gráfica. Normalmente los servicios no interactúan con clientes tipo usuario sino con otras aplicaciones.

Que sea “sin estado” quiere decir que no mantiene ni depende de condición preexistente alguna. En una Arquitectura de un Servicio (SOA), los servicios no son

dependientes de la condición de ningún otro servicio. Reciben en la llamada toda la información que necesitan para dar una respuesta. Debido a que los servicios son "sin estado", pueden ser secuenciados (orquestados) en numerosas secuencias (algunas veces llamadas tuberías o pipelines) para realizar la lógica del negocio.

Los servicios sirven para mejorar la eficiencia, ya que evitan duplicidades de código dentro de la arquitectura general de la empresa. Un ejemplo es el servicio de login, podríamos usarlo para todas las aplicaciones que necesiten logearse. De esa forma podemos adjudicar de forma centralizada permisos a los usuarios independientemente de desde donde se logeen.

Los servicios permiten exponer a terceros, incluso de fuera de la empresa, los servicios de la empresa de forma segura, ya que podemos establecer la entrada a nuestras aplicaciones vía servicio que no tiene puertas traseras.

Según van avanzando las tecnologías de servicios van mejorando los servicios usando SOA (Service Oriented Architecture). Es lo que define la utilización de servicios para dar soporte a los requisitos del negocio.

Permite la creación de sistemas altamente escalables que reflejan el negocio de la organización, a su vez brinda una forma bien definida de exposición e invocación de servicios (comúnmente pero no exclusivamente [servicios web](#)), lo cual facilita la interacción (e integración) entre diferentes sistemas propios o de terceros.

Los servicios representan la tecnología más avanzada hoy por hoy y puede significar una complejidad grande.

En Java, los servicios los podemos dividir en dos tipos, los **servicios propios de Java**, que quiere decir que tanto el cliente como el servicio están hechos íntegramente con Java, por ejemplo EJB, JMS y JMX que son APIs para servicios de Java, responden a distintos tipos de servicios. Además Java incorpora otra serie de **servicios que no son de especificación Java**, como los Web Services (WS) que pueden ser de tipo SOAP (Simple Object Access Protocol) o REST (Representational State Transfer). Hay una gran variedad de servicios y cada servicio es adecuado para un tipo de tarea. Lo lógico es que conociendo nuestra necesidad elijamos la forma más adecuada de implementarla.

Servicios Web

Web service es una especificación externa a Java, que nos dice lo siguiente:

Un servicio Web se va a definir (describir) mediante un fichero XML. Ese fichero está compuesto por una serie de etiquetas llamadas [WSDL](#) (Web Services Description Language). Es una descripción basada en XML de los requisitos funcionales necesarios para establecer una comunicación con los servicios Web. En este fichero se especifica el formato, las peticiones y el formato de las respuestas. Por lo tanto **Web service es una implementación basada en XML** y esta basado en **RPC** (remote procedure control), por lo tanto vamos a llamar a métodos.

El WSDL, también llamado contrato, es la parte más importante del Servicio Web y a pesar de que los servicios web pueden construirse creando primero las clases y a partir de estas clases generar automáticamente el archivo WSDL, si el web service va a tener cierta complejidad es MUY recomendable usar la opción llamada “contract first”, es decir, crear primero el contrato y a partir de este crear las clases, ya que la generación automática del WSDL puede tener complicaciones, sobretodo cuando se trabaja con objetos y colecciones o arrays. Crear el “contrato primero” tiene además la ventaja de que, una vez creado se puede entregar a los clientes del web service para que puedan ir trabajando en el cliente sin tener que esperar a que se termine el servicio.

Estructura del WSDL

La estructura del WSDL tiene los siguientes elementos:

Tipos de Datos

<types>: Esta sección define los tipos de datos usados en los mensajes. Se utilizan los tipos definidos en la especificación de esquemas XML.

Mensajes

<message>: Aquí definimos los elementos del mensaje. Cada mensaje puede consistir de una serie de partes lógicas. Las partes pueden ser de cualquiera de los tipos definidos en la sección anterior.

Tipos de Puerto

<portType>: En este apartado definimos las operaciones permitidas y los mensajes intercambiados en el Servicio.

Bindings

<binding>: Especificamos los protocolos de comunicación usados.

Servicios

<service>: Conjunto de puertos y dirección de los mismos. Esta parte final hace referencia a lo aportado por las secciones anteriores.

Con estos elementos no sabemos que hace un servicio pero si disponemos de la información necesaria para interactuar con él (funciones, mensajes de entrada/salida,

protocolos...).

A continuación se muestra un ejemplo de un documento WSDL y sus diferentes secciones. En este ejemplo concreto se implementa un servicio que muestra, a partir del nombre de un valor bursátil, su valor actual en el mercado.

```
<definitions name="StockQuote"
    targetNamespace="http://example.com/stockquote.wsdl"
    xmlns:tns="http://example.com/stockquote.wsdl"
    xmlns:xsd1="http://example.com/stockquote.xsd"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
        <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
            <element name="TradePriceRequest">
                <complexType>
                    <all>
                        <element name="tickerSymbol" type="string"/>
                    </all>
                </complexType>
            </element>
            <element name="TradePrice">
                <complexType>
                    <all>
                        <element name="price" type="float"/>
                    </all>
                </complexType>
            </element>
        </schema>
    </types>

    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>

    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
            <input message="tns:GetLastTradePriceInput"/>
            <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>

    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
            <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
            <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>
</definitions>
```

RPC se basa en corba, y corba usa RMI-IIOP. A su vez XML responde también a el protocolo de comunicacion de corba que se llama SOAP.

Esas peticiones y respuestas serán a la vez ficheros de texto y siempre viajaran por protocolo HTTP, al ser una especificación abierta el protocolo debe ser lo mas estándar posible, de ahi HTTP. Este servicio se debe desplegar en un servidor de servicios Web para quedar a la espera de las peticiones de los clientes. De tal forma que el cliente lo único que conoce es el servicio con el que se comunica, no quien es el proveedor.

En Java, los servicios Web se implementan sin necesidad de un servidor Web, nosotros ya tenemos las aplicaciones Web que pueden aceptar todo tipo de peticiones. Por lo tanto **en Java, un servicio web es una aplicación web normal**. En una aplicación de este tipo las peticiones las recibe un servlet, por lo tanto **en Java los servicios web son servlets**. Con el dato de que todas las implementaciones de servicios web (AXIS , AXIS2, XFIRE, JWS, etc) generan automáticamente el servicio Web. Además el fichero descriptor de servicio (WSDL) lo generan automáticamente.

Con lo cual **nosotros solo tenemos que definir las clases de java que van a alimentar el servicio**. Es decir, la lógica del mismo.

Cuando vayamos a enviar objetos, como en el caso de un web service de login que puede necesitar enviar un objeto usuario con todos los datos del usuario incluidos roles y tareas, es mejor **serializar el objeto para que se genere correctamente el WSDL**, ya que de otra manera seguramente tendría problemas para generar correctamente el WSDL. Para ello podemos **serializar** a JSON (JSONSerializer (static)) antes de enviarlo. JSON necesita unas librerías que tenemos que asegurarnos que existen en nuestro proyecto o bien importarlas. Esto ya lo vimos en la ultima parte de la sección de AJAX en este master donde ya serializamos con JSON. El tema de serializacion a JSON es muy efectivo y a la vez es complicado y hay que tener en cuenta muchas cosas sobretodo cuando se van a serializar objetos.

Si trabajamos con hibernate hay que asegurarse que la carga vaga esta resuelta (objeto Criteria) o bien ponemos a null la propiedad (el campo de la tabla que este relacionado) que usa carga vaga. Hay un ejemplo en la clase loginImpl.java, método obtener_Perfil() del proyecto j510-ws-login que es un servicio. Por otro lado tenemos que crear un método nuevo en el DAO (UsuariosDAO, método buscar_conRol()) para resolver la carga vaga y asi obtener los roles de un usuario usando criteria y su método setFetchMode del objeto Criteria, además de uniqueResult().

Se aconseja estudiar la **API de hibernate** para conocer bien esta clase **Criteria** ya que tiene una enorme cantidad de posibilidades.

Por otro lado, el cliente que recibe un objeto serializado, es decir texto en formato, por ejemplo JSON, y tiene que convertirlo en un objeto por ejemplo Usuario, en el hipotético caso de un servicio de login que envía un string con el nombre de un usuario y recibe un JSON que es un objeto Usuario serializado, el cual contiene el perfil de dicho usuario, lo que tiene que hacer es convertir el texto enviado por el servicio a un

objeto de tipo Usuario.

Para hacer esa **conversión de JSON a objeto**, por supuesto necesitamos las librerías de JSON en nuestro proyecto de cliente. Usaremos la clase JSONObject y JSONConfig, algo así:

```
public Usuarios obtener_Perfil(String nombreUsuario) {  
    String texto_usuario = servicio.obtener_Perfil(nombre_Usuario);  
    JSONObject objeto_JSON = (JSONObject)  
        JSONSerializer.toJSON(texto_usuario);  
    JsonConfig clase_aconvertir = new JsonConfig();  
    Clase_aconvertir.setRootClass(Usuarios.class);  
    Usuario = (Usuarios) JSONSerializer.toJava(objeto_JSON,  
        clase_aconvertir);  
    return Usuario;  
}
```

Podemos consultar los asuntos sobre JSON en la web <http://json-lib.sourceforge.net/> dentro de snippets (<http://json-lib.sourceforge.net/snippets.html>). También encontraremos ahí las librerías que necesitamos para JSON clasificadas por lenguajes.

Los servicios pueden ser llamados por otros servicios Web. Este es ya terreno de arquitectura de servicios (SOA) programación orientada a servicios, la cual coordina este entrelazado de servicios.

El cliente puede hacerse en Java, puede ser una aplicación web o puede ser una aplicación J2SE.

Podemos decir que tanto el servicio como el cliente se generan de forma automática con Java.

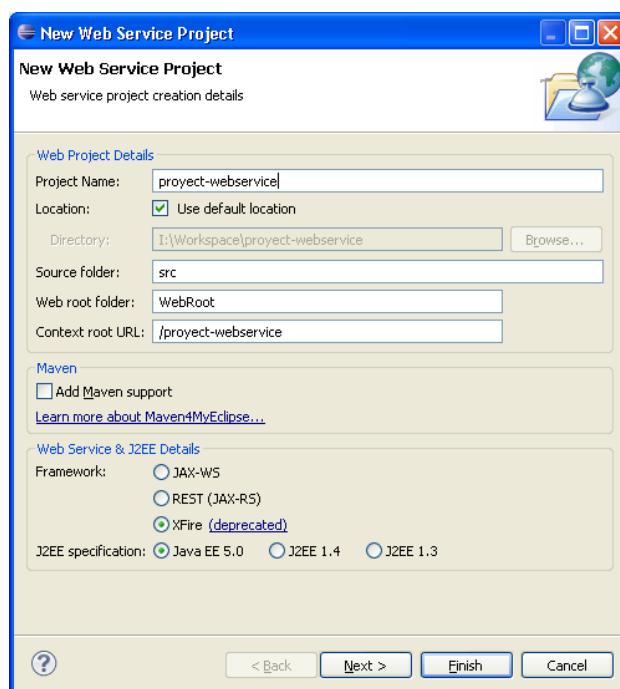
Respecto a las implementaciones, hemos hablado de AXIS, XFIRE y JWS. Si vamos a hacer una estructura de servicios sencilla usaremos XFIRE o JWS. JWS requiere el JDK 5 o superior, es propiedad de Sun. XFIRE admite el JDK 3 o superior y esa es la razón por la que nos decantaríamos por XFIRE antes que por JWS. Si vamos a usar una estructura compleja, con mucha seguridad, etc., usaremos AXIS. AXIS, en Eclipse, hay que cargarlo aparte.

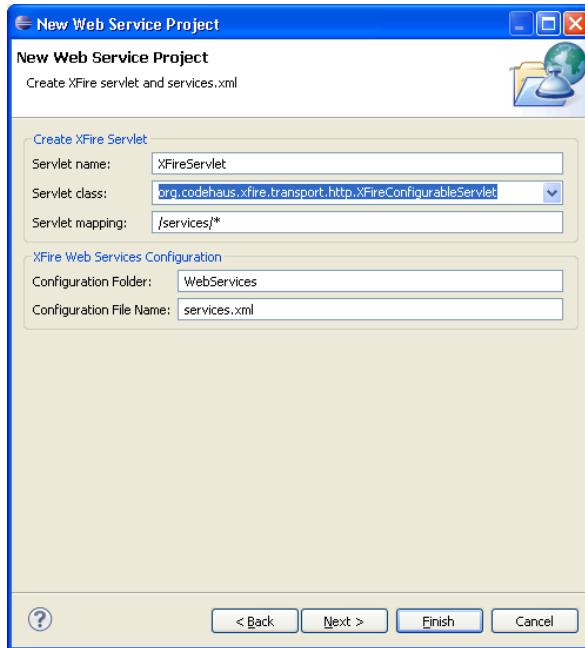
El servidor Tomcat no puede con aplicaciones muy grandes o muy concurridas respecto a tráfico o instancias. Habría que usar otros servidores como Weblogic o Websphere.

Creacion de un Servicio Web con MyEclipse

Para crear un Servicio Web con MyEclipse eljimos New – Web Service Project, le damos nombre y en “Framework” se elige la implementacion a utilizar. En principio vamos a usar XFire, luego elegimos la especificación que será Java EE 5.0. Damos a Next >.

Si elegimos JAX-WS (JWS) también le daremos nombre pero no habrá botón next, directamente le damos a Finish y después creamos un paquete (servicio) y en ese paquete creamos una Clase con los métodos que vamos a usar en el servicio. Luego nos pedirá esa clase cuando vayamos a indicarle si va a ser un cliente o un servicio (new – other – web service...).



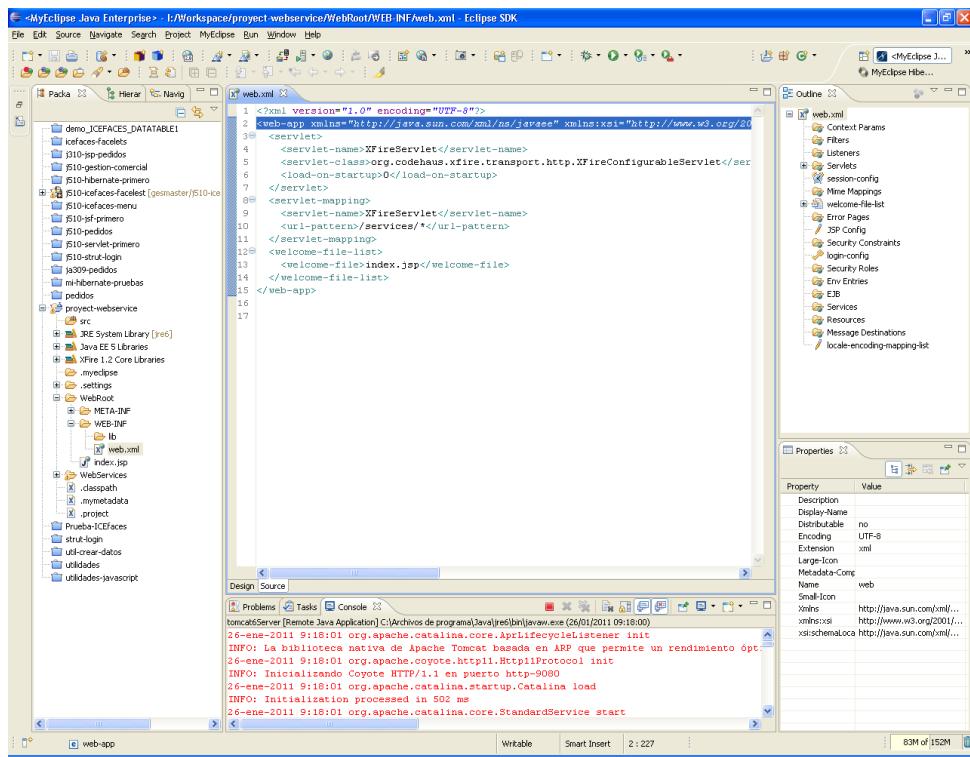


En servlet class elegimos la primera opción (en este caso), en servlet mapping podemos dejarlo como esta o lo podemos cambiar, es solo un mapeado. Tambien le vamos a decir el fichero descriptor, dejamos el “Configuration Folder” que viene por defecto y en “Configuration File Name” escribimos el nombre que queramos o mejor lo dejamos como viene, pulsamos next.



En esta pantalla se eligen las librerías a importar la podemos dejar como esta, con el XFire 1.2 Core Libraries, para un caso sencillo será suficiente con esa librería. Pulsamos finish y se genera todo lo que necesitamos.

El fichero web.xml ya viene hecho:

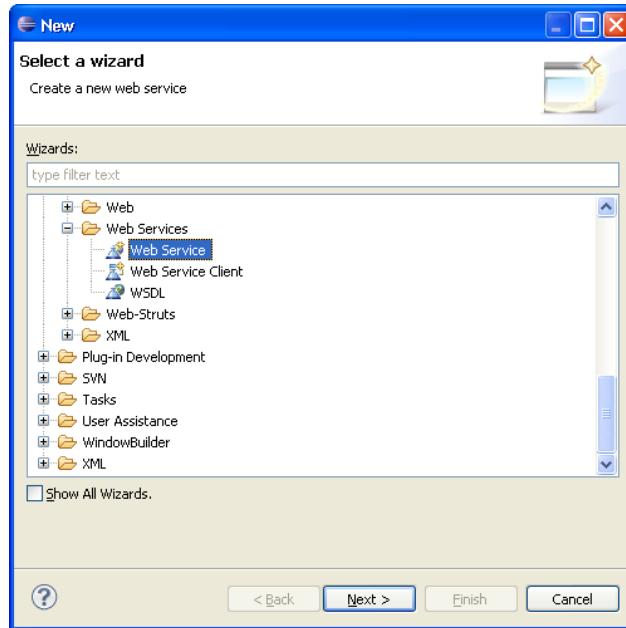


El servlet ya está hecho y declarado como que es un servicio web.

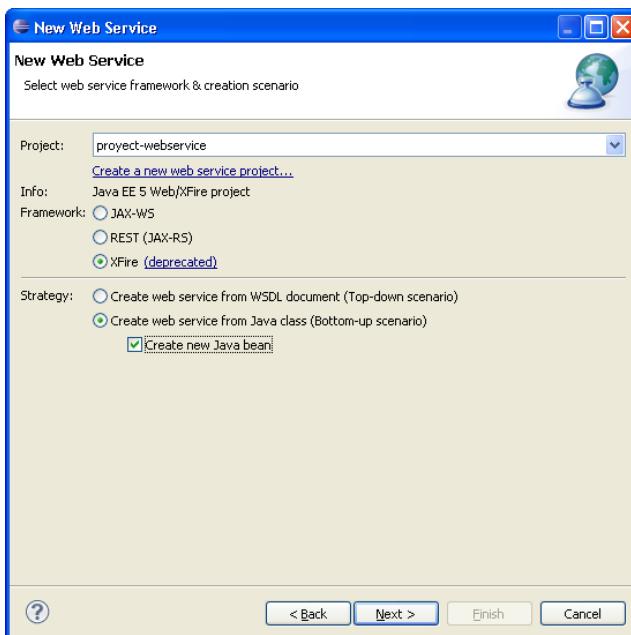
Crearemos un paquete para Hibernate y le damos capacidades de Hibernate al proyecto si lo necesitamos.

A partir de aquí lo que tenemos que hacer son las clases que alimentan el servicio. En el master tenemos algunos ejemplos de servicio que pueden valernos como referencia y ejemplo.

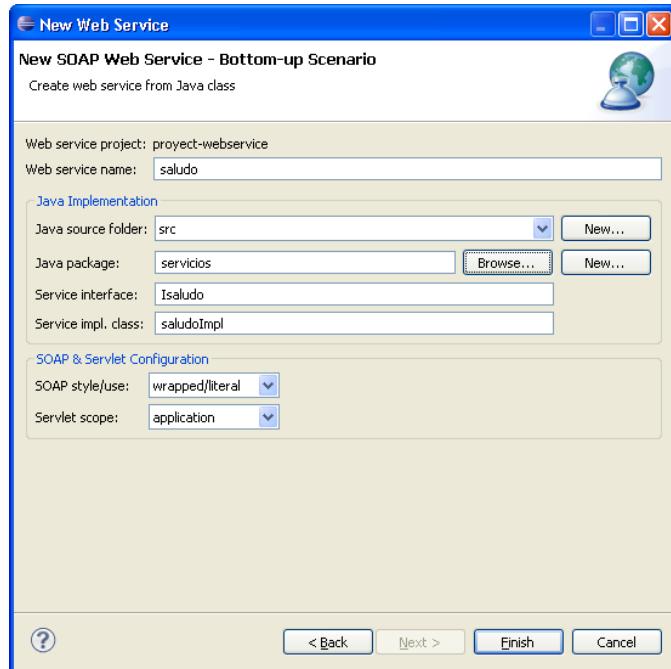
Para declarar el proyecto como web service se pulsa con botón derecho en el nombre del proyecto y se elige new – other y entonces en MyEclipse – Web Services elegimos WebService si es el servicio o Web Service Client si este proyecto va a ser el cliente.



Pulsamos en Next > y se abre esta ventana.



Elegimos el proyecto del combo, elegimos el framework que ya habíamos elegido antes, al crear el proyecto. En la sección de Strategy podemos elegir si vamos a crear el servicio web a partir de un WSDL que ya ha sido creado previamente y lo tenemos nosotros en el proyecto o a partir de una clase Java que es el caso cuando el web service va a ser creado en este momento. Pulsamos Next >.



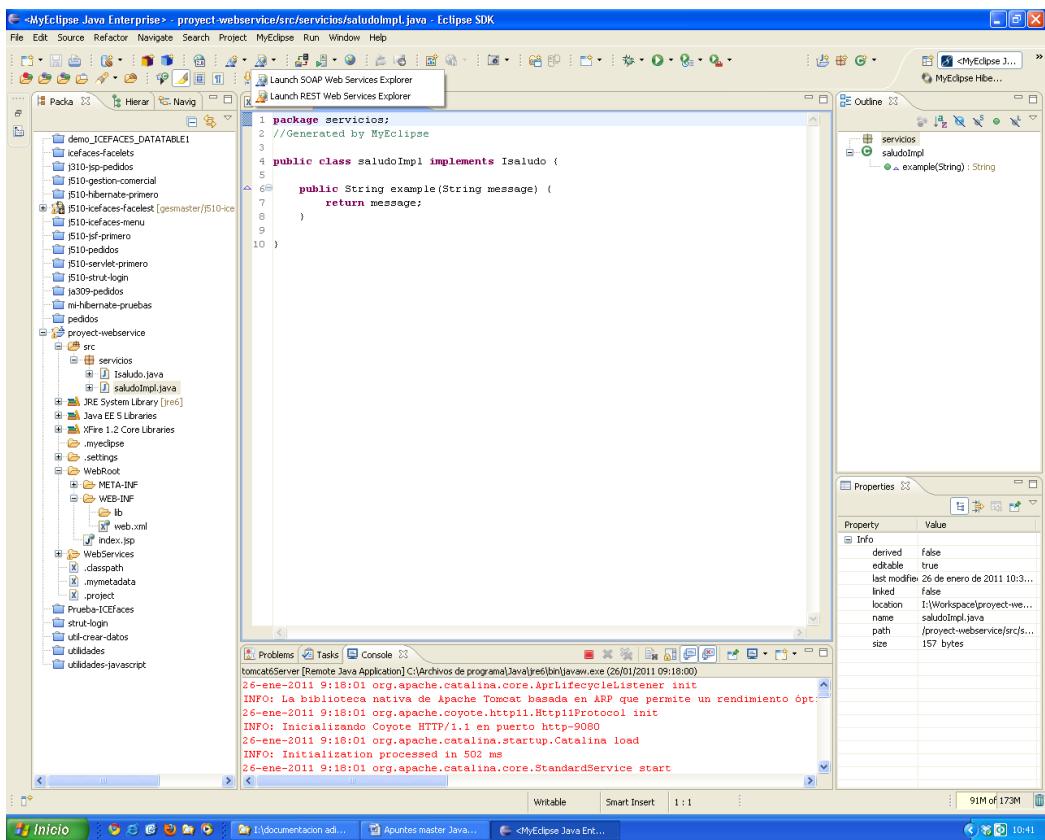
En esta pantalla damos nombre al servicio (para el mapeado), e indicamos el paquete donde va a estar. En “Service Interface” y “Service impl. Class” podemos dejarlo como esta, son los nombres que tendrán la interfaz y la clase que va a generar. La parte de abajo (SOAP & Servlet Configuration) mejor no tocarla.

Pulsamos Finish y con eso el servicio ya está hecho.

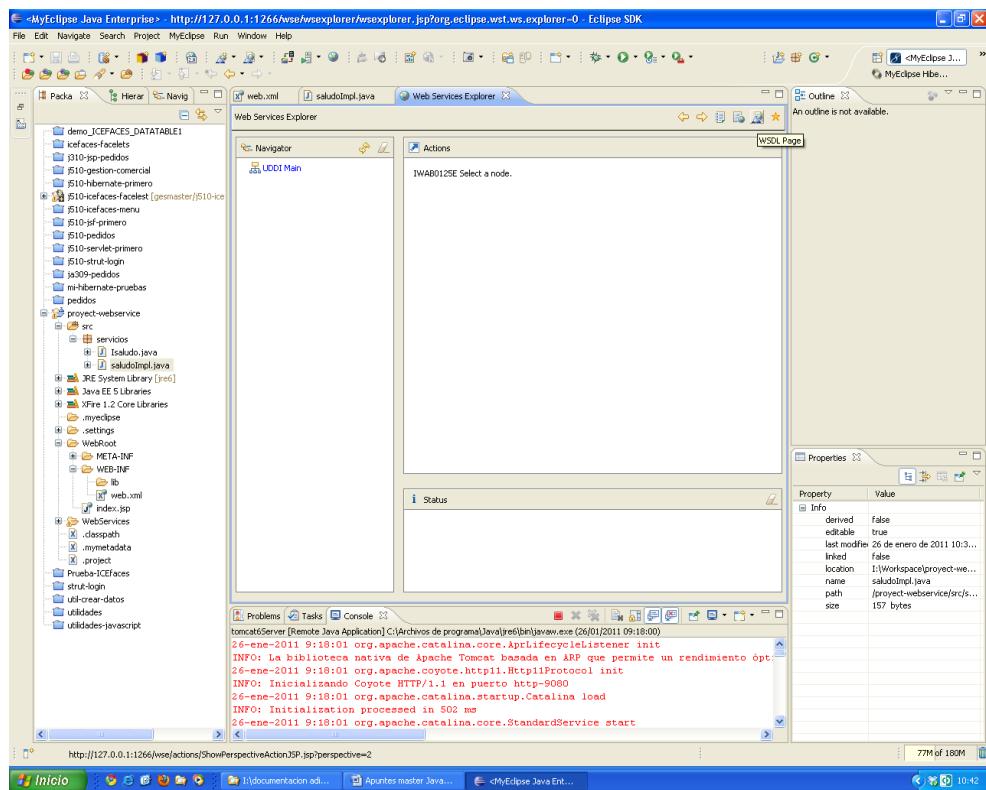
XFire, a través del servlet que lleva, generará en tiempo de ejecución el fichero descriptor WSDL. Se generará en el paquete donde hemos indicado que tendremos las clases del web service, además creará una interfaz y una clase que implementa esa interfaz, lo que tenemos que hacer es adecuar la interfaz a nuestras necesidades indicando la firma de los métodos e implementar los métodos de la interfaz en la clase escribiendo en ellos la lógica que necesita nuestro web service.

Para probar nuestros proyectos de web service no necesitamos hacer un cliente, sino que podemos usar (por cortesía de Microsoft) un **service explorer**, que es una aplicación que podemos bajar usando un internet explorer o usando el que MyEclipse ya tiene incorporado.

Es un ícono con dos opciones: Launch SOAP Services Explorer y Launch Rest Web Services Explorer, elegimos la primera en este caso, Launch SOAP Service Explorer.



Eso abre una nueva vista



Esta nueva vista tiene un botón “WSDL Page” que, hay que pulsarlo (al hacerlo cambia). A continuación pulsamos en “WSDL Main” y escribimos los datos que nos pide que es la cadena de conexión al WSDL, que en nuestro caso específico es <http://localhost:8080/j510-ws-primeros/services/saludo?WSDL>

La dirección se compone de Nombre del servidor (dominio, IP, puerto, etc), aplicación, nombre mapeado, y ?WSDL

Pulsamos en GO y en el lado izquierdo aparecerán el árbol de directorios del web service y los métodos que podemos usar, si los pulsamos podemos probarlos en el lado derecho de la vista.

La aplicación es intuitiva y podemos ir siguiendo las indicaciones que nos da.

Hay que tener cuidado con una incompatibilidad de Spring 3 con XFIRE, hay que cambiar en el fichero services.xml el atributo que aparece en la etiqueta bean (xmlns), hay que ponerlo en la etiqueta services.

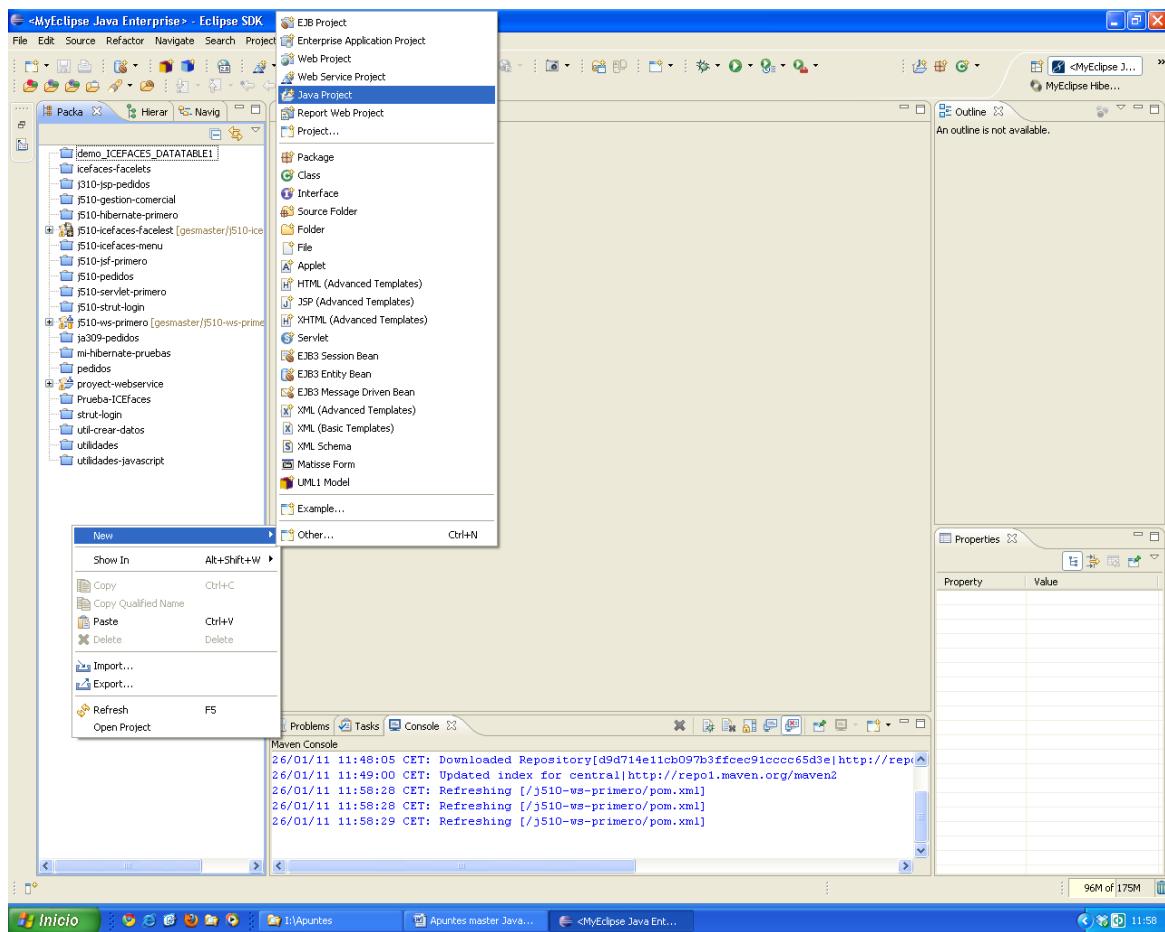
CREACION DEL CLIENTE DEL WS

Cuando vayamos a crear el cliente para usar nuestro web service podemos usar un cliente J2SE o J2EE, en cualquier caso le daremos capacidades de web service dando con botón derecho en el nombre del proyecto y luego siguiendo los pasos vistos antes pero eligiendo “Web Service Client” en vez de Web service.

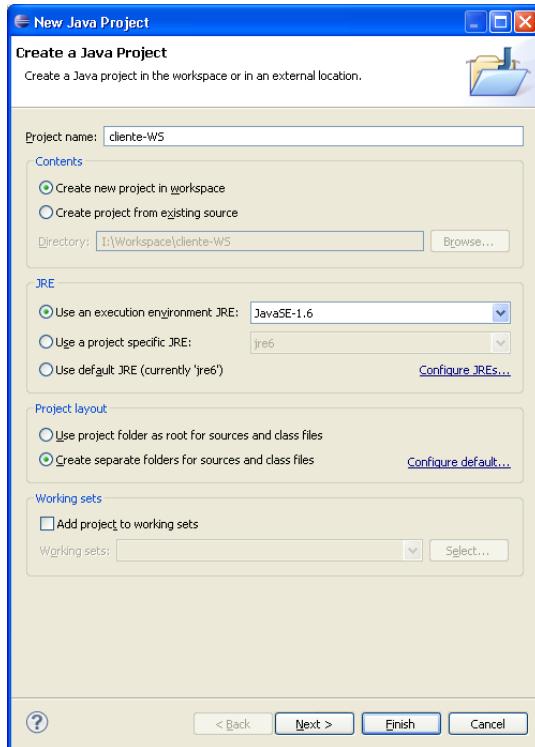
Debemos tener el servidor levantado y corriendo para poder hacer el cliente ya que le vamos a preguntar durante el proceso de creación.

El servicio debe estar hecho antes de hacer el cliente, al menos debemos tener definidos los métodos de la interfaz.

En el caso de un cliente J2SE hacemos esto:

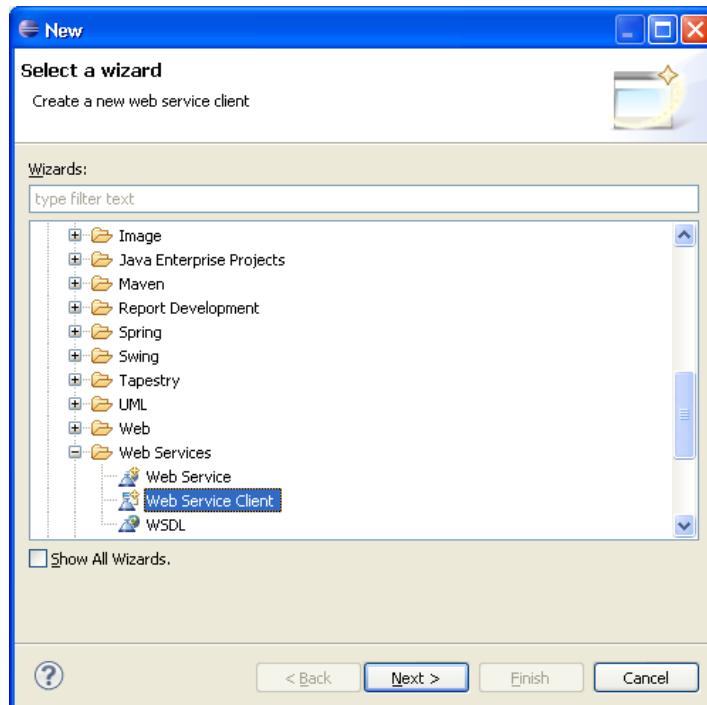


New – Java Project



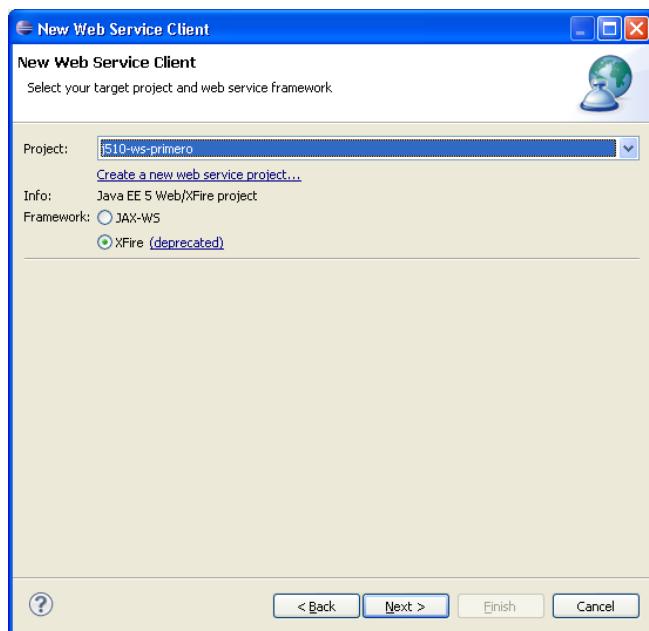
Pulsamos Finish.

Le damos capacidades de Web Service: New – Other – MyEclipse – Web Services - Web Service Client



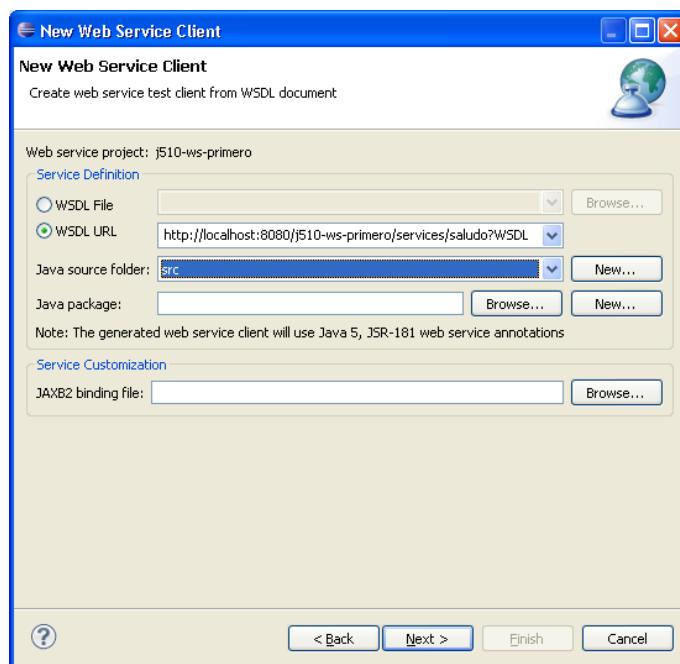
Next >

Elegimos el proyecto que sea el web service y el framework que hemos elegido que debe ser el mismo con el que hayamos creado el web service anteriormente.

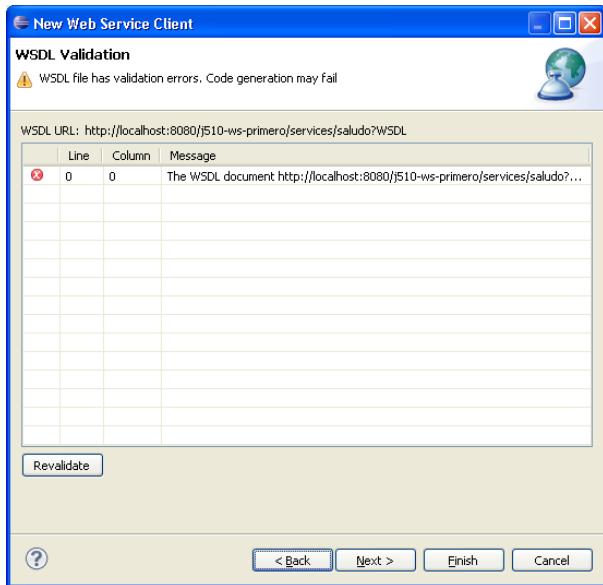


Next >

En la siguiente ventana especificamos el WSDL si lo tenemos nosotros le decimos donde esta o bien le damos url escribiendo la cadena como hicimos antes.



Pulsamos next y comienza la validación



Si la hace bien pulsamos en Finish.

Al crearse el cliente lo que se genera es un paquete nuevo y unas clases.

El paquete que se crea en el caso de XFire se llama siempre service o servicio y crea clases por cada método que tenga el servicio, una clase para cada petición o mensaje de petición y otra para cada mensaje de respuesta. Esto lo conoce por que habrá leído el WSDL.

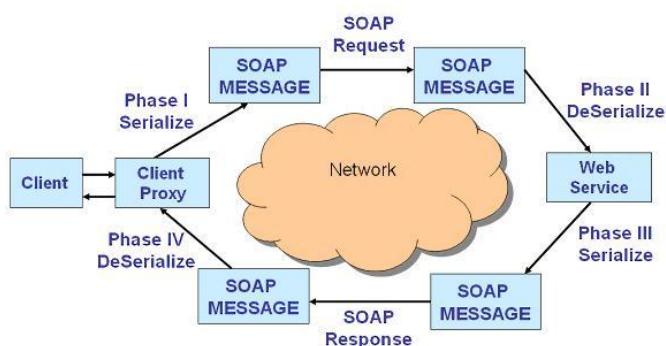
Es muy recomendable crear una clase de fachada en el servicio para que sea la que recibe las comunicaciones del cliente de forma que aislamos más el servicio y el cliente no sabe ni siquiera que estamos usando un servicio para obtener respuesta a sus peticiones.

En resumen el flujo completo de funcionamiento de un web service es así:

A partir, por ejemplo, de una pagina jsp que, por ejemplo, sea un formulario de login con usuario y contraseña al pulsarse el botón de login vamos a un método de acción del ManagedBean correspondiente, como nuestra aplicación necesita de un servicio de login para poder autorizar o no al usuario, lo que hace el método de acción es llamar a un método de la fachada que, en vez de acceder a nuestra base de datos local, usa las clases creadas cuando declaramos a nuestra aplicación como cliente Web Service y además serializa los datos que se deben enviar. La fachada instancia a esas dos clases creadas (loginClient y loginPortType), esas clases tienen la información necesaria acerca del servicio, saben como conectarse a el y como hacer las peticiones. A partir de aquí el flujo viaja hasta el servicio, que esta esperando peticiones.

En el lado del servicio las peticiones llegan a un método de una clase del servicio. Durante la creación del servicio, al declarar el proyecto como Web Service XFire se crearon también, una interfaz y una clase que implementa esa interfaz. En esa interfaz definimos los métodos que vamos a crear en la otra clase y que son los que contienen la lógica para atender las peticiones de los clientes. Es en esta clase donde recibiremos las peticiones, y de ahí lo pasamos a la fachada del servicio. En la fachada es donde tratamos los datos deserializándolos, y desde ahí consultando a la base de datos. El método debe devolver el dato, serializado de nuevo, que el método del cliente está esperando. Esto en el caso de XFire, si fuera JWS el flujo desde el cliente seria todo igual pero en el servicio, la diferencia es que en vez de haber una interfaz y una clase que la implementa, lo que hay es que, antes de crear el servicio, escribiremos una clase con los métodos que vamos a necesitar para la lógica del web service, entonces al crearse el web service de tipo JWS se crea un delegado que lee esa clase y conoce los métodos que tiene. El delegado recibe la petición del cliente y pasa el flujo a la clase que es la que tiene el método que devuelve el dato que el cliente está esperando.

XML Web Services Architecture



Esquema del flujo de funcionamiento de un Web Service típico sencillo

Servicios Web Complejos

Los servicios web pueden ser complejos, eso quiere decir que son **servicios que llaman a otros servicios**, es decir los servicios pasan a ser clientes y llaman a otros servicios.

Dependiendo del tipo de información que un servicio pueda exponer es posible que necesite filtrar a los usuarios para poder darles esa información, por ejemplo información confidencial o sensible acerca de un usuario.

Para poder hacer eso debe haber un mecanismo que me permita reconocer a los usuarios autorizados para acceder a determinada información, con lo cual tenemos un servicio de login que necesita de otro servicio de verificación de clientes para responder a su cliente.

Servicios Web tipo REST

La opinión de si este es un servicio web es discutible. Los servicios web conllevan cierta complejidad respecto a clientes y servicios, etc. Alguien (Roy Fielding) tuvo la idea de facilitar este montaje de los servicios web, usando las herramientas de Java para poder hacer las cosas, ya que al fin y al cabo se trata de una implementación cliente - servidor. De ahí nacieron los servicios REST.

Cualquier cliente Java (navegador) realiza peticiones a una aplicación servidor Java, el servlet recibe la petición y en función de dicha petición, ejecuta el método de la clase que nos interesa para satisfacer esa petición. REST propone usar HTTP de forma más a fondo y más coherentemente, usando peticiones POST, GET, PUSH y DELETE. La mayor parte de las cosas que hacemos son procesos **CRUD** (altas - **C**reate, consultas - **R**ead , modificaciones – **U**pdate, y borrados - **D**elete). En teoría se debería usar Get para consultas, Put para las altas, Delete para los borrados y Post para las modificaciones.

A partir de ahí podríamos usar cualquier método para atender las peticiones, también usar anotaciones e inyección para no usar ficheros descriptores XML.

En JEE 5 podemos usar REST añadiendo las clases necesarias, Tomcat 6 no soporta REST. En JEE6 REST si es parte del JDK, pero haría falta Tomcat 7 o GlassFish 3.

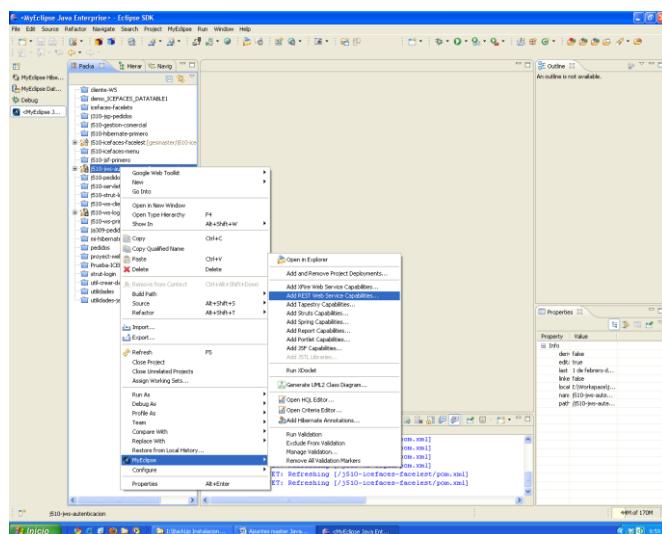
Para la serialización se suele usar JSON, aunque esto es una decisión del programador.

Creacion de un Servicio Web tipo REST en MyEclipse

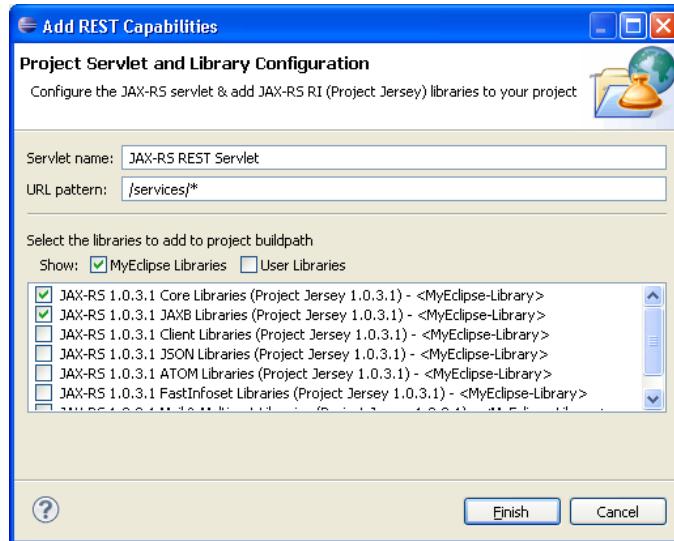
Para crear un proyecto Web Service REST con MyEclipse hacemos lo siguiente:

Para empezar creamos un proyecto Web normal (new – web Project), y el servicio REST como es un servlet hay que mapearlo como cualquier otro servlet, con la etiqueta <servlet> y <servlet-mapping>, el servlet debe ser un servlet específico que es **com.sun.jersey.spi.container.servlet.ServletContainer**.

Al proyecto hay que añadirle las librerías que vamos a necesitar bien desde Sun o bien desde el “proyecto jersey”. Las librerías están en MyEclipse Libraries y son todas las de “Project Jersey”. Se añaden dando **capacidades de REST** al proyecto (en MyEclipse 8.6.1).



Aparece la siguiente ventana:



Las dos marcadas por defecto son las que necesitamos.

Pulsamos Finish. Se añaden las librerías y se hace el mapeo automáticamente en web.xml.

Como hemos dicho tendremos que crear nosotros las clases y anotarlas, a mano.

La anotación de la clase es **@Path**, que sirve para indicar las peticiones que vamos a atender, un ejemplo sería **@Path (“/login”)**, es lo que viene en la URL. La anotación **@GET** indica que el método a continuación atenderá las peticiones de tipo GET y **@Produces** indica el tipo de respuesta que deberá ser text/plain, así: **@Produces (“text/plain”)** es el tipo MIME.

Para conocer las anotaciones de REST hay que bajar la API de REST y estudiárselas. Eclipse también tiene una ayuda con un tutorial muy útil sobre todo esto y cualquier otro tema.

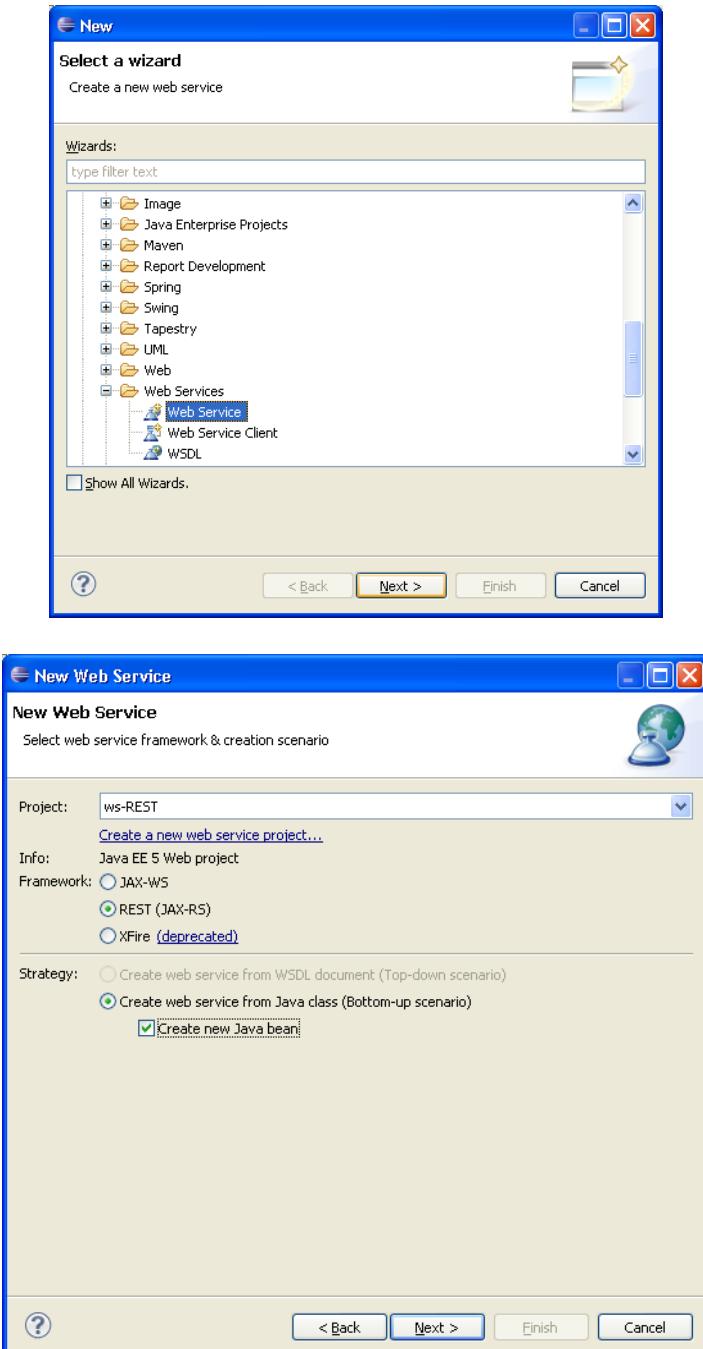
Despues de las anotaciones viene el método, el cual tendrá los parámetros inyectados usando la notación **@QueryParam** seguido del parámetro, por ejemplo:

```
@GET
@Produces ("text/plain")
Public String validarDatos(@QueryParam("nombre") String nombre,
@QueryParam("clave") String clave{
    // logica para el login
}
```

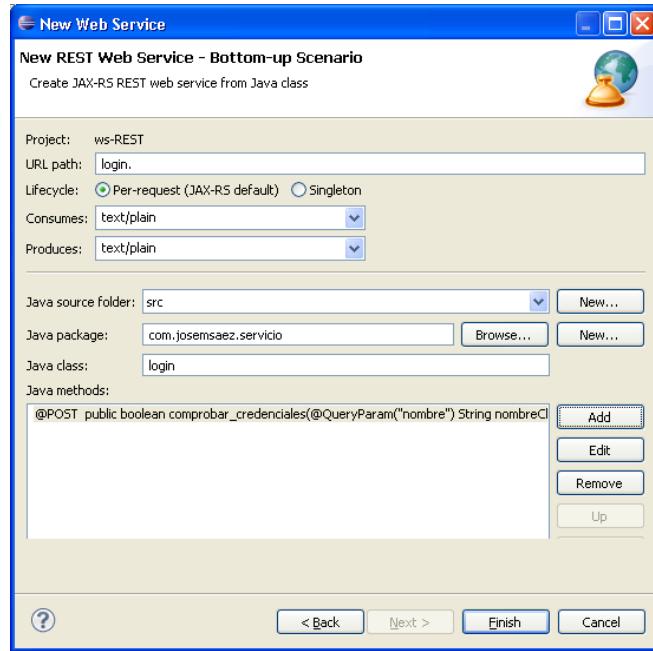
Como se puede ver los parametros se componen de la notacion seguido del parámetro, la notacion indica como llega en la URL de la petición.

A continuación le damos capacidades de Web Service tipo REST vamos a New – Other

– MyEclipse – Web Services - Web Service



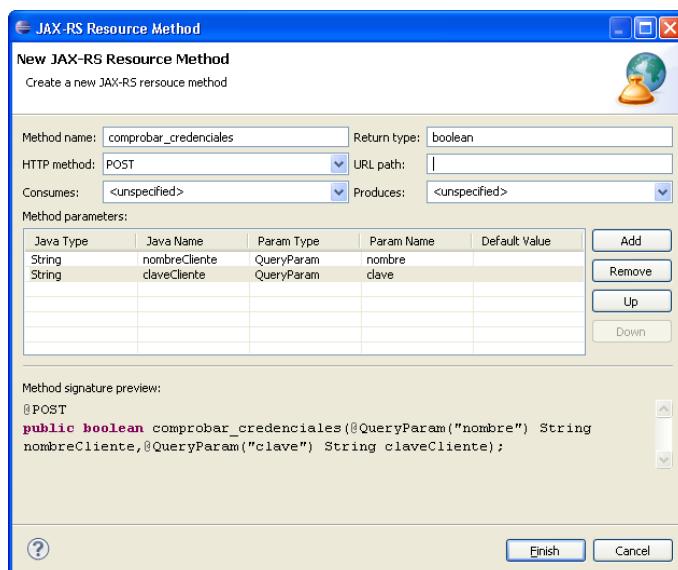
Elegimos el proyecto del desplegable Project, elegimos la opción REST en Framework y en Strategy, como es un web service de tipo REST, la única opción será Create Web Service from Java Class, lo marcamos como se ve en la imagen de arriba. Pulsamos Next >.



Escribimos el URL path (login), java class es lo que será el @Path, elegimos el lifecycle (per-request), consumes text/json (por ejemplo) según lo que nos vaya a llegar y en produces es lo que va a devolver, en java package elegimos el paquete que hayamos creado para guardarla

Se pulsa add para añadir los métodos, tantos como necesitemos.

Al pulsar add aparece la siguiente ventana para el método. Si pulsamos Finish se termina la configuración, y se crea la clase con los métodos que hemos especificado.



Hay que rellenar los campos para que se creen los métodos y se pulsa add para añadir el

método, una vez añadido se pueden especificar los campos así hacemos con cada parámetro del método que vayamos a necesitar en nuestra clase. Se puede ir viendo la firma en la parte de abajo.

La clase generada en este ejemplo se llamaría login.java y es esta:

```
package com.josemsaez.servicio;

@Produces("text/plain")
@Consumes("text/plain")
@Path("login.")
public class login {

    @POST
    public boolean comprobar_credenciales(
        @QueryParam("nombre") String nombreCliente,
        @QueryParam("clave") String claveCliente) {
        throw new UnsupportedOperationException("Not yet
implemented.");
    }
}
```

Ahora solo habría que escribir la lógica. Quitando el throw que nos pone para las pruebas unitarias.

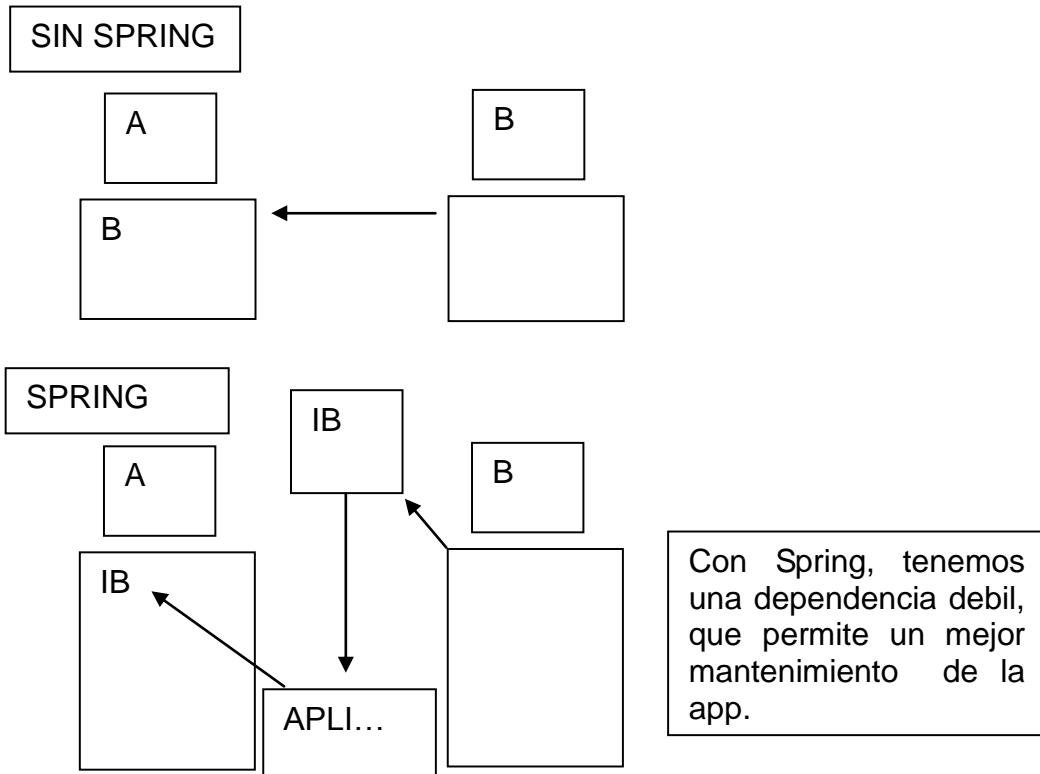
Spring (ver cap. 5 y 6 libro)

Algo a tener muy en cuenta en la programación orientada a objetos, y en concreto en Java, es que las clases tengan la menor dependencia posible entre ellas y estén lo mas aisladas posible entre sí. Lo que se llama **bajo acoplamiento y alta cohesión**. Todo eso contribuyó al nacimiento de lo que se llama **middleware** (capa intermedia), esto en Java lo representa Spring.

Spring es un marco de trabajo, creado por la empresa Spring Source, que ahora es propiedad de vmWare. Es un framework de código abierto.

La versión 3 de Spring está pensada para un desarrollo modular, es enormemente amplia debido a eso. No hace falta implementar todo, sino solamente la parte que nos interesa, además del modulo central llamado **core**.

Spring es un framework que se usa para conseguir desacoplamiento implementando dos patrones; **el patrón de DI** (inyección de dependencias) antiguamente llamado patrón Hollywood (no me llames que ya te llamare yo) o Inversión de control (IOC). Spring por si mismo no hace casi nada, no es esa su misión, su misión es actuar de intermediario y lo que haya por detrás no le importa, por eso se dice que es un middleware. Eso lo hace actuando como fabrica de objetos (object factory o BeanFactory) y poniéndolos a disposición. Los objetos los crea según se lo decimos nosotros. Hasta la versión 2 esto se hacía con un fichero XML, a partir de la 2.5 se usan anotaciones. El otro patrón es la **Programación Orientada a Aspectos** (AOP), la veremos mas adelante.



No necesariamente Spring va a crear todos los objetos de la aplicación, solamente los que necesitemos crear con Spring. Normalmente le pediremos solo los objetos de alto nivel.

Spring tiene dos inconvenientes; primero, suele ser tremadamente redundante, cada cosa puede hacerse de varias formas; el otro defecto es que los nombres de las clases son muy largos, excesivamente descriptivos.

Para trabajar con el **CORE**, tendremos un fichero descriptor en XML (Spring 2.0), llamado normalmente **ApplicationContext.xml**, es recomendable tener varios ficheros descriptores. Además necesitamos la clase de Spring que hace de **Object Factory**, que se llamará también **ApplicationContext**.

ApplicationContext es una interfaz, la clase que la implemente es lo que llamaremos, la fabrica de beans (BeanFactory). Esta clase no solo crea los objetos sino que además crea un repositorio de estos objetos. Con esto se puede implementar un patrón Singleton o no. Además se aplica un ciclo de vida a estos objetos. Esta clase es la que conoce todo lo que se ha descrito en el archivo descriptor applicationContext.xml, es decir todo sobre la estructura y los beans de nuestra aplicación.

Normalmente solo tendremos una clase ApplicationContext, ya que de tener mas de una tendremos tambien más repositorios de objetos que no tendrán nada que ver entre si. Al ser única es importante que sea visible a todo el programa, con lo cual en el caso de una

aplicación J2EE, es candidato a vivir en el ámbito de la aplicación. En J2SE el ApplicationContext será externo y así tendrá alcance a todo el programa.

Como el ApplicationContext es una interfaz, no tiene constructor por lo tanto para obtener los objetos se hace a través del archivo de configuración applicationcontext.xml.

Dependiendo de donde se ubique físicamente el fichero descriptor habrá diferentes contextos, por ejemplo, si esta en nuestro proyecto el contexto será **ClassPathXmlApplicationContext**, si el fichero esta en una carpeta externa al proyecto, entonces el contexto a usar deberá ser **FileSystemXmlApplicationContext** este permite indicar la trayectoria de carpetas para localizar el fichero descriptor. En caso de que el fichero descriptor se encuentre fuera de nuestra máquina, el contexto será **XmlWebApplicationContext**.

```
ApplicationContext xxx = new ClassPathXmlApplicationContext(ruta  
+ nomrefichero);
```

O bien

```
ApplicationContext xxx = new  
FileSystemXmlApplicationContext(ruta + nomrefichero);
```

O bien

```
ApplicationContext xxx = new XmlWebApplicationContext(ruta +  
nomrefichero);
```

Todos reciben la ruta y el nombre del fichero XML separando los paquetes o directorios con barras /.

Los objetos se definen mediante la etiqueta bean, una etiqueta por cada objeto en el fichero descriptor. Esa etiqueta lleva los siguientes atributos obligatorios:

ID = El nombre del bean. Nombre único y case sensitive.

class = La clase de la que se tiene que instanciar el objeto (reflexión). Paquete + clase.

Hay otros atributos optativos, pero si no se ponen mas, quiere decir que el applicationContext me puede servir objetos de esa clase usando los valores por defecto de los otros atributos optativos sin mas. Esas clases han de ser javabeans, admitiendo constructores con argumentos. Además **al no poner mas atributos, estamos indicando que usaremos el patrón Singleton (por defecto)**. También significa que lo creará aplicando la regla **lazy-init = false (por defecto)**. Carga vaga en falso, es decir se crea inmediatamente después de crearse el ApplicationContext.

Para cambiar ese comportamiento usamos los atributos:

scope = (cómo) Para indicar si queremos Singleton o no. Las opciones son:

singleton = (por defecto) único objeto cada vez

prototype = instancia un nuevo objeto cada vez, al contrario que singleton

request = (JEE), se almacena en el request (petición) y tiene ese alcance y vida

session = (JEE) se almacena en la sesión y tiene ese alcance y vida

lazy-init = (cuando) opciones true/false. Si es false (por defecto) quiere decir que el objeto se crea al principio, si es true el objeto no se crea hasta que no se pide.

La decisión de aplicar una opción u otra la decidimos nosotros meditadamente.

Una vez creado el objeto, ApplicationContext tiene la capacidad de, a los objetos creados, pasarles otros objetos, es decir **inyectarles las dependencias**. Para hacer eso, en el bean, debe haber una propiedad y unos métodos accesores para esa propiedad. Para especificar estas propiedades dentro de la etiqueta <bean> debe haber otra etiqueta <property> con el atributo **name** para el nombre de la propiedad, si el valor es constante ponemos el atributo **value**, puede ser un valor numérico aunque vaya entre comillas, se usa para dar valor a una de las propiedades del bean a través de su accesos y esa propiedad es la que lleva el nombre indicado por name. Con el atributo **ref** podemos pasarle otro objeto como propiedad, es decir **inyectarle una dependencia con el objeto aquí indicado, ref y value son incompatibles** se usa **una u otra**, no las dos. **En ref ponemos la ID del objeto (bean)** que queremos pasarle, es decir el bean que queremos inyectarle a través de su interfaz. Es decir cuando instancie esa interfaz, puede acceder a sus métodos, sinó no sabría como acceder a los métodos de la clase que esta instanciando y a los que antes accedia instanciando la clase.

```
<bean id="clase1" class="com.paquete.NombreClase1" />
<bean id="clase2" class="com.paquete.NombreClase2">
    <property name="inyeccionAClase1" ref="clase1">
    </bean>
```

En este ejemplo se describe como se inyecta a la clase2 una dependencia con la clase1, de forma que, cuando la clase2 instancie a la interfaz de la clase1, podrá acceder a los métodos que la clase1 habrá implementado de su interfaz con la lógica que le interesa. Con esto se pueden facilitar las pruebas unitarias fácilmente evitando el problema de las dependencias de unas clases con otras y además, como una interfaz puede tener varias clases que la implementen, podrían crearse distintas implementaciones de la misma interfaz e inyectar unas u otras a las distintas clases según nos interese una implementación u otra.

Como la interfaz ApplicationContext conoce todo lo que se ha definido en el descriptor applicationcontext.xml, podriamos “instanciar” los beans que se han declarado en ese archivo de esta forma:

```
ApplicationContext      xxx      =
ClassPathXmlApplicationContext(pa.applicationcontext.xml);
Iclase1 claseUno = (Iclase1) xxx.getBean("nombreBean");
```

Donde xxx es la instanciación del contexto de la aplicación.

Ciclo de Vida

Dentro de la clase ApplicationContext existe un **ciclo de vida**, es el siguiente:

1.- **Crear el bean.** Se llama al constructor y se instancia el objeto.

a- **Si el constructor tiene argumentos** le tenemos que pasar esos argumentos. Para eso usamos la etiqueta <constructor-arg>, que lleva el atributo **name** para el nombre del atributo del constructor, o bien la propiedad **index** para indicar la posicion en la lista de argumentos del constructor, uno de los dos, no los dos, o también se le puede indicar por tipo usando el argumento **type**, no puede haber dos argumentos con el mismo tipo. Además pueden usarse los argumentos **value** o **ref** para un valor fijo o un objeto.

b- Tambien es posible que **el constructor no sea visible**, en ese caso se puede instanciar la clase llamando al método de instancia, para eso existe el atributo de la etiqueta <bean> llamado **factory-method**.

c- Otra opción es que **el constructor no tenga argumentos** en ese caso no hace falta indicar nada.

2.- Inyeccion de Dependencias/Propiedades

Se usan los metodos set..., son los métodos accesores del bean, que se usaran para darle valor a las propiedades.

3.- Establecer nombre Bean

BeanNameAware

Cuando tenemos mas de un ApplicationContext necesitamos mas datos acerca de ellos. Tiene que implementar la interfaz BeanNameAware. Con un único método setBeanName(). Si no la implementa no se ejecuta este paso.

4.- Establecer Nombre de la Fábrica

BeanFactoryAware

Cuando hay mas de un Applicationcontext necesitamos mas información sobre los BeanFactory. Tiene que implementar la interfaz BeanfactoryAware. Con un solo método setBeanfactory(). Si no la implementa no se ejecuta este paso.

4.5.- Establecer Applicationcontext

Solo si usamos el ApplicationContext tenemos este paso adicional. Debe implementar la interfaz **ApplicationContextAware** con un único método **setApplicationcontext()**. Este paso sirve para que el objeto reciba y conozca el Applicationcontext. Con ese método podemos conocer todos los beans.

Los pasos 5, 6 y 7 forman la parte del post procesado. Antes, durante y después. No tiene porque ser así de complicado, pero existe la posibilidad de hacer cosas antes, durante o después.

5.- Método de inicio

Init-Method

Tiene que implementar la interfaz BeanPostProcesor con un único método postProcessBeforeInitialization().

Una vez recibidos los valores puede interesarnos usar algún método adicional para terminar de construir el objeto. Solo lo usaremos en caso de que lo necesitemos.

Lo indicamos mediante la propiedad de la etiqueta <bean> **init-method**. A esa propiedad le decimos el nombre del método que necesitamos para inicializar el objeto.

A partir de aquí vienen los pasos del ciclo de vida del ApplicationContext.

Postproceso Antes -> BeanPostProcessor -> postProcessBeforeInitialization()

6.- Postproceso -> InitializingBean -> afterPropertiesSet()

Es útil si, por ejemplo, necesitamos que muchos beans de un mismo proceso se inicien de una misma forma. Debe implementar la interfaz InitializingBean que tiene un único método afterPropertiesSet().

7.- Postproceso después -> BeanPostProcessor -> postProcessAfterInitialization

Son los pasos que queremos hacer después de la inicialización, por ejemplo cerrar una conexión con una base de datos. Interfaz BeanPostProcessor con un único método postprocessAfterInitialization().

8.- **Puesto a disposición.** Cuando tenemos el objeto puesto a disposición. Debe implementar la interfaz **DisposableBean** que tiene un único método **destroy()**.

9.- **Finalización Bean.** A partir de aquí el bean vive lo que tenga que vivir dependiendo de si está en el request, o en la sesión, etc. Pero cuando el Bean ya va a ser eliminado podemos hacer más cosas si queremos y para eso tenemos el método destroy con el atributo **destroy-method** en la etiqueta <bean>.

```
ApplicationContext contexto = new  
ClassPathXmlApplicationContext("paquetes+nombre");  
Interfaz objeto = (Interfaz)Contexto.getBean("id");  
objecto
```

Propiedades Indexadas

A veces necesitaremos inyectar colecciones o arrays. Se llaman propiedades indexadas. Los valores que se le pasan son objetos. En un mapa la clave sera un string y el valor puede ser otra cadena o el ID de un objeto. Si es un properties tanto la clave como el valor serán textos. Dentro de la etiqueta <bean>, lo usamos así:

```
<bean>
    <property name="xxx">
        Para lista seria esta
        <list>
            <ref bean="vvvv" />
        </list>
        O bien para Sets
        <set>
            <ref bean="vvvv" />
        </set>
        O bien para mapas (clave/valor)
        <map>
            <entry key="clave" value-ref="valor u ID de un objeto" />
        </map>
        O bien cuando el objeto es un properties
        <props>
            <prop key="clave">valor de esta clave, es texto</prop>
        </props>
    </property>
</bean>
```

Cuando queremos inicializar una propiedad en null, Spring nos permite hacer esa inyección, así:

```
<property name="nombrePropiedad">
    null
</property>
```

Todo este proceso de las propiedades indexadas es un trabajo que hay que hacer a mano. Pero existe una propiedad que sabiendo manejarla bien nos reduce mucho el numero de líneas para definir los beans. Es una propiedad de la etiqueta <bean> y se llama **autowire**, significa autoenlazado, nos permite que, de forma automática, el propio ApplicationContext enlace unos bean con otros. Por defecto tiene valor de **none** pero tiene otros valores posibles que son:

ByName: Cuando en un bean hay una propiedad que se llame igual que en otro bean lo inyecta. Por lo tanto hay que tener mucho cuidado con los nombres que le damos a las propiedades del proyecto si vamos a usar este atributo.

ByType: Comprueba el tipo de la propiedad del bean y si coincide lo inyecta. Es útil si tenemos un objeto que siempre va a ser igual.

Constructor: Sirve para autoenlazar propiedades del constructor, si es que nuestro constructor tiene argumentos, y lo intentara satisfacer bien por nombre o bien por tipo.

Autodetect: Esta opción primero hace el trabajo del anterior (constructor) y si no hay constructor intenta hacer el trabajo del ByType.

Los dos últimos NO son muy recomendables ya que las posibilidades de meterlos en un charco importante son muchas.

```
<bean autowire="none" /> ByName /> ByType /> Constructor /> Autodetect">
```

El **autowire** se puede declarar bean a bean o **puede declararse de forma global** si lo usamos en la etiqueta <beans> llamándose la propiedad **default-auto-wire**:

```
<beans default-auto-wire="none" /> ByName /> ByType /> Constructor /> Autodetect">
```

Se pueden expresar **excepciones al auto enlazado usando la etiqueta <property>** dentro del bean, indicando así que el autowire se aplique a todas las propiedades menos a esas que indicamos con la etiqueta <property>.

Dentro de la etiqueta beans podemos indicar los métodos de inicio y de fin por defecto con los atributos **default-init-method** y **default-destroy-method** respectivamente.

Técnicas Avanzadas de Spring

Otra técnica para reducir el proceso laborioso de la creación de los beans en el applicationContext.xml es que, en el caso que usen una misma propiedad se puede definir herencia entre los bean, pero no hablamos de herencia tipo padre – hijo como la que se consigue en las clases usando la palabra *extends*, sino que las propiedades de un bean se pasen a todos los beans que nos interese. Cuando tengamos este caso podemos definir el bean digamos “padre” con el atributo **abstract=”true”**, no estamos diciendo que la clase sea abstracta, sino que ese objeto no dé propiedades hacia fuera. Las clases que queremos que hereden de este “padre” llevaran el atributo **parent** con el ID del que definimos como abstract:

```
<bean ID="xxxx" abstract="true">
...
</bean>
<bean ID="yyyy" parent="xxxx">
...
</bean>
```

Son clases independientes con propiedades comunes, solo eso. No significa que hereden unas de otras.

Otra característica avanzada que tiene el core de Spring es el API **JavaXBean**, que entre otras cosas tiene una clase que es el **propertyEditorSupport**, que me permite definir de forma personalizada los getters y setters de un bean. Esto tiene que ver con Spring. Con esta técnica podemos definir editores personalizados para propiedades de un bean.

Supongamos que tenemos una clase Teléfono, recibimos un texto y lo queremos transformar a la clase teléfono, Spring no sabe hacer eso. Usamos la posibilidad que tenemos en Spring de hacer un editor para transformar una propiedad determinada y por otra parte usamos esa clase de la API **JavaXBean**.

Lo haríamos definiendo nuestro bean y creando la clase bean con sus accesores. **Necesitaremos un intermediario** que hará la conversión de texto a objeto y viceversa. Esta clase intermediaria debe ser indicada a Spring en el archivo descriptor definiéndolo como un **<bean>** interno, y eso se define dentro de la etiqueta del bean como un **<property>** en el que el ID es optativo, debe llevar el class con la ruta completa y nombre, y será una clase que **extiende PropertyEditorSupport**. Esa clase PropertyEditorSupport tiene varios métodos, de los cuales sobreescribiremos los que necesitemos. Algunos de esos métodos son:

getAsString: getter que devuelve el objeto que nos interesa

setAtText: setter que trabajando con reflexión recibe un String y lo divide y le da los valores a decuados al objeto de nuestro interés

setValue: Establece el valor al objeto.

Declaracion en el descriptor:

```
<bean id="" class="" ....>
    <property name="tel" value="915554433">
        <bean class="ruta + .Editor" />
    </property>
</bean>
```

La clase seria:

```
public class xxxx {
    private Telefono tel;

    public Telefono getTelefono(){
        return tel;
    }
    public setTelefono(Telefono tel){
        this.tel = tel;
    }
}
```

La clase intermediaria seria:

```
public class Editor extends PropertyEditorSupport{

    private String prefijo;
    private String numero;

    public String getAsText(){
        Telefono tel = this.getValue();
        String salida = tel.getPrefijo() + "-" + tel.getNumero();
    }

    public void setAsText(String texto){
        Telefono tel = new Telefono();
        String[] valores = texto.split("-");
        tel.setPrefijo(valores[0]);
        tel.setnumero(valores[1]);
        this.setValue(tel);
    }
}
```

En resumen ApplicationContext lo que hace es crear objetos en el modulo CORE, y es la base a través de la cual los bean, usando interfaces, pueden usar o relacionarse con otras clases de la aplicación.

Por lo tanto, esto es muy importante y siempre hay que tenerlo en cuenta ya que es una de las bases de Spring: Como las clases se definen en el archivo descriptor applicationContext.xml, “**Todos los objetos definidos en el applicationContext.xml cuando sean llamados desde una vista, o desde donde sea, se han de llamar a través**

del ApplicationContext, no puede llamarse al constructor (new)".

Para llamar al contexto se pueden usar varias tácticas, por ejemplo, si es una aplicación J2EE debería estar en el Servletcontext (application scope). Si es una aplicación J2SE podemos implementar la interfaz (**ApplicationContextAware**) en la clase desde la que queramos acceder al applicationcontext. Así podemos acceder a los otros bean. Por ejemplo desde una fachada podemos acceder a un DAO, o desde una vista a una fachada, etc.

Modulo de Base de Datos

Hay que tener en cuenta que Spring no accede a la base de datos, Spring es solo una capa intermedia. Por lo tanto, si usamos Hibernate, el proyecto debe tener capacidades de Hibernate, lo mismo si usamos JPA, JDBC, etc.

Las ventajas que nos da utilizar Spring con el modelo, es que al ser un sistema abstracto, las cosas se usan de una forma muy parecida independientemente de la tecnología que usamos para trabajar con la base de datos.

Independientemente de eso, Spring usará DAOs siempre. Spring realiza una serie de trabajos el solo:

1.- Prepara Recursos

Dependiendo de lo que haya por detrás será una cosa u otra, por ejemplo, si usamos JDBC creará la conexión, etc. HibernateSessionFactory en caso de Hibernate, cierra la sesión el solo, etc.

2.- Inicia la Transaccion

Nosotros no necesitamos usar el commit.

3.- Retrollamada

Usando unos métodos llamados de **retrollamada** (van y vuelven) establecemos la lógica. Es decir ejecutan la transacción si la necesitan.

4.- Devuelve los datos obtenidos de la consulta

5.- Cierra los Recursos y maneja las Excepciones

Otra cosa que también hace Spring es manejar las excepciones, convirtiendo las excepciones de las tecnologías ORM que nosotros usemos a las excepciones de Spring.

Por lo tanto podemos decir, con matices, que podemos manejar Hibernate con Spring. **Desde nuestra fachada llamaremos a Spring en vez de a Hibernate.** Con MyEclipse se puede hacer la ingeniería inversa para crear los DAOs de Spring, más adelante está explicado cómo se hace.

Logicamente, hay diferencias dependiendo de la tecnología que usemos, no será lo mismo usar Hibernate, que JPA que JDBC. Para cada una de las tecnologías usa unas plantillas diferentes, con JDBC usa la plantilla `JDBCTemplate`, con JPA usa la `JPATemplate`, con hibernate usa dos `HibernateTemplate` y `Hibernate3Template`. Una es para las versiones anteriores a la 3 y la otra para la versión 3 o superior de hibernate. **Las templates (plantillas) son clases, no interfaces.**

El flujo va desde nuestra fachada a los DAOs de Spring, de ahí se llama al template, que a su vez debe conocer la conexión, que se la habremos dicho nosotros.

Las templates son clases que ya están hechas, nosotros lo que vamos a escribir son los DAOs, que serán bean definidos en el `applicationcontext.xml` y que se enlazan unos con otros.

Existe otro conjunto de clases que se llaman **DAOSupport** y cada tecnología tiene las suyas, por ejemplo, `JPADAOsupport`, `JDBCDAOsupport`, `HibernateDAOsupport`. La diferencia con el template es que el `DAOSupport` ya tiene su template y puede usarse directamente. Las clases `DAOSupport` adecuadas han de ser implementadas en nuestros DAOs, lo cual ya debe hacer MyEclipse.

AUTOMATICA

MANUAL

GESTION CONEXIÓN
CREAR RECURSOS

LIBERAR RECURSOS

CERRAR CONEXION

Hibernate con Spring

Para trabajar con **Hibernate desde Spring**, para empezar debemos darle al proyecto capacidades de Hibernate y de Spring. El fichero de configuración de Hibernate lo puedo definir o no, según me interese. La clase concreta que conecta Spring con hibernate puede leer ese fichero de configuración o directamente se lo puedo pasar como propiedades de ese bean.

En caso de que se lo queramos pasar como propiedades del bean se especifica así:

```
<bean id="sessionFactory"
      class="org.springframework.org.hibernate3.LocalSessionFactoryBean">
    <property name="configLocation"
              value="classPath:com/atrium/hibernate/xxxx.xml">
    </property>
</bean>

// Como vamos a usar los DAOs los declaramos asi
<bean id="xxx" class="xxx/yyy..." scope="prototype" lazy-init="true" />

// Al DAO no hay que decirle nada mas porque usamos las clases
DAOsupport

// Las fachadas también son beans que hay que declarar. Usamos
autowire
<bean id="yyy" class="xxx/yyy/..." scope="prototype"
autowire="ByType">
```

En el value del property configLocation, en caso de que el archivo no este en nuestro proyecto se usa **file:src/c:/....**, no clasPath

Ese código (los tres beans) es por cada tabla de la base de datos que vayamos a usar.

MyEclipse nos lo escribe pero hay que saberlo.

Para crear las clases que son los DAOs se puede usar la forma más cómoda que implica hacer esto:

```
public class xxxxx extends HibernateDAOSupport{

    public void Save(obj transito){
        getHibernateTemplate().metodoAccionSpring(.....);
    }

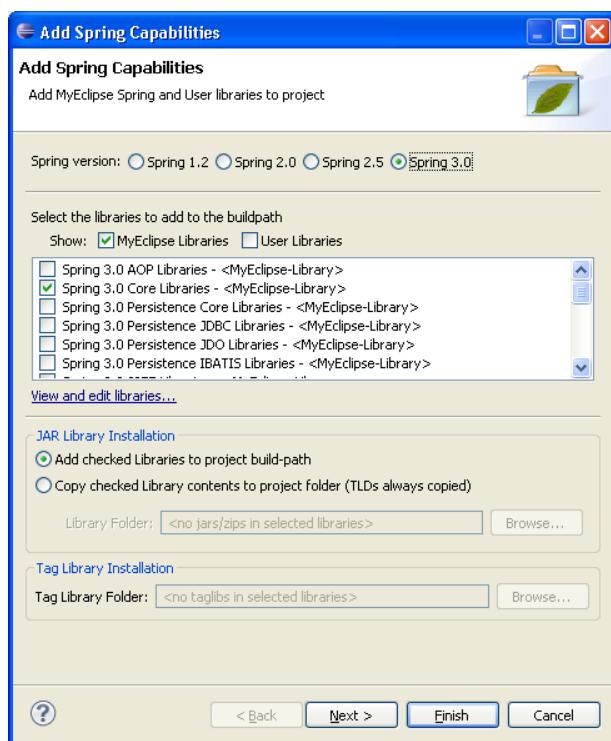
    // mas metodos...
```

}

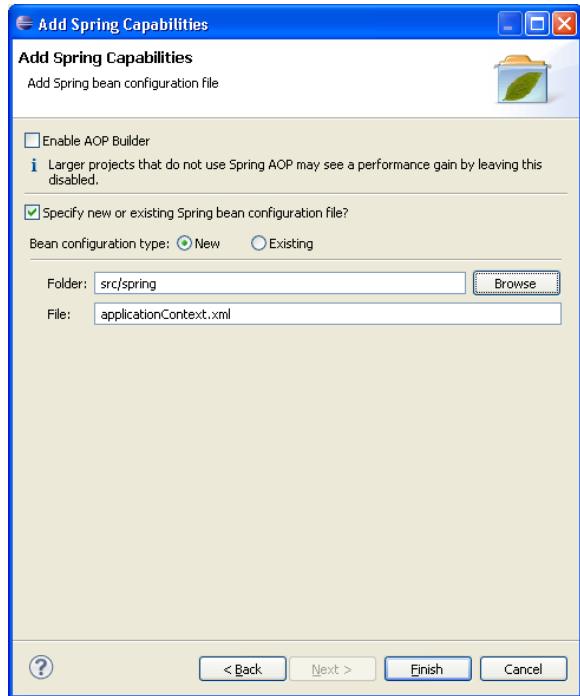
Extiende HibernateDAOsupport que ya tiene la plantilla por herencia, y usara los métodos que se necesiten para trabajar con la base de datos. **Los DAOs de Hibernate ya NO se usarán**, solo se usarán los de Spring.

Implementación de un proyecto Spring (solo) en MyEclipse

A partir de un proyecto (J2SE o J2EE) creamos un paquete “spring” y al proyecto le damos capacidades de spring



Elegimos la ultima opción para la versión, Spring 3, elegimos las librerías que son los modulos de Spring que vamos a usar en nuestro proyecto, pulsamos Next >



Aquí nos pregunta, enable AOP (aspect oriented programming) Builder, eso se lo quitamos porque de momento no lo vamos a usar en este caso (ya veremos mas adelante lo que es), sirve para enlazar con servicios. En Bean Configuration type podemos elegir entre usar uno nuevo o uno ya existente, si es un proyecto nuevo elegimos nuevo y elegimos el paquete creado para ello, si no es nuevo pulsamos en existing y buscamos el bean. Pulsamos Finish

Se crean los ficheros XML, y se importan las librerías necesarias.

Para hacer un simple hola mundo creamos una clase Saludo.java por ejemplo, en el archivo applicationContext.xml creamos la etiqueta bean de esa clase.

Además si es un proyecto J2SE creamos una clase con el método Main y ahí iniciamos el contexto y obtenemos la instancia del objeto.

```
ApplicationContext context = new  
ClassPathXmlApplicationContext("spring.applicationContext.xml");  
Saludo saludo = (Saludo)context.getBean("saludo");
```

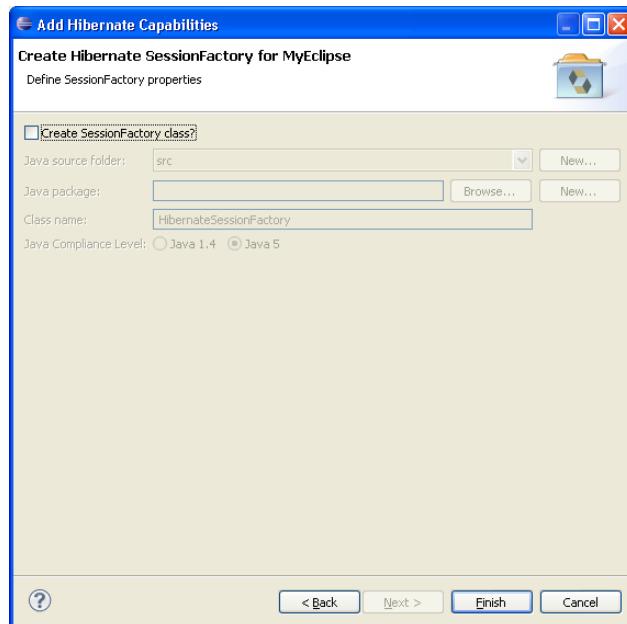
Haciéndolo **así existirá una dependencia fuerte** con la clase Saludo, así que lo haremos de otra manera, a través de Interfaces, con lo cual creamos una interfaz para la clase Saludo. Así:

```
ApplicationContext Context = new  
ClassPathXmlApplicationContext("spring.applicationContext.xml");  
Isaludo saludo = (Isaludo)Context.getBean("saludo");
```

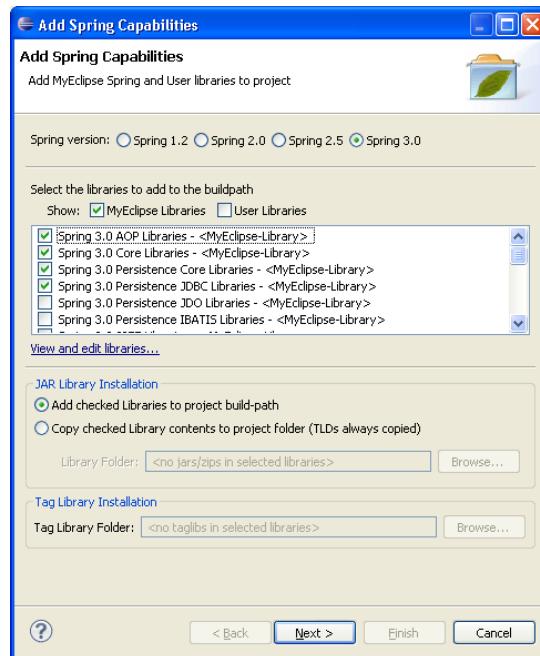
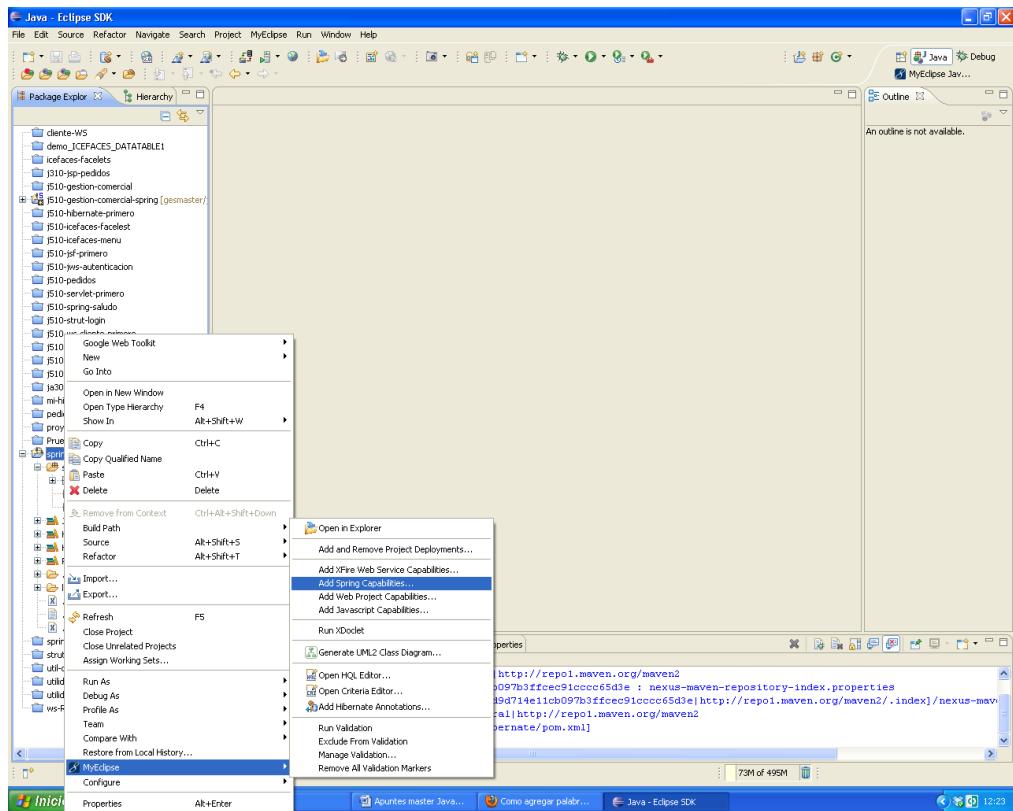
Creacion de Proyecto Hibernate con Spring en MyEclipse

Se crea un proyecto cualquiera, J2EE o J2SE.

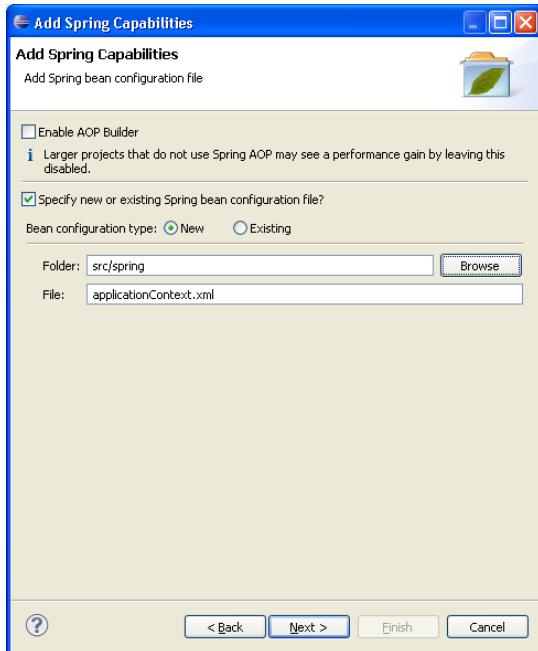
Se crean los paquetes, se le da capacidad de **Hibernate primero**, con anotaciones o no a Spring le da igual. Con la salvedad de que en el último paso, en el de las capacidades de Hibernate, donde nos pregunta si queremos **crear el SessionFactory class** le decimos **que NO**.



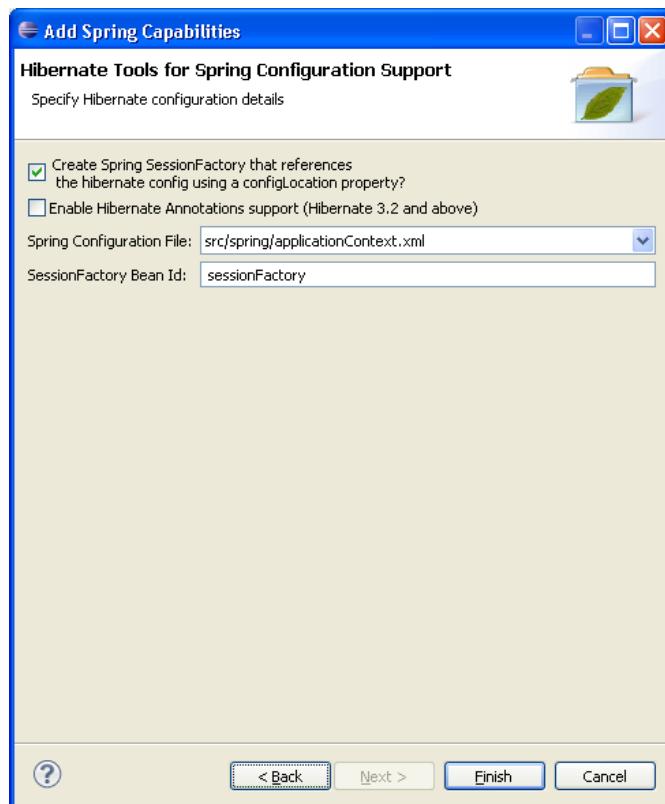
A continuación le damos capacidades de Spring.



Version de Spring 3.0. Marcamos las 4 primeros librerías. Lo demás como aparece en la imagen. Pulsamos Next >.



Esta ventana como aparece en la imagen, sin AOP Builder y elegimos proyecto nuevo indicando el paquete o si ya existe el bean usamos Existing y le indicamos donde está.
[Next >](#)



Aquí nos pregunta si queremos que nos cree el sessionFactory y le decimos que si

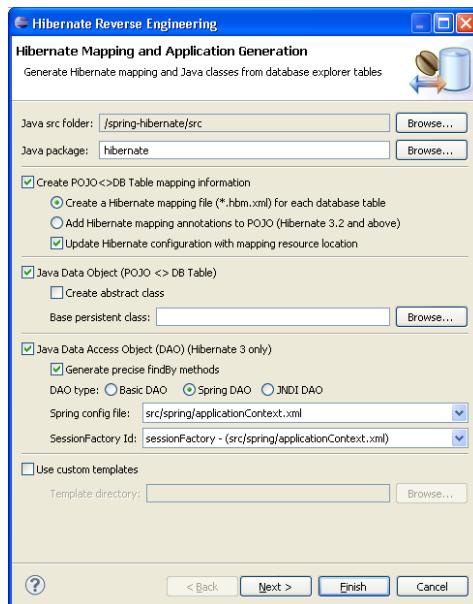
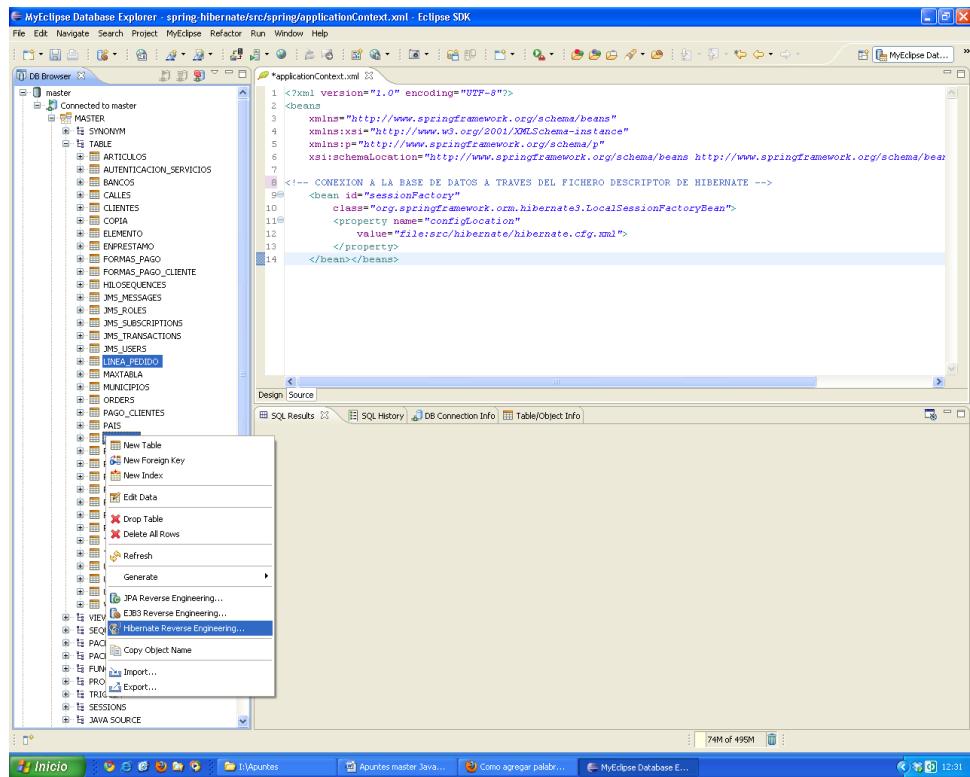
(recordemos que en el proceso de añadir las capacidades de Hibernate le dijimos que no. Creará el sessionFactory), y también nos pregunta si queremos anotaciones o no, además nos da una lista de descriptores en el combo (si es que hay mas de uno), que será útil cuando tengamos mas de uno para decirle en cual lo queremos. Pulsamos Finish.

Nos creara el applicationContext.xml,

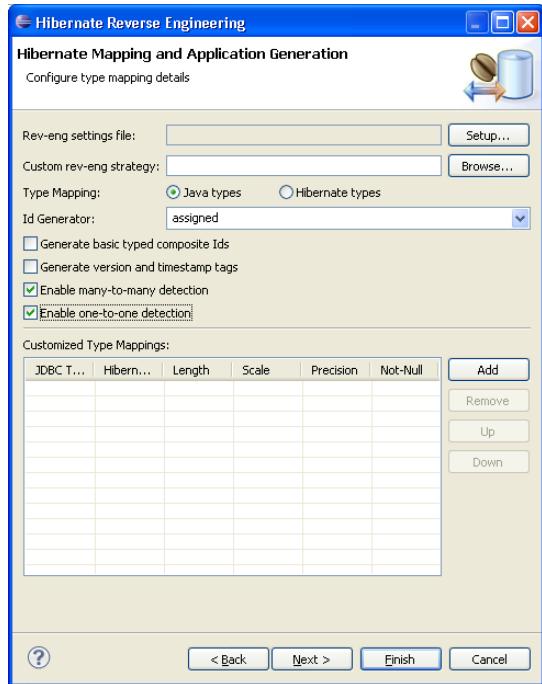
```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-
        3.0.xsd">

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation"
            value="file:src/hibernate/hibernate.cfg.xml">
        </property>
    </bean></beans>
```

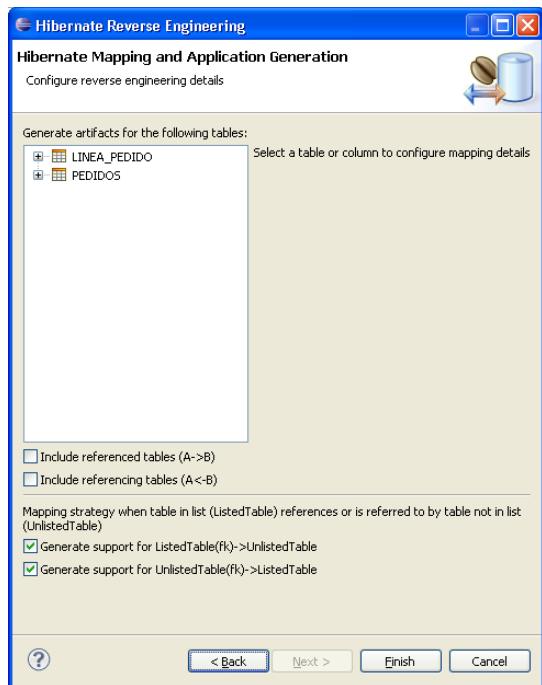
Nos vamos a hibernate database explorer y hacemos la ingeniería inversa eligiendo las tablas de nuestro interés.



Elegimos paquete y le damos las tres opciones, create POJO con las opciones por defecto, Java Data Object y Java Data Access Object (DAO), aquí podemos ver que ya reconoce a Spring y por supuesto lo marcamos. Pulsamos Next >



Como siempre, en Id Generator elegimos assigned y marcamos las dos ultimas casillas, Enable many-to-many detection y Enable one-to-one detection. [Next >](#)



Aquí podemos excluir algún campo de las tablas elegidas, dejamos las opciones por defecto y pulsamos Finish.

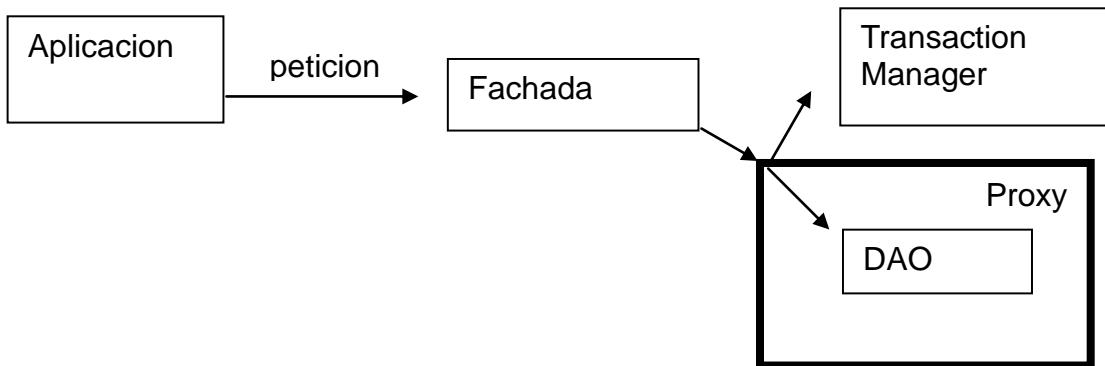
Se vuelve a cambiar el applicationContext.xml, al que despues le podemos dar los ajustes que nos interese. Añade los DAOs y las clases de persistencia.

Una cosa que no hacen los DAOs de Spring son las transacciones. Tenemos que hacerlo nosotros con programación orientada a aspectos.

También hay que adaptar los ficheros de configuracion de los DAOs, **si lo necesitamos**, por ejemplo Pedidos.hbm.xml, para cambiar las relaciones a propiedades individuales. Una tipo date puede cambiarse a String, etc.

Despues de esto se trabaja en la creación de las fachadas como hemos hecho cuando usábamos solo Hibernate pero ahora usamos los DAOs de Spring, lógicamente. Las fachadas tambien hay que declararlas en el fichero applicationcontext.xml. La creación de Fachadas implica la creación de sus interfaces. **La fachada no llevará constructor para crear el DAO ya que el DAO lo va a crear desde el applicationContext gracias al autowire, solo lo declaramos (`clienteDao cliDAO;` sin el new). Tampoco hace falta cerrar las conexiones.**

Transacciones Declarativas



Antes la fachada pedía al DAO, pero ahora; fachada pide y el Proxy, que engloba al DAO, decide si lo gestiona el DAO o se pasa al Transaction manager.

Entendamos por transacción una orden o mas que necesitan confirmación de la base de datos, independientemente de que sean updates, select, etc. Es decir todo lo que implique modificación en la base de datos.

Para hacer las transacciones, se usa el **TransactionManager**, que son las clases que manejan los commit dependiendo de la tecnología. No solo hacen el commit o el rollback, utiliza lo que se llama las **transacciones declarativas**. Esto consiste en decir como queremos que se comporte la transaccion pero no escribimos el código. El Transaction Manager trabaja con transacciones declarativas. De acuerdo a lo que nosotros indiquemos, es cómo se hará la transacción. Las transacciones declarativas pueden ser muy complejas dependiendo del caso.

La transacción debe ir unida al modelo (DAO). Para ello, en Spring se ha implementado **AOP**, Programacion Orientada a Aspectos, no tiene nada que ver con el aspecto grafico, sino más bien con el concepto, refiriéndonos a algo abstracto, conceptual.

Pongamos por caso un objeto modelo (DAO) que hace llamadas a la plantilla, la cual accede a la base de datos. Cuando vamos a transaccionar algo, el TransactionManager, es el encargado de manejar el modelo para realizar la transacción, es decir dar un alta, hacer una modificación, etc. En AOP, el objeto modelo, queda envuelto por un proxy, el cual es un objeto de comunicación, de tal forma que este proxy es el que va a capturar todas las llamadas que yo haga al objeto modelo. Cuando generamos el bean en el context, no generamos el bean del modelo sino del proxy. El proxy recibe las llamadas y las envía al modelo o al TransactionManager dependiendo de si hay que transaccionar o no.

Al actuar a través de un proxy tenemos el modelo totalmente desacoplado de la aplicación. El proxy será el mismo para cualquier tecnología, ya sea Hibernate, iBatis, JPA, etc.

Como hemos dicho, nosotros no escribimos el código solo hacemos la declaración de lo que queremos y declaramos el bean en el applicationContext.xml. La declaración de un bean para hibernate en el applicationContext.xml sería así:

```
// El TransactionManager es propio de la tecnología, en este caso
// Hibernate
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionMa
nager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

// El proxy es igual para todos independientemente de la
// tecnología usada
<bean id="pedido_tran"
      class="org.springframework.transaction.interceptor.TransactionPr
oxyFactoryBean">
    <property name="transactionManager" ref="transactionManager" />
    //El target puede ser el DAO o la Fachada. Nosotro lo hacemos
    en fachada
    <property name="target" ref="ges_pedidos" />
    <property name="transactionAttributes" //es una colección. Hay
    set, list y props
        <props> //Colección que requiere strings
            <prop key="alta_Pedido"> //key= nombre del método a
            transaccionar
                PROPAGATION-REQUIRED
            <prop>
        </props>
    </property>
</bean>
```

En el código Java de la vista de nuestra aplicación, desde donde llamamos a nuestra transacción, en este caso lo hacemos a través de una fachada, lo llamariamos así:

```
IGestionPedidos gesPed = (IGestionPedidos)
contexto.getBean("pedido_tran");
```

Antes hemos estado llamando a las fachadas, pero ahora, para realizar la transacción, llamamos al Proxy (`IGestionPedidos`) `contexto.getBean("pedido_tran");`

Tendremos que incorporar un Proxy (interceptor) por cada DAO. Puesto que cada DAO tiene que tener su propio DAO y definir la transacción con la necesitemos.

El ejemplo mostrado en estos códigos de arriba es el usado en el proyecto del master llamado j510-gestion-comercial-spring.

El bean del proxy lleva dos property, uno es el que se refiere a la clase del modelo y el otro al transactionmanager. Además lleva una tercera property que encierra las propiedades del método del modelo. Que lleva las características de la transacción. Le indica al factory bean que transacciones va a hacer y como las va a hacer. Por cada metodo llevara una etiqueta **<props>** con una etiqueta **prop** que a su vez lleva un atributo **key** que es el nombre del método y dentro del cuerpo del prop va el texto que define las **características** de esa transaccion.

Por método entendemos el método o los metodos que nosotros escribimos para dar solución a los posibles problemas que se puedan presentar por el uso de la aplicacion cuando se va a hacer una transacción.

Esto que se escribe dentro de las etiquetas prop es lo realmente importante, se escribirán separadas por comas:

Hay cinco **características para las transacciones**, se escriben en mayusculas:

1.- **Comportamiento de la transacción**. Es la única obligatoria y debe ser la primera. Las posibilidades son estas seis y hay que legir una de ellas:

PROPAGATION_MANDATORY -> Indica que el método tiene que ejecutarse dentro de una transacción obligatoriamente, si el método no se ejecuta dentro de una transacción genera un error y por lo tanto no se ejecuta. Esto quiere decir que la transaccion debe estar abierta. Ejemplo, una transacción de varios pedidos que actualiza mas de una tabla

PROPAGATION_NESTED -> Cuando se va a ejecutar ese método se tiene que abrir una transacción anidada distinta para ejecutar ese método. Si no hubiera una transaccion abierta se abriría una nueva transaccion independiente. Esa transacción independiente o anidada se puede confirmar o anular aparte de la transacción anterior. Ejemplo, si lanzamos un pedido con actualización de artículos y registro de operación, a continuación tenemos que actualizar una tabla de contabilidad. Esa ultima transacción la podemos poner como nested de forma que aunque falle la transacción de contabilidad, el pedido se contabiliza aunque sea mas tarde porque queda anotado, gracias a nuestro código y se hace mas tarde.

PROPAGATION_NEVER -> Este método no se puede realizar dentro de una transacción, lo excluimos, generaría una excepción en otro caso pero la transacción continuaria.

PROPAGATION_NOT_SUPPORTED -> Indica que ese método no se puede dejar dentro de una transaccion y además si se llamara dentro de una transacción suspendería la transacción.

PROPAGATION_REQUIRED -> Es el mas usado, indica que el método hay que transaccionarlo, si no hay transaccion la abre o crea una nueva.

PROPAGATION.Requires_New -> Este es una especialización del anterior, el método se ejecuta dentro de una transaccion si la transaccion no existe la abre, si la transacción existe, abre una nueva transacción y la ejecuta, la diferencia con el anterior es que si hay otra transaccion funcionando la para, ejecuta esta y luego la anterior continua.

2.- **Nivel de aislamiento.** Previene las lecturas fantasma, o fallidas, etc, debemos tener privilegios para hacer bloqueos en la tabla. Las posibilidades que tiene son (también siempre en mayúsculas):

ISOLATION_DEFAULT -> Cuando no tenemos privilegios para hacer bloqueos se usa la configuración por defecto de la base de datos.

ISOLATION_READ_UNCOMMITTED -> No previene ningun problema, se usa para desbloquear lo que estuviera previamente bloqueado.

ISOLATION_READ_COMMITED -> Previene el problema de lecturas Sucias

ISOLATION_REPEATABLE_COMMITED -> Previene las lecturas Sucias y Repetidas

ISOLATION_SERIALIZABLE -> Previene las lecturas Sucias, Repetidas y Fantasmas.

Si no se pone ninguna se aplica la primera (default).

3.- Solo Lectura

ReadOnly – En algunas bases de datos como Oracle, si la transaccion contiene una consulta y yo se lo especifico como ReadOnly es capaz de ejecutarlo mas rápido. Si la transacción no contiene ninguna consulta este valor no sirve de nada. Solo se puede poner con las transacciones PROPAGATION_REQUIRED, PROPAGATION_REQUIRED_NEW y con PROPAGATION_NESTED.

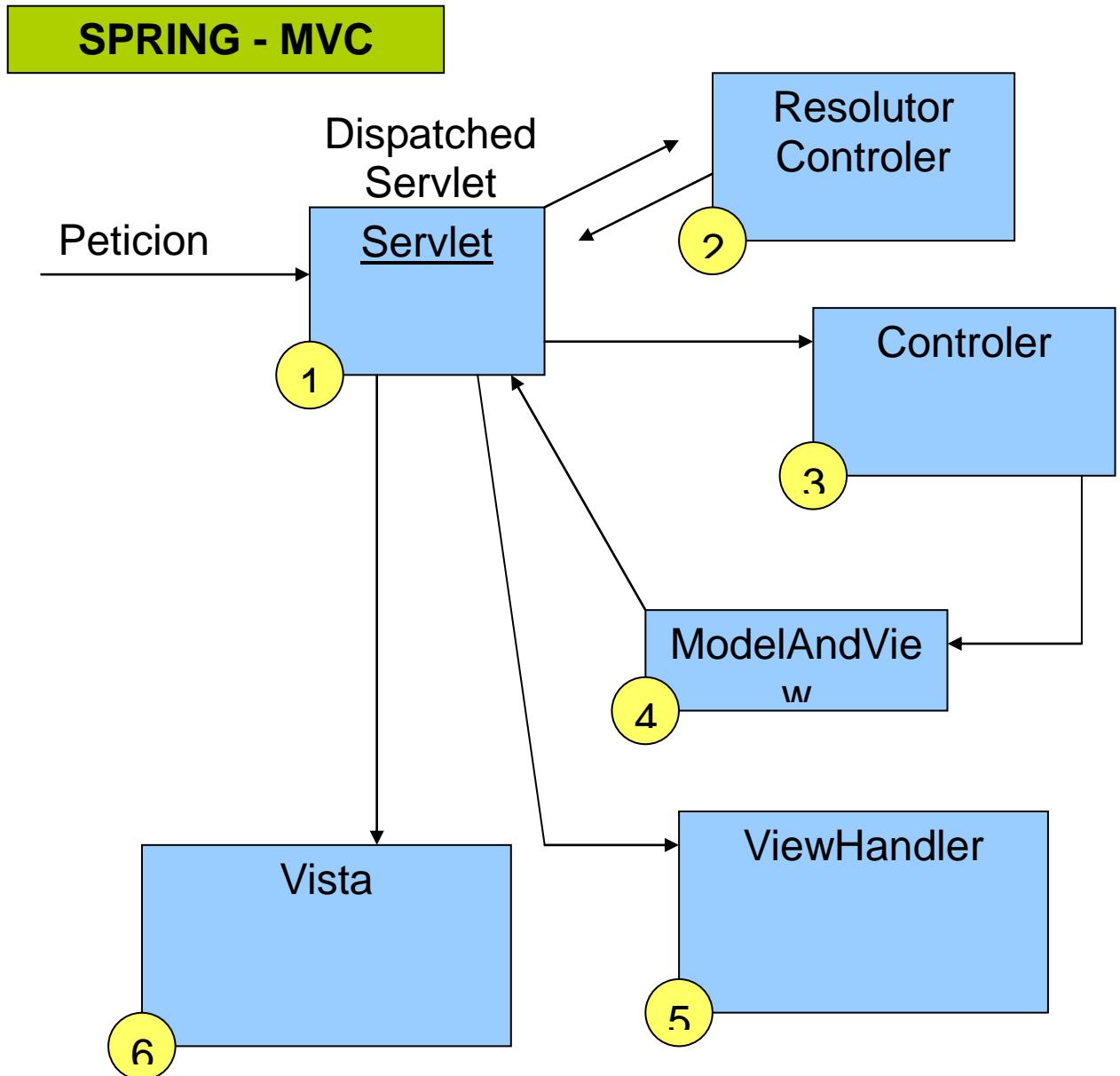
4.- **Intervalo.** Es la duración en segundos (es posible que sea en milisegundos) de la duración de la transacción, si la transacción no se satisface en ese tiempo se anula (rollback).

5.- **Reglas de Eliminacion.** Con esto indicamos cuando una transacción se elimina o no. Es decir con esto podemos contradecir las reglas descritas anteriormente, por ejemplo si a una transaccion le damos un intervalo x, aquí podemos hacer que no se anule si llega ese tiempo. Puede llevar un signo + (mas/anula) o - (menos/continua) delante para indicar cuando se aplica una de estas reglas o cuando no. Son mecanismos de excepciones.

Todas estas reglas van en cada etiqueta <props>, y va una etiqueta <props> por cada método.

PROPAGACION	AISLAMIENTO	SOLO LECTURA	INTERVALO	REGLA ELIMINACION
PROPAGATION_MANDATORY PROPAGATION_NESTED PROPAGATION_NEVER PROPAGATION_NOT_SUPPORTED PROPAGATION_REQUIRED PROPAGATIONQUIRES_NEW	ISOLATION_DEFAULT ISOLATION_READ_UNCOMMITTED ISOLATION_READ_COMMITTED ISOLATION_REPEATABLE_COMMITTED ISOLATION_SERIALIZABLE LECTURAS SUCIAS (actualización) LECTURAS NO REPETIDAS (lectura) LECTURAS FANTASMAS (Lectura actualización)	READ_ONLY	N SEGUNDOS	+ XCEPT -EXCEPT

Spring-MVC



El **dispatchedServlet** como todo servlet se define en el web.xml. Si no especifico como se llama el dispatcherServlet, este tendrá por defecto el nombre que le asignemos miControlador-servlet.xml y se guardará en el WEB-INF

Definicion en el fichero descriptor de Spring del resolutor

RESOLUTOR

```
<bean id="simpleResolutor"  
      class="org.springframework.web.servlet.handler.SimpleURLHandlerMapping">
```

```

<property name="mappings">
    <props>
        <prop key="/login.htm">idController</prop>
    </props>

```

CONTROLER

```
<bean id="idControler" classs="xxxxxx">
```

El resolutor ya está hecho y el controler hay que hacerlo, ya que será la clase/s de la lógica.

Cuando el controler devuelve el modelAndView, necesitamos el viewHandler

```

<bean id="viewHandler"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/jsp/" />
    <property name="sufix" value=".jsp//.iface//.jspx" />

```

Esta clase lo que hace es tomar del modelAndView, el view que es un string y colocarle el prefix y sufijo y devuelve la vista.

Siguiendo el formato de Spring 2.5, el dispatcher se obtiene por herencia. Básicamente se recomienda el uso de dos. AbstractController, que es el más simple, para tratar, por ejemplo, hipervínculo; y SimpleFormController, que es el que contendrá los métodos necesarios para validación, formateo, etc.

AbstractController

Tiene solo un método.

```

Public class miControlador extends AbstractController() {

    public ModelAndView handlerRequestInternal(HttpServletRequest,
                                              Response) {
        new ModelAndView("view", "model", obj);
    }
}

```

La primera cadena de texto es el view y es con el que trabajará el resolutor que en el siguiente paso devolverá la vista.. Los otros dos argumentos tienen que ver con el model, el segundo será la ruta del modelo y el tercero, el objeto del modelo.

SimpleFormController

Tiene un ciclo de vida mucho más complejo, pero a nosotros solo nos interesan 3

metodos que tendremos que sobreescribir, el metodo de conversion, validación y acción.

```
Public class miControlador extends SimpleFormController() {  
  
    public void initBinder(HttpServletRequest request, ServletRequestDataBinder binder) {  
  
    }  
    public void onBindAndValidate(HttpServletRequest request, Object object,  
        BindException bindException) {  
  
    }  
    public ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response, Object object,  
        BindException bindException) {  
  
    }  
}
```

ARGUMENTOS DE LOS METODOS

ServletRequestDataBinder: enlaza una propiedad con un conversor. El paquete java.Bean tiene clases para ello.

request: peticion

response: respuesta

object: es el command

bindException: registra los mensajes de excepcion

el onSubmit al final devolverá el modelAndView que se tendrá que mostrar.

En el constructor de la clase, que es el que primero se ejecuta, en el setCommandName le digo el nombre del objeto y en el setCommandClass le paso el class del objeto que luego resolverán por reflexión.

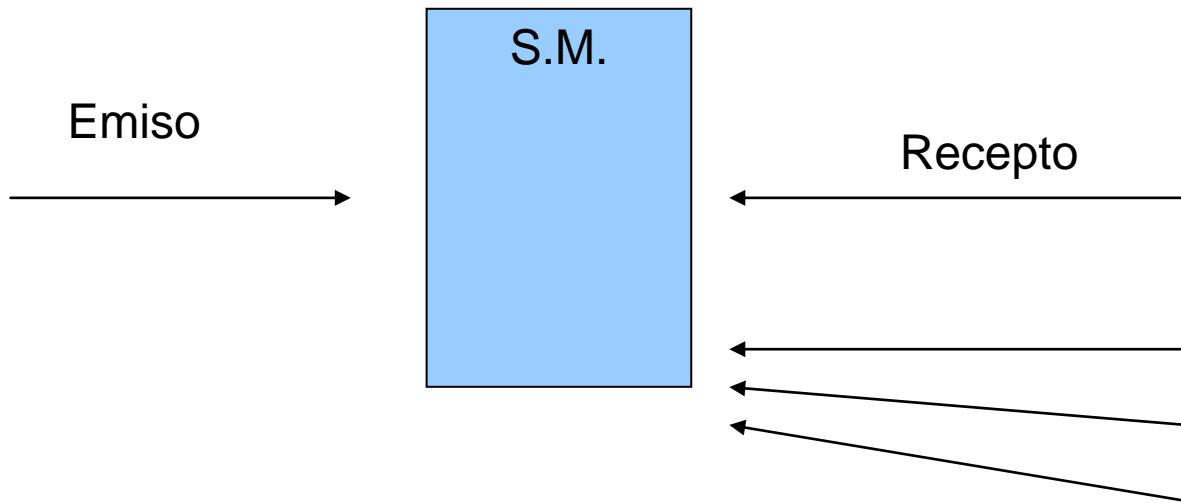
```
Public miControlador() {  
  
    setCommandName("");  
    setCommandClass(miControlador.class);  
}
```

VER PROYECTO J511-SPRING-MVC

En este proyecto utilizamos un springBean para ello definimos más espacios de nombres, el de aop y el de tx, que son los que tienen que ver con la transaccionalidad.

Mensajería JMS (Active MQ)

Utilizaremos JMS 1.1 que es la segunda versión. Cuidado que no es igual que la 1.0. Los servicios de mensajería están soportados por un servidor. Hay un emisor, que envía el mensaje al servicio de mensajería y un receptor que se lo solicita. El servicio de mensajería es asíncrono a diferencia del servicio web.

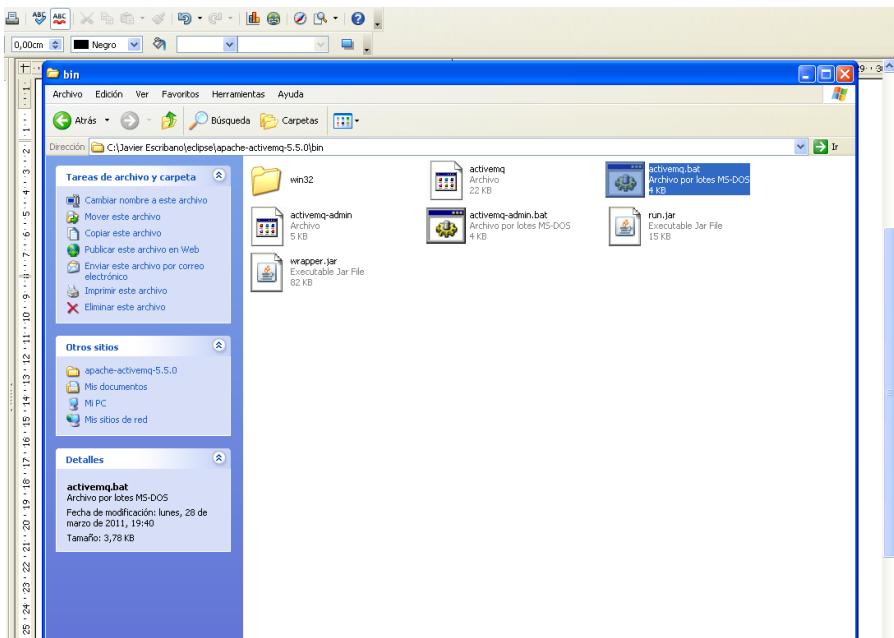


Las clases de un servicio de mensajería son interfaces, que implementará el servidor. Hay muchos servicios de mensajería. Nosotros vamos a usar uno libre, totalmente desarrollado en JAVA, aunque sirve para conectarse con casi cualquier servidor desarrollo en cualquier lenguaje. Además es un software muy utilizado en desarrollo. Su nombre es ActiveMQ.

Para conectar con el servidor se necesita un driver. Lo que haremos será una clase para la conexión, otra será el mensaje, otra para la emisión de mensajes, otra para parametrizar y que sea reutilizable y otra para la recepción.

Instalación del servidor

Copiamos el archivo de instalación, apache-activemq-5.5.0-bin.zip. Descomprimimos en la carpeta del eclipse. De los archivos que se descomprimen, el activemq-all-5.5.0.jar es el driver que necesitaremos para conectar la aplicación con el servidor. Para ponerlo en funcionamiento necesitamos hacer un doble click en la carpeta bin, en el archivo activemq.bat



Apuntes master Mensajería JMS.odt - OpenOffice.org Writer

C:\WINDOWS\system32\cmd.exe

```

INFO | For help or more information please see: http://activemq.apache.org/
INFO | Listening for connections at: tcp://A3-7JM:61616
INFO | Connector openwire Started
INFO | ActiveMQ JMS Message Broker <localhost, ID:A3-7JM-1290-1327653444891-0:1> started
INFO | jetty-7.1.6.v20100715
INFO | ActiveMQ WebConsole initialized.
INFO | Initializing Spring FrameworkServlet 'dispatcher'
INFO | ActiveMQ Console at http://0.0.0.0:8161/admin
INFO | Initializing Spring root WebApplicationContext
INFO | OSGi environment not detected.
INFO | Apache Camel 2.7.0 (CamelContext: camel) is starting
INFO | JMX enabled. Using ManagedManagementStrategy.
INFO | Found 5 packages with 16 @Converter classes to load
INFO | Loaded 152 type converters in 0.930 seconds
INFO | Connector vm://localhost Started
INFO | Route: routetl started and consuming from: Endpoint[activemq://example.A]
INFO | Total 1 routes, of which 1 is started.
INFO | Apache Camel 2.7.0 (CamelContext: camel) started in 2.031 seconds
INFO | Camel Console at http://0.0.0.0:8161/camel
INFO | ActiveMQ Web Demos at http://0.0.0.0:8161/demo
INFO | RESTful file access application at http://0.0.0.0:8161/fileserver
INFO | Started SelectChannelConnector@0.0.0.0:8161

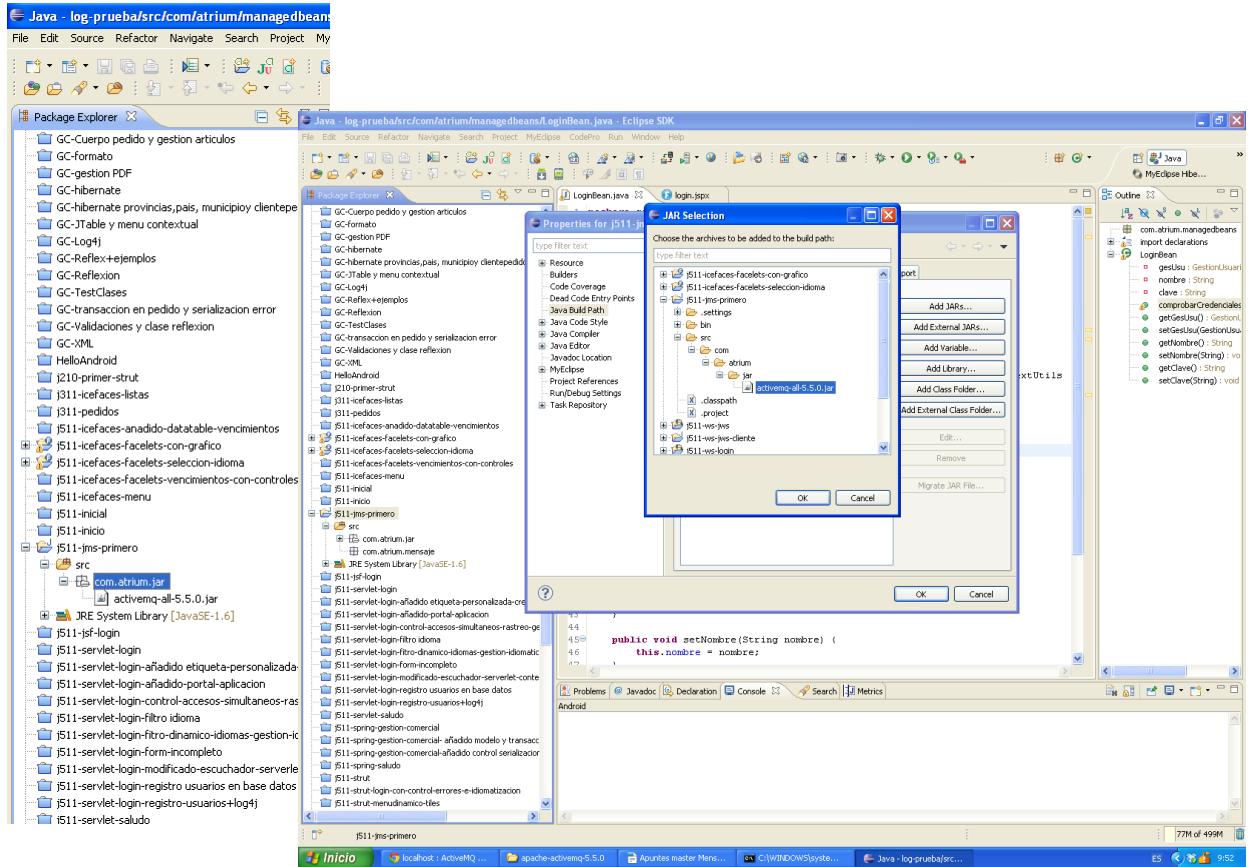
```

La administración del servicio de mensajería se lleva a cabo desde el navegador accediendo a **localhost:8161/admin**

The screenshot shows the Apache ActiveMQ Admin Console running on localhost:8161/admin/. The title bar displays the URL and a message in Spanish: "Esta página está escrita en inglés. ¿Quieres traducirla? Traducir No". The top right corner has a "Configuración" dropdown and a close button. The main header features the ActiveMQ logo (feather icon) and "The Apache Software Foundation" with the URL "http://www.apache.org/". Below the header is a navigation menu with links: Home, Queues, Topics, Subscribers, Connections, Network, Scheduled, and Send. To the right of the menu is a "Support" link. The main content area starts with a "Welcome!" message. It then displays broker information: Name (localhost), Version (5.5.0), ID (ID:A3-7JM-1290-1327653444891-0:1), and resource usage percentages (Store, Memory, Temp). On the right side, there are three sections: "Queue Views" (Graph, XML), "Topic Views" (XML), and "Useful Links" (Documentation, FAQ, Downloads, Forums). At the bottom left is a copyright notice: "Copyright 2005-2011 The Apache Software Foundation. ([printable version](#))". The bottom right credits "Graphic Design By Hiram".

Ejemplo práctico

Creamos un proyecto nuevo, creamos un paquete com.atrium.jar y dentro colocamos el driver y añadimos al buildPath



Creamos otro paquete com.atrium.jms para crear la clase Emisor_Mensajes.

```
public class Emisor_Mensajes {
    // VALORES POR DEFECTO PARA CONEXION CON ACTIVEMQ
    private String user = ActiveMQConnection.DEFAULT_USER;
    private String password = ActiveMQConnection.DEFAULT_PASSWORD;
    private String url = ActiveMQConnection.DEFAULT_BROKER_URL;
    // INDICARA SI EL TIPO DE GESTION DE ENVIO SERA POR PTP(QUEUE) O
    PUB/SUB
    // (TOPIC)
    private boolean topic = true;
    // INDICARA SI LA SESION ES TRANSACCIONAL O NO
    private boolean sesion_tran = false;
    // TIPO DE DESTINO DEL MENSAJE
    private Destination destino;
    // NOMBRE DEL TIPO DE DESTINO
    private String nombre_destino = "j311";
    // TIPO DE PERSISTENCIA DEL MENSAJE EN EL SERVIDOR
    private boolean tipo_persistencia = true;

    public static void main(String hh[]) {
        Emisor_Mensajes emi = new Emisor_Mensajes();
        emi.crear_Conexion();
    }

    public void crear_Conexion() {
```

```

        Connection connection = null;
        try {
            // CREAMOS LA CONEXION CON EL SERVIDOR. LAS
            CREDENCIALES SON LAS
            // DEFINIDAS POR DEFECTO
            ActiveMQConnectionFactory connectionFactory = new
                ActiveMQConnectionFactory(
                    user, password, url);
            connection = connectionFactory.createConnection();
            if (!topic) {
                connection.setClientID("12345");
            }
            connection.start();

            // CREAMOS LA SESION Y EL DESTINO
            Session session =
            connection.createSession(session_tran,
                Session.AUTO_ACKNOWLEDGE);
            if (topic) {
                destino = sesion.createTopic(nombre_destino);
            } else {
                destino = sesion.createQueue(nombre_destino);
            }
            // CREAMOS EL EMISOR DEL MENSAJE Y LA PERSISTENCIA
            DEL MENSAJE
            MessageProducer emisor_mensaje =
            sesion.createProducer(destino);
            if (tipo_persistencia) {

            emisor_mensaje.setDeliveryMode(DeliveryMode.PERSISTENT);
            } else {

            emisor_mensaje.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
            }
            // CREAMOS EL MENSAJE.
            TextMessage mensaje = sesion.createTextMessage();
            // CUERPO DEL MENSAJE
            mensaje.setText("PRUEBA INICIAL j311");
            // PROPIEDADES OPTATIVAS (APLICACION)
            mensaje.setStringProperty("propiedad_uno", "hola");
            mensaje.setIntProperty("propiedad_dos", 2);
            // PROPIEDADES DE LA CABECERA
            mensaje.setJMSPriority(4);

            // ENVIAMOS MENSAJE
            emisor_mensaje.send(mensaje);

            // LIBERAMOS RECURSOS CERRANDO LA CONEXION
            connection.close();
        } catch (Exception e) {
    }
}

```

La conexión se crea a través de la clase Conection de JMS, para ello requeriremos un

objeto ActiveMQConnectionFactory y con su método createConnection, ya tendremos el objeto conexión. A partir de esta conexión creamos la session, que lleva dos argumentos. El primero para indicar si es transaccional y el segundo para indicar si el mensaje saldrá automáticamente o si quedará bloqueado por el servidor.

Cuando tenemos la session abierta indicamos si ese mensaje se va a enviar a un buzón o a múltiples receptores createTopic() o createQueue() que recibirán como argumento el nombre del destino.

Creamos el MessageProducer, el emisor del mensaje, a partir de la session indicandole el destino. También le tendremos que indicar la persistencia

```
emisor_mensaje.setDeliveryMode(DeliveryMode.NON_PERSISTENT)
```

Lo que nos falta es el mensaje. La cabecera del mensaje tiene propiedades, además de las propiedades que tiene, también nosotros podemos crear nuevas propiedades. Por ejemplo, si queremos en gestión comercial incluir el servicio de que la condición de entrega es de inmediato y otro cliente una entrega en 24 horas. Nuestro sistema de mensajería creará las órdenes de pedido y en la cabecera le indicaré qué tipo de entrega conlleva cada pedido.

```
// CREAMOS EL MENSAJE.  
TextMessage mensaje = sesion.createTextMessage();  
// CUERPO DEL MENSAJE  
mensaje.setText("PRUEBA INICIAL j311");  
// PROPIEDADES OPTATIVAS (APLICACION)  
mensaje.setStringProperty("propiedad_uno", "hola");  
mensaje.setIntProperty("propiedad_dos", 2);  
// PROPIEDADES DE LA CABECERA  
mensaje.setJMSPriority(4);
```

Una vez creado el mensaje solo nos queda enviarlo y liberar los recursos

```
// ENVIAMOS MENSAJE  
emisor_mensaje.send(mensaje);  
  
// LIBERAMOS RECURSOS CERRANDO LA CONEXION  
connection.close();
```

Una vez tenemos el mensaje enviado necesitaremos un receptor, que será la clase Receptor_Mensajes. Esta necesita los mismos datos de conexión que el emisor del mensaje para conectar con el servidor.

```
public class Receptor_Mensajes implements ExceptionListener,  
MessageListener {  
  
    // VALORES POR DEFECTO PARA CONEXION CON ACTIVEMQ  
    private String user = ActiveMQConnection.DEFAULT_USER;  
    private String password = ActiveMQConnection.DEFAULT_PASSWORD;
```

```

private String url = ActiveMQConnection.DEFAULT_BROKER_URL;
// INDICARA SI EL TIPO DE GESTION DE ENVIO SERA POR PTP(QUEUE) O
PUB/SUB
// (TOPIC)
private boolean topic = true;
// INDICARA SI LA SESION ES TRANSACCIONAL O NO
private boolean sesion_tran = false;
// TIPO DE DESTINO DEL MENSAJE
private Destination destino;
// NOMBRE DEL TIPO DE DESTINO
private String nombre_destino = "j311";
// TIPO DE PERSISTENCIA DEL MENSAJE EN EL SERVIDOR
private boolean tipo_persistencia = true;
// PRODUCTOR DEL MENSAJE
private MessageProducer respuesta_alemisor;
// LECTOR DEL MENSAJE
private MessageConsumer lector_mensaje;
// NOMBRE DEL CONSUMIDOR DEL MENSAJE
private String nombre_consumidor = "Juan";
// IDENTIFICADOR DEL CLIENTE
private String clienteID = "12345";
// DECIMOS LA FORMA DE LEER EL MENSAJE
private boolean escuchador = false;

public static void main(String hh[]) {
    Receptor_Mensajes rec_mensaje = new Receptor_Mensajes();
    rec_mensaje.leer_Mensaje();
}

public void leer_Mensaje() {

    try {
        // CREAMOS LA CONEXION CON EL SERVIDOR. LAS CREDENCIALES
SON LAS
        // DEFINIDAS POR DEFECTO
        ActiveMQConnectionFactory connectionFactory = new
            ActiveMQConnectionFactory(
                user, password, url);
        Connection connection =
connectionFactory.createConnection();
        // ESTABLECEMOS EL IDENTIFICADOR DEL CLIENTE EN LA CONEXION
        connection.setClientID(clienteID);
        // ESTABLECEMOS COMO ESCUCHADOR DE EXCEPCIONES LA PROPIA
CLASE.
        connection.setExceptionListener(this);
        connection.start();

        // CREAMOS LA SESION Y EL DESTINO
        Session session =
connection.createSession(sesion_tran,
            Session.AUTO_ACKNOWLEDGE);
        if (topic) {
            destino = sesion.createTopic(nombre_destino);
        } else {
            destino = sesion.createQueue(nombre_destino);
        }
        // PREPARAMOS UNA POSIBLE RESPUESTA AL EMISOR DEL MENSAJE
}

```

```

        respuesta_alemisor = sesion.createProducer(null);

        respuesta_alemisor.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

        // ESTABLECEMOS EL TIPO DE LECTOR DE MENSAJES EN FUNCION DE
        QUE SEA
        // PERMANENTE O NO, Y DE QUE SEA A UN TOPIC O A UNA QUEUE
        if (tipo_persistencia && topic) {
            lector_mensaje =
sesion.createDurableSubscriber(
                                (Topic) destino,
nombre_consumidor);
        } else {
            lector_mensaje =
sesion.createConsumer(destino);
        }

        if (escuchador) {
        // ESTABLECEMOS EL ESCUCHADOR PARA RECIBIR EL MENSAJE
            lector_mensaje.setMessageListener(this);
        } else {
        // LEEMOS EL MENSAJE CON UNA ESPERA MAXIMA DE UN
SEGUNDO.

            Message mensaje = lector_mensaje.receive(1000);
        // ELIMINAMOS RECURSOS
            lector_mensaje.close();
connection.close();

            if (mensaje instanceof TextMessage) {
                TextMessage mensaje_texto = (TextMessage)
mensaje;
                String cuerpo_mensaje =
System.out.println("SOY EL CUERPO " +
cuerpo_mensaje);
            }
        }

        } catch (JMSEException e) {
            System.out.println("hola");
        }
    }

    /**
     * Tratamiento personalizado de las excepciones que se puedan
producir al
     * leer el mensaje
     */
@Override
public void onException(JMSEException arg0) {

}

@Override
public void onMessage(Message mensaje) {
    if (mensaje instanceof TextMessage) {
        TextMessage mensaje_texto = (TextMessage) mensaje;
        String cuerpo_mensaje = null;

```

```

        try {
            cuerpo_mensaje = mensaje_texto.getText();
        } catch (JMSEException e) {
            e.printStackTrace();
        }
        System.out.println("SOY EL CUERPO " +
cuerpo_mensaje);
    }
}

```

Un elemento que se añade a la conexión es el setClientID(clientID), que solo será necesario si tiene suscriptores para indicárselos. También se le puede añadir un escuchador de excepciones para resolver posibles incidencias.

```

// ESTABLECEMOS EL IDENTIFICADOR DEL CLIENTE EN LA CONEXION
connection.setClientID(clientID);
// ESTABLECEMOS COMO ESCUCHADOR DE EXCEPCIONES LA PROPIA
CLASE.
connection.setExceptionListener(this);
connection.start();

```

Al crear la session del mensaje se indicará si es transaccional y la constante AUTO_ACKNOWLEDGE para que el mensaje se autonenvíe y asegurarse de la recepción.

```

// CREAMOS LA SESION Y EL DESTINO
Session session =
connection.createSession(session_tran,
                        Session.AUTO_ACKNOWLEDGE);
if (topic) {
    destino = sesion.createTopic(nombre_destino);
} else {
    destino = sesion.createQueue(nombre_destino);
}

```

Se puede leer el mensaje de dos formas. Modo conectado (conexión abierta) y desconectado.

En el conectado el modelo es a partir de un escuchador. Cada vez que llega un mensaje se genera un evento y se trata en el método del escuchador.

```

if (escuchador) {
// ESTABLECEMOS EL ESCUCHADOR PARA RECIBIR EL MENSAJE
lector_mensaje.setMessageListener(this);

```

De la otra forma se abre la conexión y se mantiene durante un tiempo especificado y si no llega nada se cierra la conexión

```

} else {
// LEEMOS EL MENSAJE CON UNA ESPERA MAXIMA DE UN
SEGUNDO.
Message mensaje = lector_mensaje.receive(1000);

```

Por ultimo, si es el caso de modo conectado, pues cerramos la conexión y liberamos los recursos

```
// ELIMINAMOS RECURSOS  
lector_mensaje.close();  
connection.close();
```

Para optimizar la aplicación, las clases Emisor_Topic y Emisor_Queue que comparten gran parte del código, lo pierden para pasárselo a una clase abstracta y heredar de ella, de esta forma no se podrá instanciar un objeto de la clase abstracta y ambas clases compartirán esa parte del código.

EJB

En Java no solo tenemos servicios como servicios Web, sino que hay una extensa lista de servicios propios hechos en Java. Con lo cual, tanto el cliente como el servidor van a ser Java.

Todos los servicios Java van a requerir un **servidor de aplicaciones** que no es lo mismo que un servidor Web. Es decir no es Tomcat, no es apache. Por lo tanto los servicios no funcionan desde el punto de vista de servidor. Los servicios son implementación de Fabricante, es decir en Java están definidos como Interfaces, y luego cada fabricante hace su implementación. Por lo tanto un mismo servicio puede ser completamente diferente dependiendo del servidor de aplicaciones.

Los servicios no son aplicaciones, estan una capa por encima. Todo lo que sea compartido por dos o mas aplicaciones es candidato a ser implementado como servicio.

Los servidores de aplicaciones pueden ser gratis o de pago. Por ejemplo JBoss es gratis y puede ser bastante interesante. JBoss lleva un Tomcat embebido y puede atender peticiones web también. Hay que saber trabajar con el para poder usarlo y configurarlo, usa archivos descriptores XML para la configuración.

Los servidores comerciales de aplicaciones tambien suelen tener versiones de prueba que se pueden probar gratuitamente.

EJB (Enterprise Java Bean) es un servicio Java, es otra parte de J2EE, aparte de las aplicaciones Web. Nació con la primera especificación de J2EE, hacia el año 2001, y no cumplió las expectativas, en realidad se utiliza poco y cada vez menos. Para cualquier cosa que haga un EJB hay otras alternativas mas fáciles de usar y mas útiles.

En la versión EJB 3, hubo algunas modificaciones que mejoran los beans de entidad (entity beans), es decir los ORM como Hibernate, aunque Hibernate es mucho mejor que EJB. EJB 3 usa JPA (Java Persistence API) para la persistencia, pero JPA necesita un motor para la persistencia, es decir es como un SessionFactory por si mismo no hace nada, con lo cual a JPA habría que meterle Hibernate. Aun así existen aplicaciones fabricadas con EJB, además es un proyecto de Sun con lo cual hay que conocerlo.

El problema de trabajar con beans de entidad es que se van encolando y la aplicación cada vez se vuelve mas lenta, sobre todo por que esta pensado para sistemas de alta concurrencia.

Hay tres tipos de EJB, los SessionBean, los EntityBean y los MessageBeans. Cada uno hace cosas distintas, prestan servicios distintos y funcionan de forma distinta.

Un **SessionBean** nos sirve para mostrar o tener logica de proceso. La sesión a la que se refieren no tiene nada que ver con la sesión que ya conocemos. Dentro de los

sessionBean, en función de cómo funcionen pueden ser con estado (statefull) o sin estado (stateless).

Los **EntityBean** sirven para la persistencia de datos. Hace un trabajo parecido al que hace Hibernate. En función de cómo realicen esa persistencia pueden ser **BMP** (Bean Manage Process) donde el SQL los escribimos nosotros a mano ó **CMP** (Container Manage Process), donde el código SQL lo genera el servidor como hace Hibernate.

Los **MessageBean** sirven para trabajar con servicios Java JMS (Java Message Service). Un JMS es un servicio de mensajería, es decir desde un objeto genero y lanza un texto que recibirá otro objeto, eso es un servicio de mensajería. Este tipo de mensajes en un web Service son sincrónicos, es decir el cliente queda a la espera de la respuesta. En este caso de JMS se trata de un servicio asíncrono, el cliente no queda a la espera, cuando la respuesta está lista se lanza al cliente y además se asegura que el mensaje llega.

Con estos servicios JMS todo está hecho en Java. Entonces una clase Java emite un mensaje y se almacena en un servidor intermedio que es un servidor de mensajes, queda ahí almacenado a la espera de que el cliente que lo tiene que recibir se conecte y entonces se lo envía. Gracias a este servidor intermedio es como se consigue que sea asíncrono.

Los mensajes pueden recibirlas uno o muchos clientes, en este caso hablaríamos de publicar el mensaje.

Este servicio de mensajería puede configurarse dándole tiempos de duración a los mensajes y muchas otras cosas. Un servidor intermedio del tipo que hemos explicado gratuito es **QMD**, es de Apache, y JBoss incorpora uno. Es lo que se llama un Message Driving. El cliente hace una petición cada cierto tiempo para saber si hay algún mensaje para él. Es un servicio de correo tipo POP3 que es ideal para servicios internos.

Los EJB se basan en RMI-IIOP, esto significa que nosotros exponemos el servicio a partir de interfaces, con lo cual tenemos una metodología segura de acceso, pero también hace que su desarrollo sea complicado.

Un caso de uso puede ser el de una aplicación en una máquina virtual que quiere acceder a otra aplicación dentro de otra MV que está dentro de un servidor de aplicaciones. La aplicación a (cliente) pide acceso al servidor de aplicaciones usando JNDI para ello. El servidor le da acceso a la aplicación del servidor y envía una copia de la interfaz de ese objeto del servidor y a través de él, el objeto del cliente accede a los métodos de la aplicación del servidor a través de esa interfaz la cual tiene que hacer una llamada al objeto del servidor.

Creación de un SessionBean de EJB 2.2:

1.- Como RMI trabaja con interfaces el primer paso es definir las interfaces. Si el acceso es remoto necesitamos dos interfaces; **home** y **remoto**. Si el acceso es local necesitamos la interfaz **Local** y **LocalHome**. Con lo cual siempre necesitaremos, dos ó cuatro interfaces.

2.- Por encima de estas interfaces habrá **una clase de implementación** que implemente esas interfaces, y además todas las clases que necesitemos para implementar la lógica de nuestro servicio.

3.- Necesitamos también un **fichero descriptor XML** del EJB llamado **EJB-JAR.xml**

4.- Un **fichero propio de despliegue**. Esto quiere decir que cada servidor necesita un fichero distinto para poder desplegar los EJB. Distinto es distinto. Nosotros como vamos a usar JBoss usaremos el de JBoss.

```
public interface xxxx extends EJBHome {  
    public tipoInterfazRemota create() throws CreateException,  
    RemoteException;  
}
```

El interfaz hereda de otra interfaz EJBHome y define un único método llamado create que hace las veces de constructor. El contenido al método se lo da el servidor de aplicaciones.

La interfaz remota seria:

```
public interface yyyy extends EJBObject {  
  
    // aqui se definen todos los metodos que queramos que tenga el  
    // servicio, el bean de sesión lanzando todos ellos la excepción  
    // RemoteException como mínimo.  
}
```

La Clase de implementación será:

```
public class zzzz implements SessionBean{  
  
    ejbActivate  
    ejbPassivate  
    ejbRemove  
    ejbCreate  
    setSessionContext
```

}

Implementa la interfaz SessionBean para recibir el ciclo de vida y ha de implementar todos sus métodos ya que son los del ciclo de vida del bean de sesión. El ciclo de Vida funciona así:

Cuando un bean de sesión recibe una petición, el primer método que se llama es el setSessionContext, que solo se ejecutara una vez cuando se instancia el objeto, como siempre esto nos permite acceder a cualquier recurso externo al SessionBean que podamos necesitar.

A continuación se llama el EJBCreate, también se ejecuta una sola vez si el SessionBean es con estado, si es stateless no actuara así. Este EJBCreate debe coincidir con el EJB definido en la interfaz, de recibir lo mismo y devolver lo mismo que se ha indicado.

A continuación si trabajamos con un SessionBean con estado se ejecuta el EJBActivate y queda a la espera de que ese cliente quiera algo más. Puede quedar a la espera en memoria de forma activa, o de forma pasiva siendo serializado y guardado en el disco. Dependiendo de esto se ejecutara EJBActivate o EJBPassivate. Si es un SessionBean sin estado no se ejecutaría ni el EJBActivate ni el EJBPassivate.

Cuando se acabe la sesión se ejecuta el EJBRemove, que se ejecutara una sola vez antes de eliminar el SessionBean.

El código que ponemos en estos métodos será el que necesitemos, y si no necesitamos alguno de esos métodos lo dejaremos vacío.

Además de esos 5 métodos escribiremos otros métodos que puedan ser necesarios.

Después de crear esas clases e Interfaces procedemos a crear los ficheros descriptores. EJB-JAR.xml será el utilizado para describir todos los beans que necesitemos dentro de una etiqueta <enterprise-beans>. Por cada SessionBean va una etiqueta <session> con una etiqueta <ejb-name> con el nombre del bean, las interfaces <home> y <remote> y si hace falta <localhome> y <local>, mas una etiqueta <ejb-class> para indicar el paquete + nombre de la clase y además el <sesión-type> y el <transaction-type>, así:

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>xxxx</ejb-name>
      <home>paquete.NombreClaseHome</home>
      <remote> paquete.NombreClaseRemote</remote>
      <ejb-class>paquete.nombreClase</ejb-class>
      <localhome>paquete.NombreClase</localhome>
      <local> paquete.NombreClase</local>
```

```

<session-type>Stateless o Statefull</session-type>
<transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
</ejb-jar>

```

El despliegue necesita un fichero mas que es el del despliegue del servicio en el servidor que será diferente para cada tipo de servidor. El de JBoss se llama jboss.xml.

Tanto este fichero como el anterior deben estar en la carpeta META-INF.

Es así con la versión 4.x de JBoss:

```

<jboss>
  <enterprise-bean>
    <session>
      <ejb-name>se corresponde con el del archivo anterior
      (xxxx)<ejb-name>
        <jndi-name>es el nombre del servicio el que nosotros
        queramos</jndi-name>
      </session>
    </enterprise-bean>
  </jboss>

```

Hasta aquí la parte del servicio de EJB.

MyEclipse NO tiene un plugin para la versión 2 de EJB.

En este pasopodemos desplegar (deploy) el proyecto, y probarlo como un proyecto web cualquiera. Arrancando el servidor (JBoss) al arrancar el servidor podemos ver si se ha desplegado bien en los mensajes que obtenemos por consola.

Si todo va bien ya tenemos el servicio funcionando a la espera de que algún cliente se conecte a el.

Preparación del Cliente EJB:

El cliente puede ser cualquier tipo de proyecto (que sea Java claro está), ya sea web o J2SE, al igual que pasaba con los Web Services.

El **cliente** se maneja con **JNDI**. JNDI es otro API de Java muy extenso que da soporte a la búsqueda de archivos y directorios. Se pueden establecer permisos, crear rutas, etc. Por lo tanto solo los programas Java se pueden conectar a este servicio.

Hay que crearse un OBJETO properties para guardar los datos de la conexión para poder conectarse al servicio. Necesitamos la cadena de conexión incluyendo la dirección, puerto, protocolo de comunicación, password, etc. Por otro lado debe incluir también el driver que vamos a utilizar. El cliente debe tener el driver de comunicación. El driver cambia a cada versión del servidor o también de un servidor a otro. El driver de la versión **4.x de JBoss** se llama **jbossall-client.jar**, en la versión 5 no es el mismo, debemos tenerlo para **añadirlo al proyecto del cliente**, podemos encontrarlo en la carpeta “client” de la instalación de JBoss (JBoss/client). Por ejemplo, para crear un objeto properties (NO un archivo properties) con JBoss versión 4.x seria:

```
Properties nombreProperties = new Properties();
xxxx.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory") // Este es el driver,
el valor es el driver.
xxxx.put(Context.URL_PROVIDER, "jnp://maquina:puerto(1099)") // 
Esta es la cadena de conexión, el puerto por defecto es 1099.
```

En nuestro proyecto j510-ejb-primeroy hay una clase llamada conexión_EJB.java que tiene un buen ejemplo reutilizable de una conexión a un servicio EJB.

JNDI contiene una serie de constantes de clase que guardan esa información que necesitamos para crear el properties de la conexión. El JNDI normalmente va con el driver.

Una vez cargado el objeto properties querremos **establecer la conexión**. Para ello a partir de un objeto de la Clase **InitialContext** que necesitará el properties que hemos preparado:

```
InitialContext contexto = new InitialContext("nombreProperties");
```

Cuando se hace este new se prueba la conexión y si no puede lanzará una excepción, con lo cual debe ir en un try/catch.

El objeto de la clase InitialContext tiene un método para conectar con el servicio, el método es **lookUp** al que hay que pasarle el nombre del servicio, que es el que le dimos en el archivo descriptor dentro de la etiqueta <ejb-name> del session o entity, etc., que

devuelve un Object

```
Object servicio = contexto.lookup("nombreServicio"); // el  
nombreServicio es el que le dimos en la etiqueta <jndi-name> del  
fichero jboss.xml
```

Aquí es donde termina JDNI y comienza RMI.

El cliente debe conocer las interfaces del servicio, para poder funcionar. Es decir debe tener una copia de ellas y deben estar en un paquete con el mismo nombre que en el servicio.

Con RMI necesitamos una copia de la interfaz remota como ya dijimos antes. Eso RMI lo hace usando la clase **PortableRemoteObject** que usa un metodo estatico llamado **narrow**, que usa dos paramtros, el primeor es el objeto Servicio que hemos obtenido a través del método lookUp y el segundo es lo que le voy a pedir al servicio que es el objeto que incorpora la interfaz Home.Class si todo esta bien hecho obtendremos una copia de la interfaz remota con la que trabajaremos en local. Después, para arrancar el servicio usamos create para arrancarlo.

```
Tiporemote yyyy = (Tiporemote)  
PortableremoteObject.narrow(servicio, interface Home.Class);  
zzz = yyyy.create();
```

Para acceder a sus métodos usamos el objeto del tipo **remote**.

```
zzz.nombremetodo();
```

En el proyecto llamado j510-ejb-primeroy hay una clase llamada Gestion_Servicios.java donde tenemos un buen ejemplo de esta parte de JNDI y RMI-IIOP que puede ser reutilizada solo haciendo los cambios necesarios.

Como este proceso implica un cliente y un servidor, tendremos disponibles dos consolas una para el cliente y otra para el servidor. A través de esas consolas vemos si todo esta funcionando bien cuando probemos nuestras aplicaciones. Para acceder a uno u otra consola hay unicono enla zona de abajo a la derecha de eclipse que es el icono de las consolas, parace un pequeño TV o monitor.

Estos ejemplos vistos aqui tratan el mas simple caso que se puede dar con los SessionBean que son los EJBs mas sencillos.

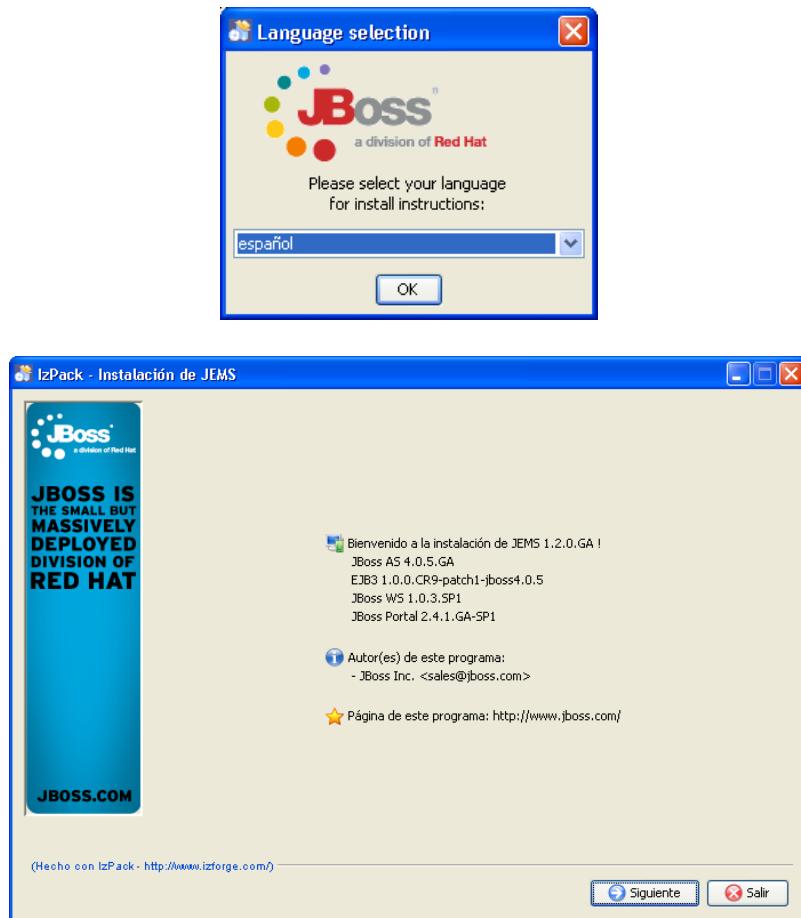
Este tipo de servicio de EJB se puede hacer mucho mas fácil con web service, los Entitybean se pueden hacer de una forma mucho mas sencilla a través de Hibernate. MessageBean puede sustituirse por el uso de JMS sin usar EJB con lo cual los tres EJBs pueden estar quedando obsoletos...

Instalacion de JBoss sobre Windows XP

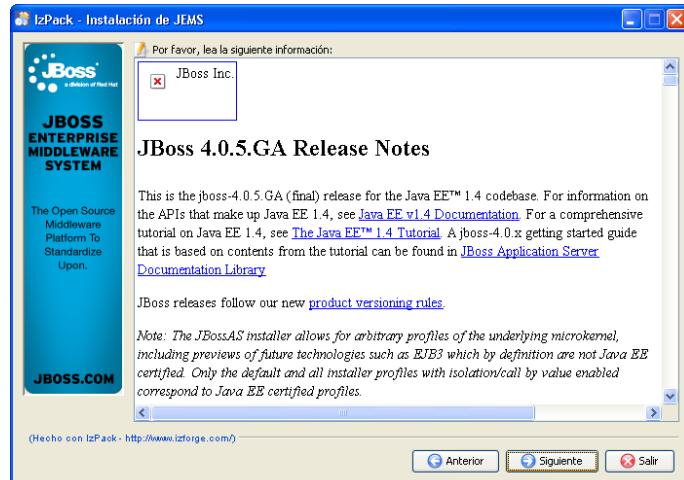
Debemos tener el servidor JBoss descargado, nosotros lo tenemos en una carpeta llamada jboss, en esa carpeta hay dos versiones que son la 4.0.5 y las 5.0.1 (requiere JDK 6). Esas dos están en archivos zip que bastaría con descomprimir donde queramos para tener el servidor, además hay un archivo llamado **jems-installer-1.2.0.GA.jar** que es un **instalador para Windows**. Tambien hay dos manuales en ingles sobre JBoss, uno trata la instalacion y el otro son los básicos.

La instalación la haremos usando el instalador así:

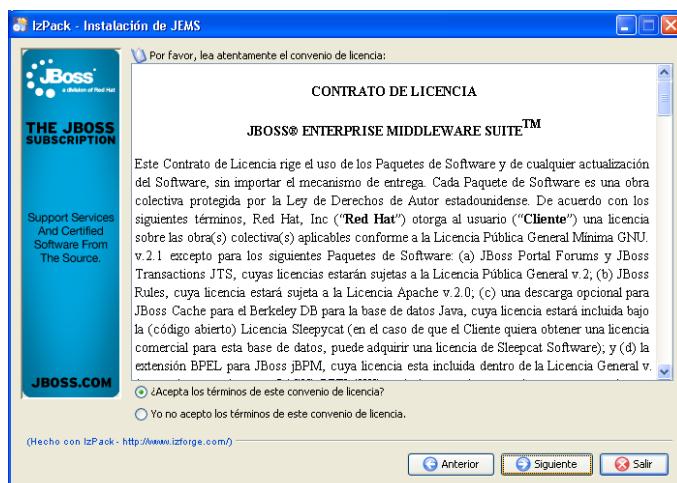
Al darle doble click sale la pantalla del idioma del instalador



Siguiente



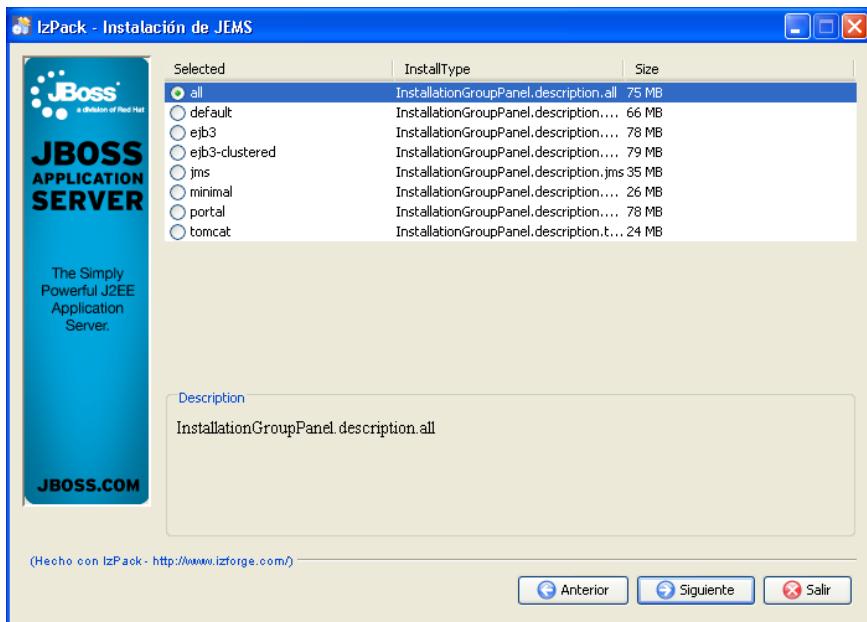
Siguiente



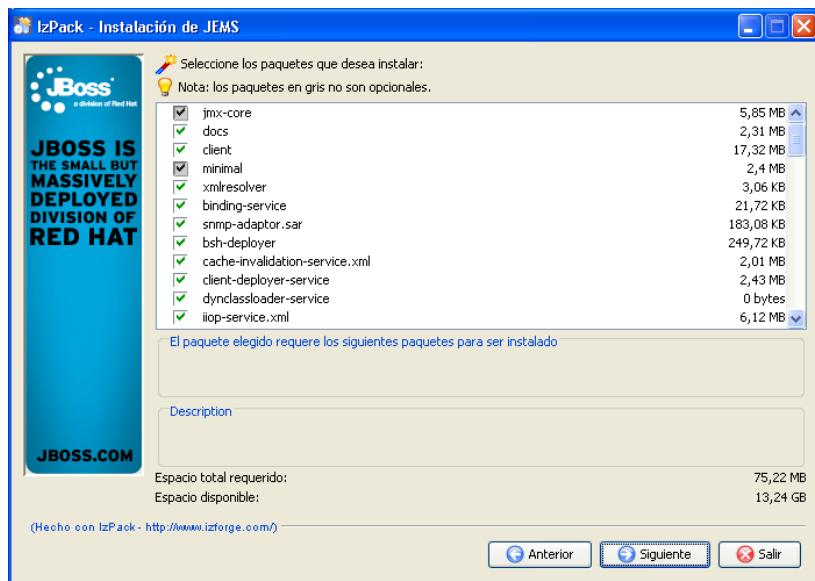
Aceptamos la licencia



Podemos elegir la ruta de instalación que queramos, y pulsamos Siguiente

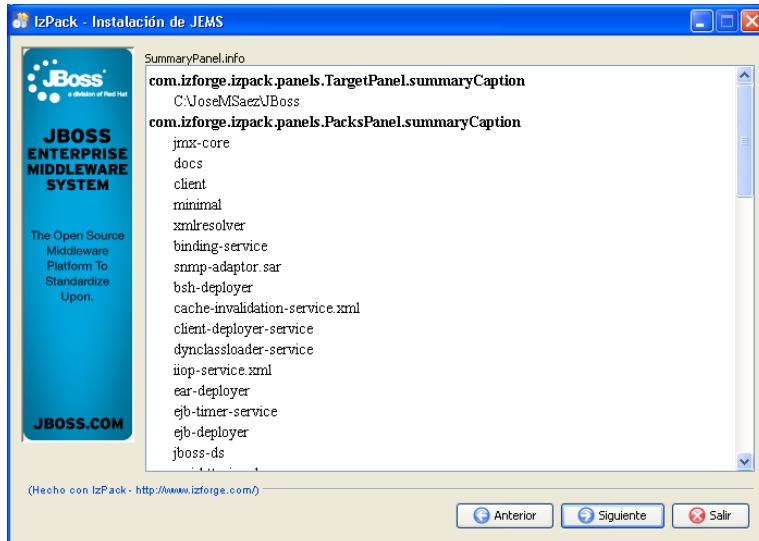


Nos pide que le digamos los modulos que queremos instalar, podemos elegir desde solo un tomcat donde se comportaría solo como un servidor hasta todo (all) donde va a instalarlo todo, eso es lo que elegiremos nosotros. Siguiente

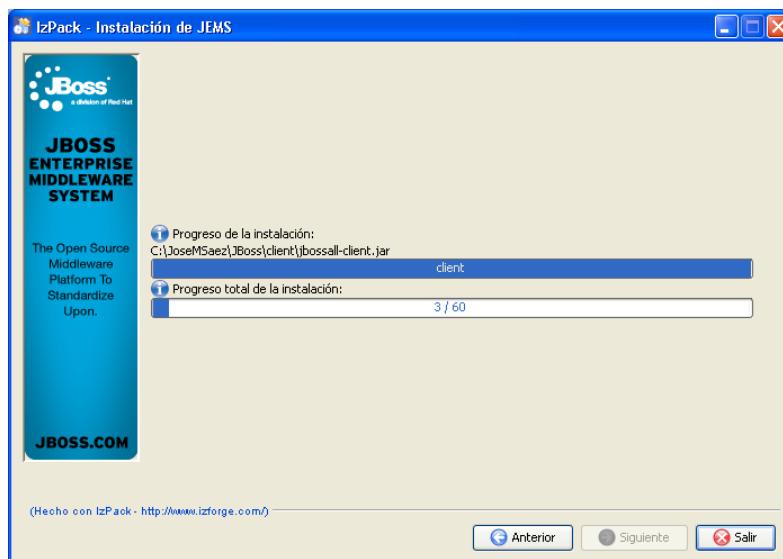


Siguiente

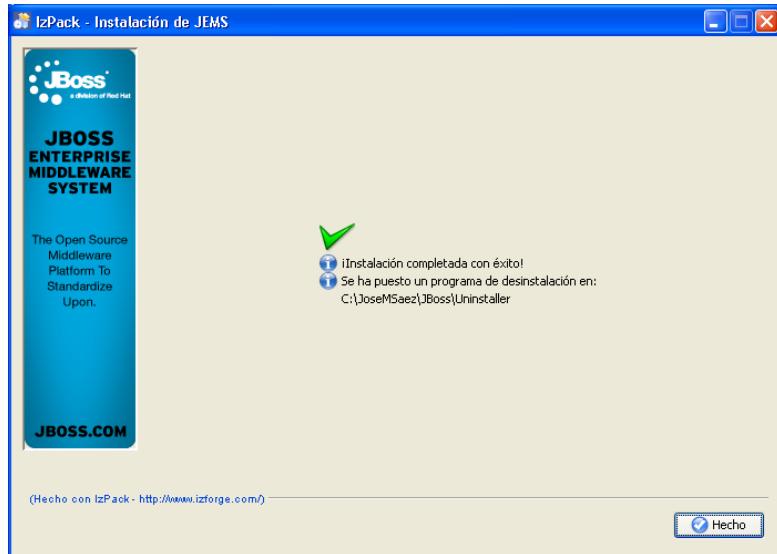
Pregunta que tipo de instalación queremos: Standard no dejará retocar nada si elegimos avanzada podremos modificar los archivos descriptores de la instalación, elegiremos estándar ya que en este caso no vamos a necesitar tocar nada



Siguiente

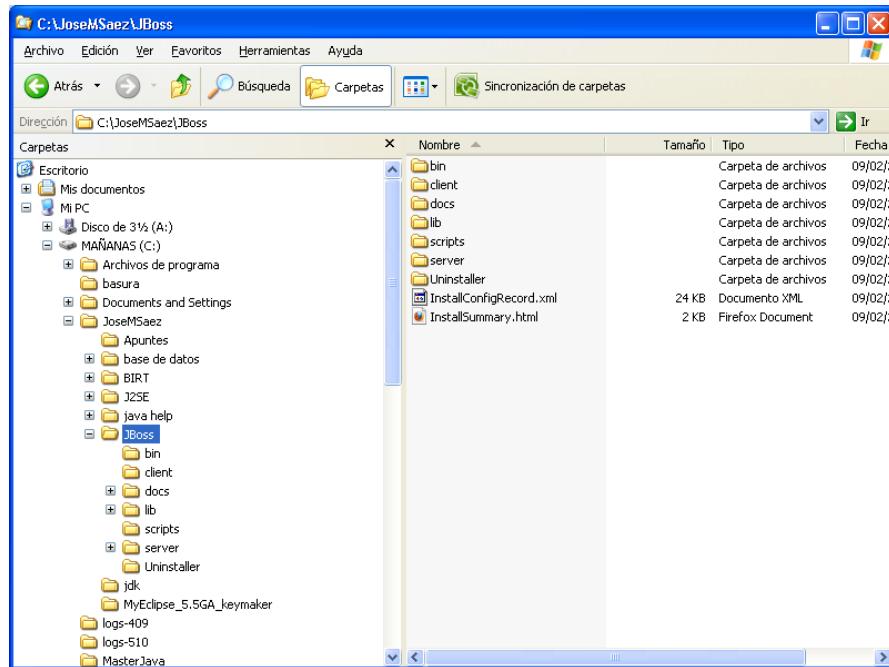


Comienza el proceso de instalación. Cuando termina Siguiente

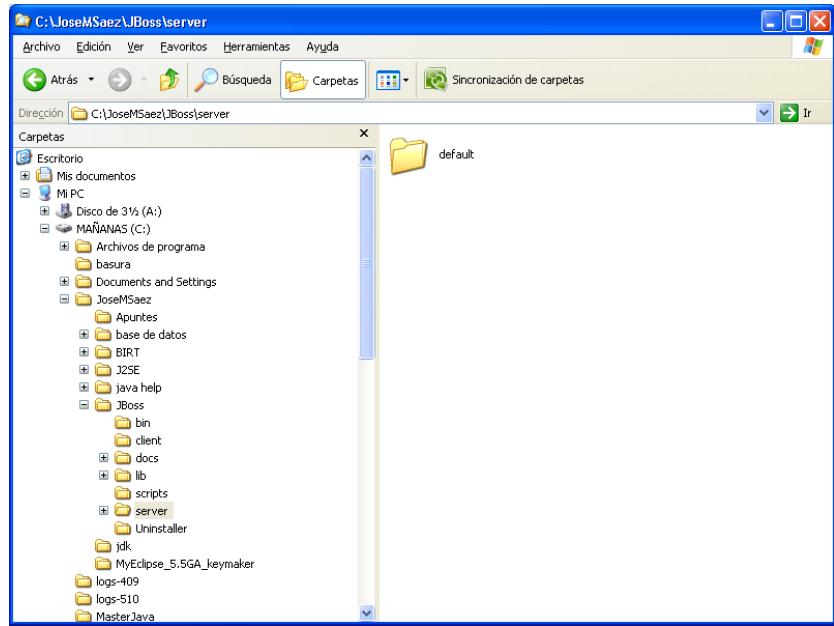


Hecho

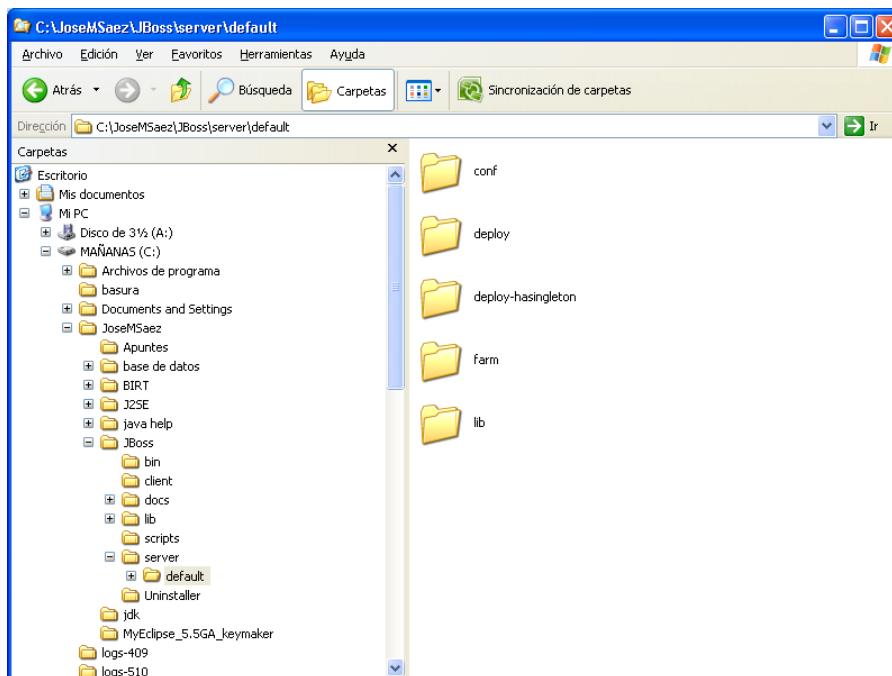
Nos habrá creado esta estructura de directorios:

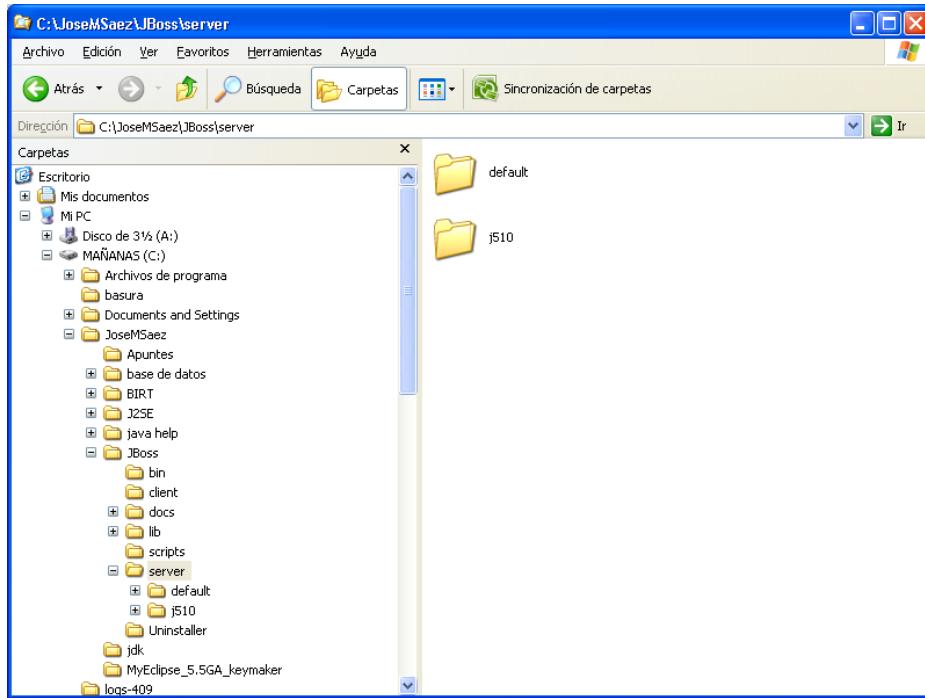


Los servidores que nosotros vayamos a tener se pondrán en la carpeta “server”, como habíamos elegido la instalación estándar en esa carpeta habrá una carpeta default:

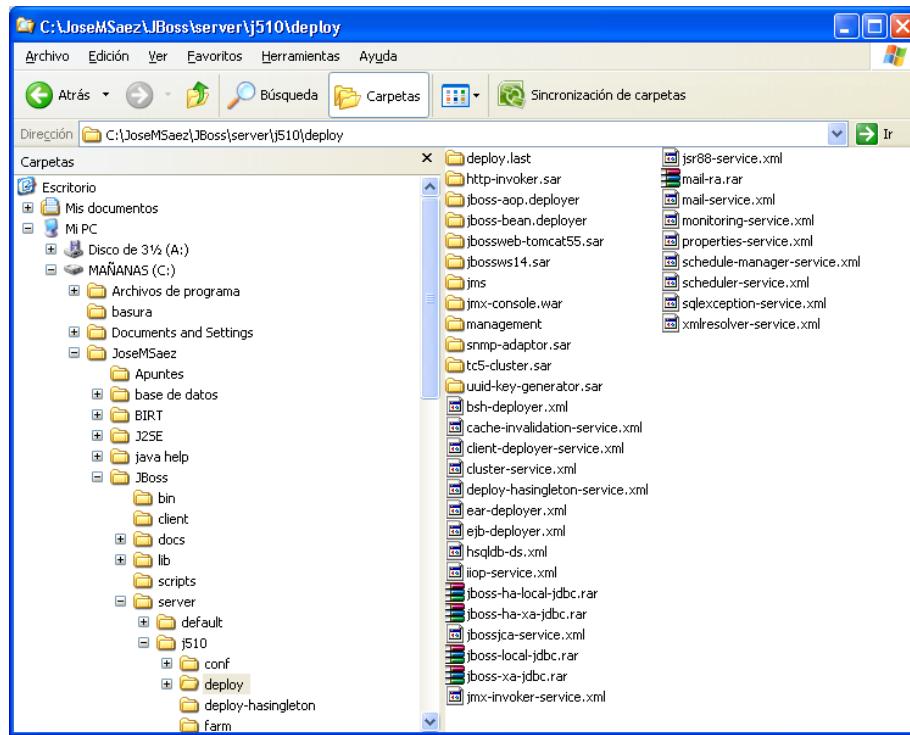


Dentro de la carpeta default esta todo lo que se necesita para que el servidor funcione, con lo cual si quisiéramos un segundo servidor bastaría con copiar esas carpetas en otra carpeta a la misma altura que default y ya lo tendríamos:

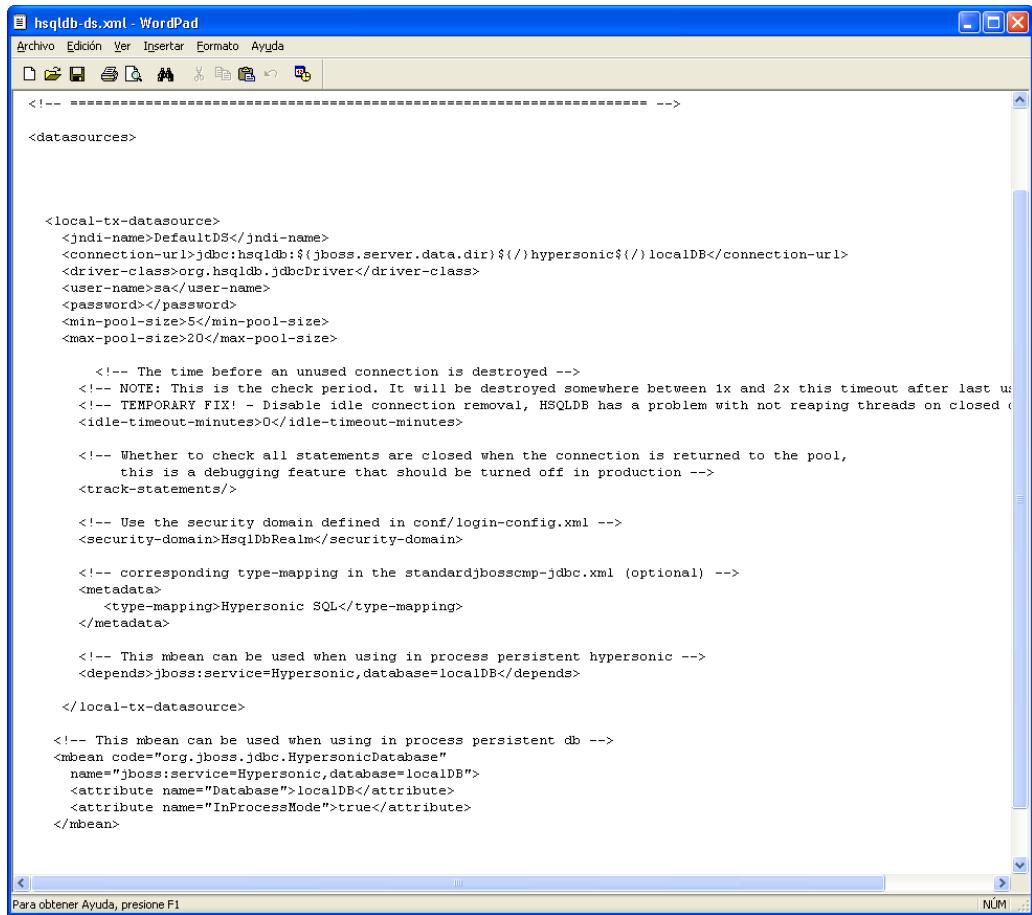




Dentro de la carpeta deploy del servidor hay un montón de cosas, entre otras hay ficheros XML que ajustan y modifican la funcionalidad de un aspecto del servidor JBoss



En el archivo **hsqldb-ds.xml** es donde se puede configurar la conexión a la base de datos.



```
<!-- ----- -->
<datasources>

<local-tx-datasource>
<jndi-name>DefaultDS</jndi-name>
<connection-url>jdbc:hsqldb:${jboss.server.data.dir}${/}hypersonic${/}localDB</connection-url>
<driver-class>org.hsqldb.jdbcDriver</driver-class>
<user-name>sac</user-name>
<password></password>
<min-pool-size>5</min-pool-size>
<max-pool-size>20</max-pool-size>

    <!-- The time before an unused connection is destroyed -->
    <!-- NOTE: This is the check period. It will be destroyed somewhere between 1x and 2x this timeout after last use -->
    <!-- TEMPORARY FIX! - Disable idle connection removal, HSQLDB has a problem with not reaping threads on closed connections -->
    <idle-timeout-minutes>0</idle-timeout-minutes>

    <!-- Whether to check all statements are closed when the connection is returned to the pool,
        this is a debugging feature that should be turned off in production -->
    <track-statements/>

    <!-- Use the security domain defined in conf/login-config.xml -->
    <security-domain>HsqlDbRealm</security-domain>

    <!-- corresponding type-mapping in the standardjbossescmp-jdbc.xml (optional) -->
    <metadata>
        <type-mapping>Hypersonic SQL</type-mapping>
    </metadata>

    <!-- This mbean can be used when using in process persistent hypersonic -->
    <depends>jboss:service=Hypersonic, database=localDB</depends>

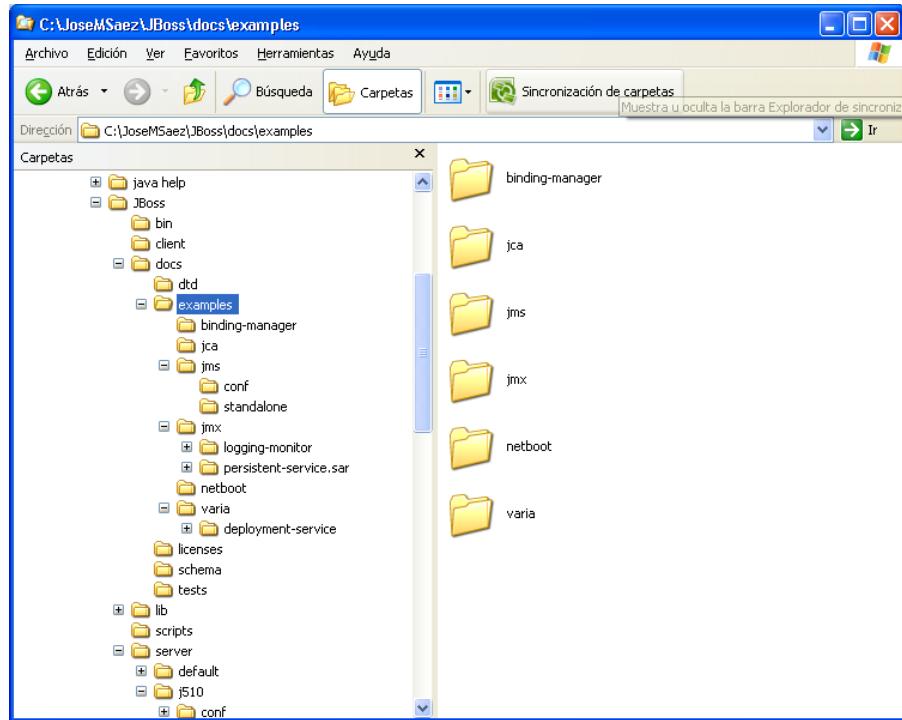
</local-tx-datasource>

<!-- This mbean can be used when using in process persistent db -->
<mbean code="org.jboss.jdbc.HypersonicDatabase"
      name="jboss:service=Hypersonic, database=localDB">
    <attribute name="Database">localDB</attribute>
    <attribute name="InProcessMode">true</attribute>
</mbean>

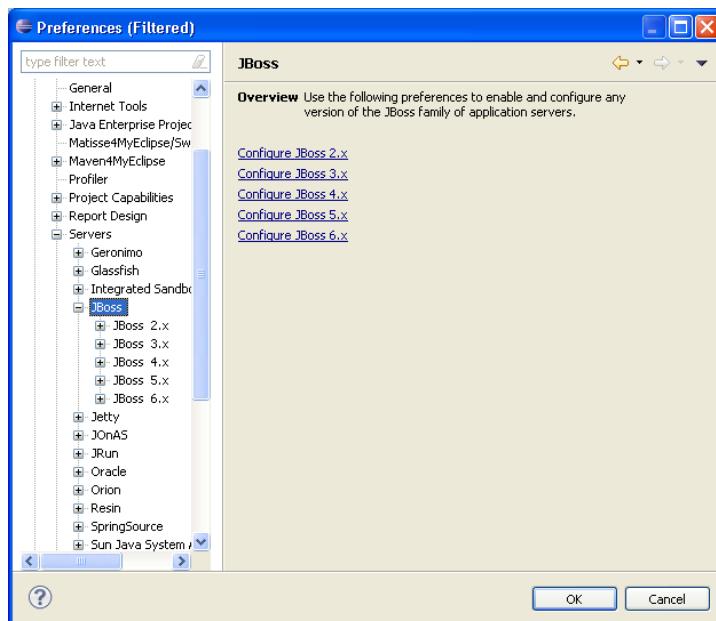
<!-- ----- -->

```

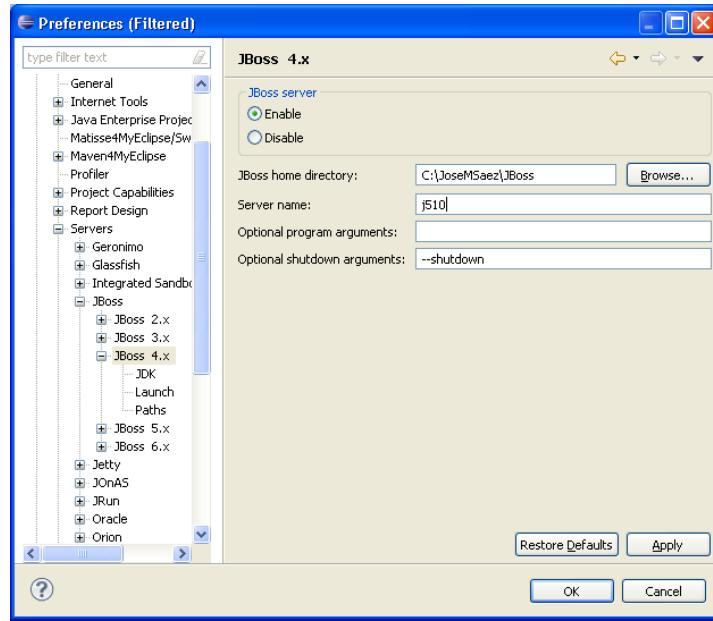
Existen ejemplos de configuración dentro del paquete de archivos de JBoss, en la carpeta docs/examples, que nos puede ayudar para la configuración del servidor:



En MyEclipse le explicaremos donde esta el JBoss, para ello vamos a MyEclipse – preferences, abrimos MyEclipse – Server – Jboss:

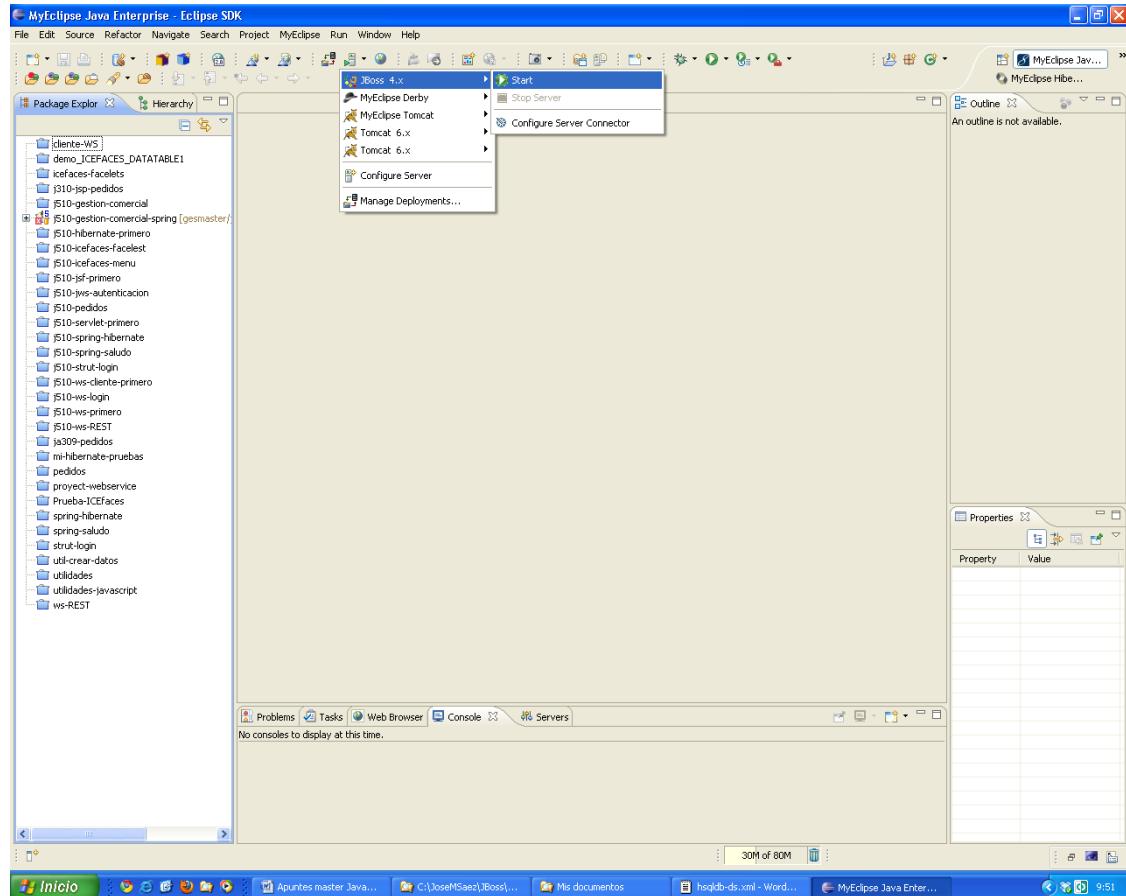


Como nuestra versión es 4.x vamos a 4.x

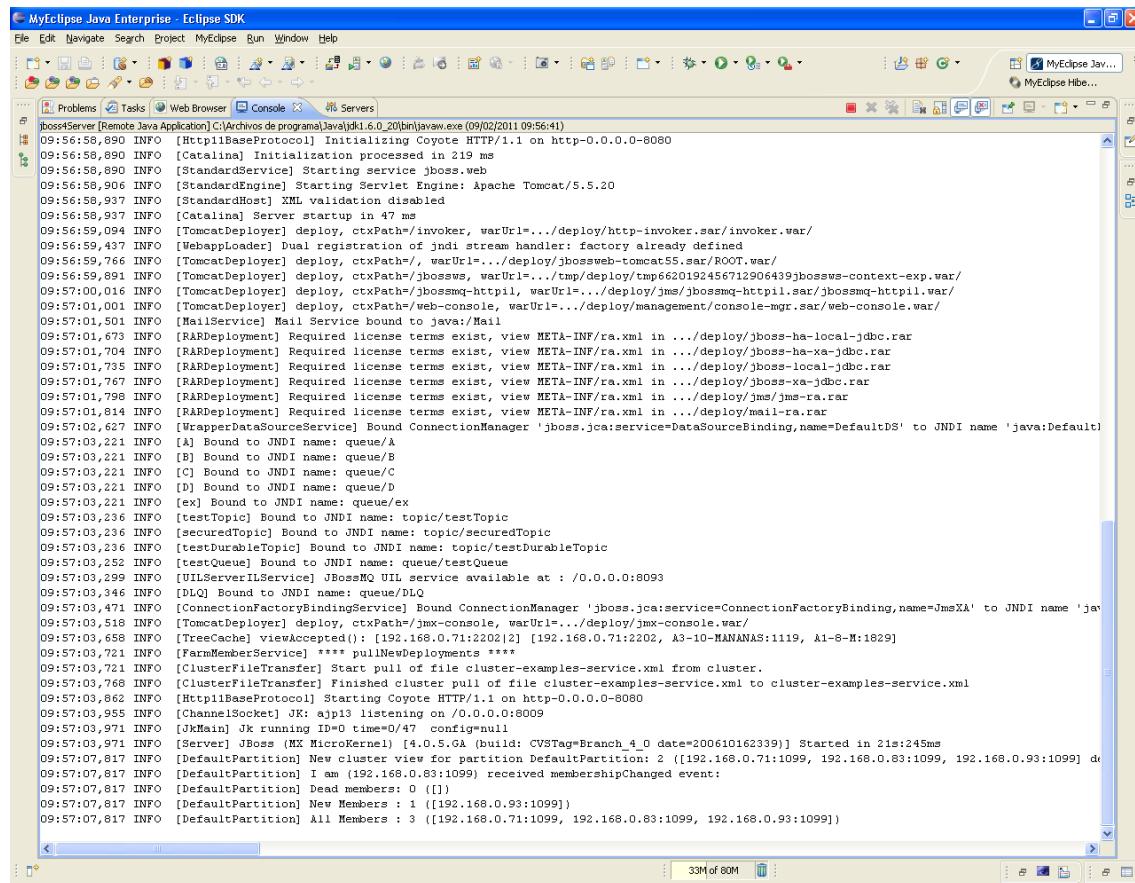


Enable, buscamos la carpeta “jboss” donde lo instalamos, le decimos el nombre del servidor, en nuestro caso era j510, lo demás como esta y le damos a OK.

Lo probamos pulsando el icono del servidor donde ahora aparece una opción de JBoss



En la consola podremos ver el proceso de instalación del JBoss. Como este proceso implica un cliente y un servidor, tendremos disponibles dos consolas una para el cliente y otra para el servidor.

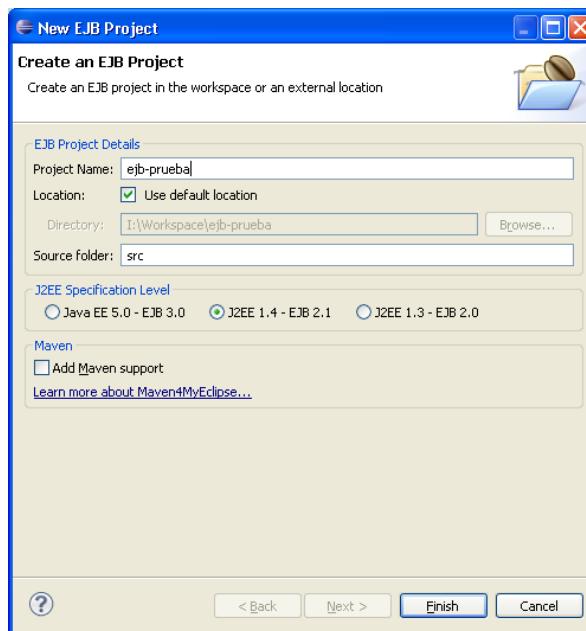
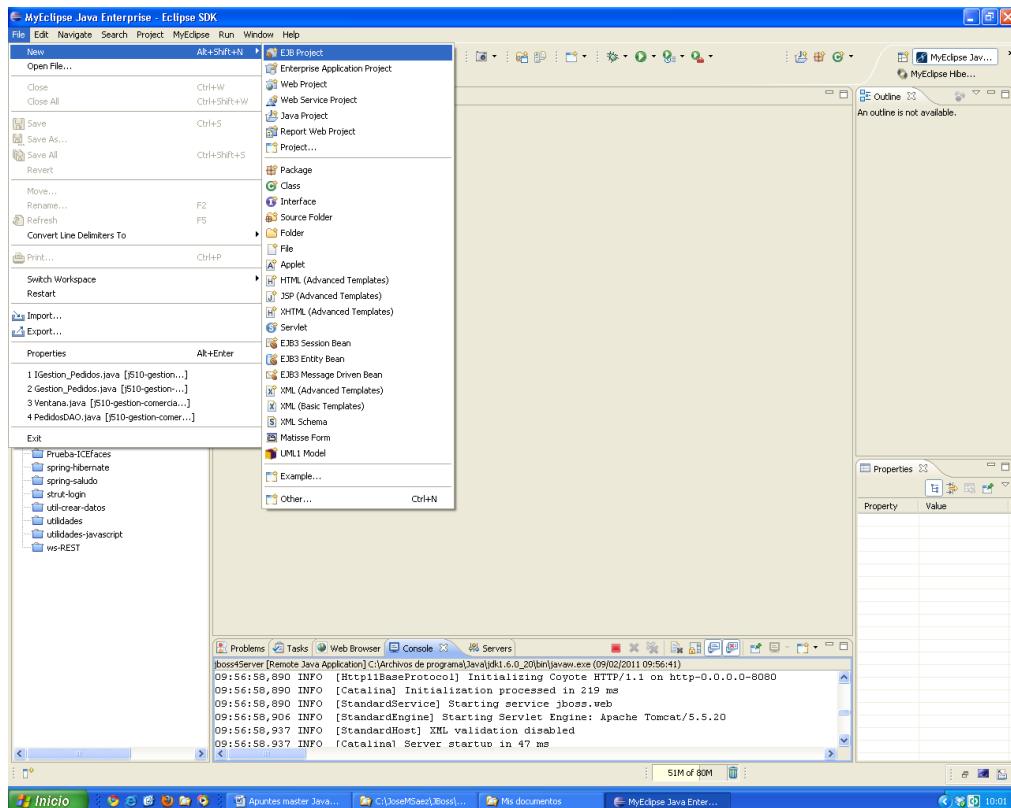


The screenshot shows the MyEclipse Java Enterprise IDE interface. The main window has a title bar "MyEclipse Java Enterprise - Eclipse SDK". The menu bar includes File, Edit, Navigate, Search, Project, MyEclipse, Run, Window, Help. The toolbar contains various icons for file operations like Open, Save, Cut, Copy, Paste, etc. The central area is a "Console" tab showing the output of a JBoss server. The log output is as follows:

```
jboss4Server [Remote Java Application] C:\Archivos de programa\Java\jdk1.6.0_20\bin\javaw.exe (09/02/2011 09:56:41)
09:56:58,890 INFO  [Http11BaseProtocol] Initializing Coyote HTTP/1.1 on http-0.0.0.0-8080
09:56:58,890 INFO  [Catalina] Initialization processed in 219 ms
09:56:58,890 INFO  [StandardService] Starting service jboss.web
09:56:58,906 INFO  [StandardEngine] Starting Servlet Engine: Apache Tomcat/5.5.20
09:56:58,937 INFO  [StandardHost] XML validation disabled
09:56:58,937 INFO  [Catalina] Server startup in 47 ms
09:56:59,094 INFO  [TomcatDeployer] deploy, ctxPath=/invoker, warUrl=.../deploy/http-invoker.sar/invoker.war/
09:56:59,437 INFO  [WebappLoader] Dual registration of jndi stream handler: factory already defined
09:56:59,766 INFO  [TomcatDeployer] deploy, ctxPath=/, warUrl=.../deploy/jbossweb-tomcat55.sar/ROOT.war/
09:56:59,891 INFO  [TomcatDeployer] deploy, ctxPath=/jbossws, warUrl=.../tmp/deploy/tmp6620192456712906439/jbossws-context-exp.war/
09:57:00,016 INFO  [TomcatDeployer] deploy, ctxPath=/jbossmq-http1, warUrl=.../deploy/jbossmq-http1.sar/jbossmq-http1.war/
09:57:01,001 INFO  [TomcatDeployer] deploy, ctxPath=/web-console, warUrl=.../deploy/management/console-mgr.sar/web-console.war/
09:57:01,501 INFO  [MailService] Mail Service bound to java:Mail
09:57:01,673 INFO  [RARDeployment] Required license terms exist, view META-INF/ra.xml in .../deploy/jboss-ha-local-jdbc.rar
09:57:01,704 INFO  [RARDeployment] Required license terms exist, view META-INF/ra.xml in .../deploy/jboss-ha-xa-jdbc.rar
09:57:01,735 INFO  [RARDeployment] Required license terms exist, view META-INF/ra.xml in .../deploy/jboss-local-jdbc.rar
09:57:01,767 INFO  [RARDeployment] Required license terms exist, view META-INF/ra.xml in .../deploy/jboss-xa-jdbc.rar
09:57:01,798 INFO  [RARDeployment] Required license terms exist, view META-INF/ra.xml in .../deploy/jms/jms-ra.rar
09:57:01,814 INFO  [RARDeployment] Required license terms exist, view META-INF/ra.xml in .../deploy/mail-ra.rar
09:57:02,627 INFO  [WrapperDataSourceService] Bound ConnectionManager 'jboss.jca:service=DataSourceBinding,name=DefaultDS' to JNDI name 'java:Default'
09:57:03,221 INFO  [A] Bound to JNDI name: queue/queueA
09:57:03,221 INFO  [B] Bound to JNDI name: queue/queueB
09:57:03,221 INFO  [C] Bound to JNDI name: queue/C
09:57:03,221 INFO  [D] Bound to JNDI name: queue/D
09:57:03,221 INFO  [ex] Bound to JNDI name: queue/ex
09:57:03,236 INFO  [testTopic] Bound to JNDI name: topic/testTopic
09:57:03,236 INFO  [securedTopic] Bound to JNDI name: topic/securedTopic
09:57:03,236 INFO  [testurableTopic] Bound to JNDI name: topic/testDurableTopic
09:57:03,252 INFO  [testQueue] Bound to JNDI name: queue/testQueue
09:57:03,299 INFO  [UILServerILService] JBossM2 UI service available at : /0.0.0.0:8093
09:57:03,346 INFO  [DLQ] Bound to JNDI name: queue/DLQ
09:57:03,471 INFO  [ConnectionFactoryBindingService] Bound ConnectionManager 'jboss.jca:service=ConnectionFactoryBinding,name=JmsXA' to JNDI name 'java:Default'
09:57:03,518 INFO  [TomcatDeployer] deploy, ctxPath=/jmx-console, warUrl=.../deploy/jmx-console.war/
09:57:03,658 INFO  [TreeCache] viewAccepted(): [192.168.0.71:2202] [192.168.0.71:2202, A3-10-MANANAS:1119, A1-8-M:1829]
09:57:03,721 INFO  [FarmMemberServices] *** pullNewDeployments ***
09:57:03,721 INFO  [ClusterFileTransfer] Start pull of file cluster-examples-service.xml from cluster.
09:57:03,768 INFO  [ClusterFileTransfer] Starting cluster pull of file cluster-examples-service.xml to cluster-examples-service.xml
09:57:03,862 INFO  [Http11BaseProtocol] Starting Coyote HTTP/1.1 on http-0.0.0.0-8080
09:57:03,955 INFO  [ChannelSocket] JK: ajp13 listening on /0.0.0.0:8009
09:57:03,971 INFO  [JkMain] JK running ID=0 time=0/47 config=null
09:57:03,971 INFO  [Server] JBoss (MX MicroKernel) [4.0.5.GA (build: CVSTag=Branch_4_0 date=200610162339)] Started in 21s:245ms
09:57:07,817 INFO  [DefaultPartition] New cluster view for partition DefaultPartition: 2 ([192.168.0.71:1099, 192.168.0.83:1099, 192.168.0.93:1099] down)
09:57:07,817 INFO  [DefaultPartition] I am (192.168.0.83:1099) received membershipChanged event:
09:57:07,817 INFO  [DefaultPartition] Dead members: 0 (!)
09:57:07,817 INFO  [DefaultPartition] New Members : 1 ((192.168.0.93:1099))
09:57:07,817 INFO  [DefaultPartition] All Members : 3 ((192.168.0.71:1099, 192.168.0.83:1099, 192.168.0.93:1099))
```

Creación de un Proyecto EJB con MyEclipse

Boton derecho - new - EJB Proyect.



Le damos nombre y especificación acorde a nuestro JBoss, Finish

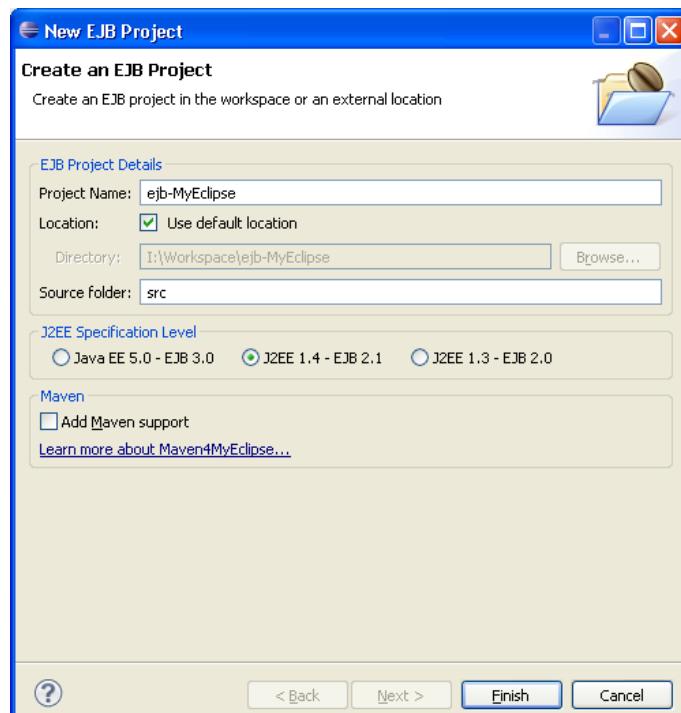
Nos crea el árbol de directorios e importa las librerías necesarias.

Hemos de tener en cuenta que un proyecto EJB irá junto con un proyecto J2EE o J2SE (el cliente) y luego han de ir enlazados, para eso existe una opción al dar file – New que es **Enterprise Application Project**, ese es un empaquetado que junta el proyecto J2EE (WAR) con el Proyecto EJB.

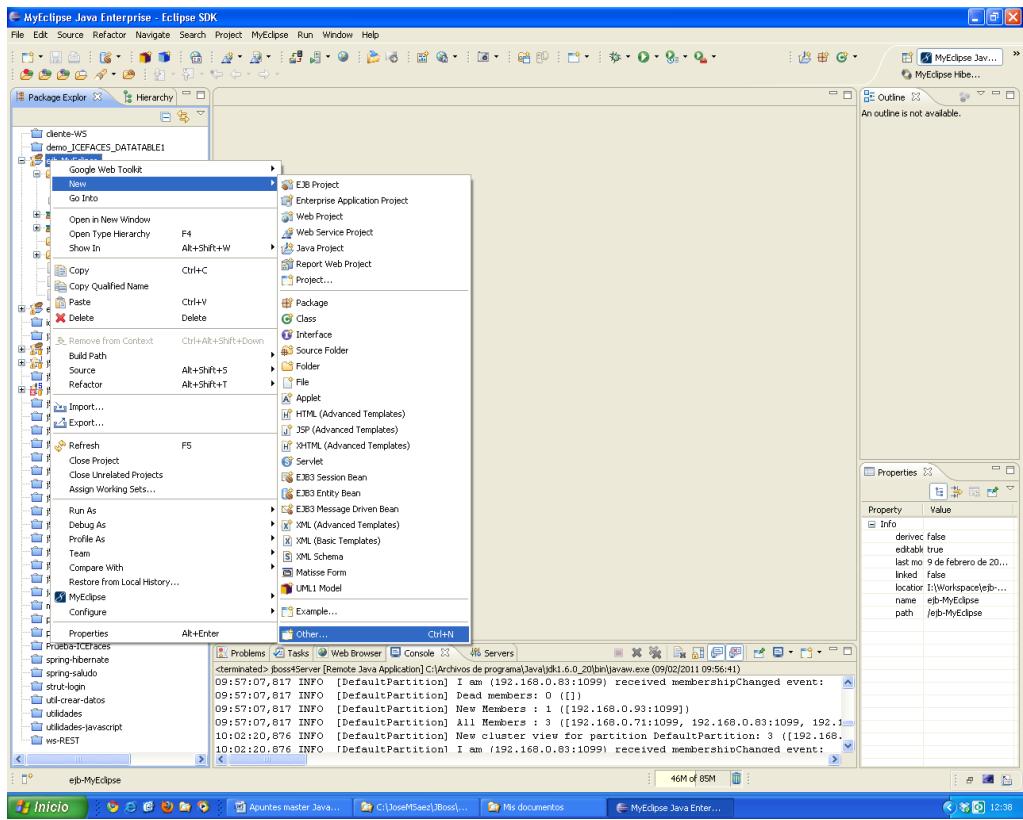
Como ya sabemos en un proyecto EJB necesitaremos bien 2 o bien 4 Interfaces según nuestro proyecto vaya a ser accedido por otro objeto Java que este en la misma máquina virtual o fuera de ella. Una vez conozcamos este dato procederemos a crear estas interfaces, que extienden EJBObject la remota y la Home extiende EJBHome. Esto está explicado más arriba al principio de este capítulo de EJB.

Con MyEclipse puede automatizarse una buena parte del proceso. Así:

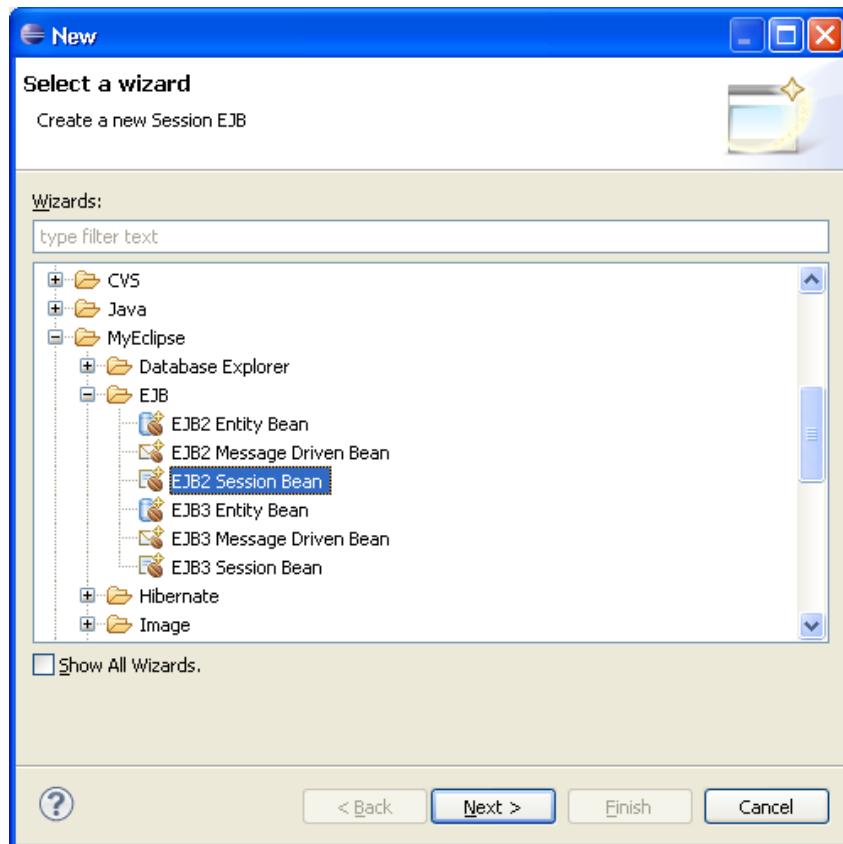
Creamos un New – EJB Project



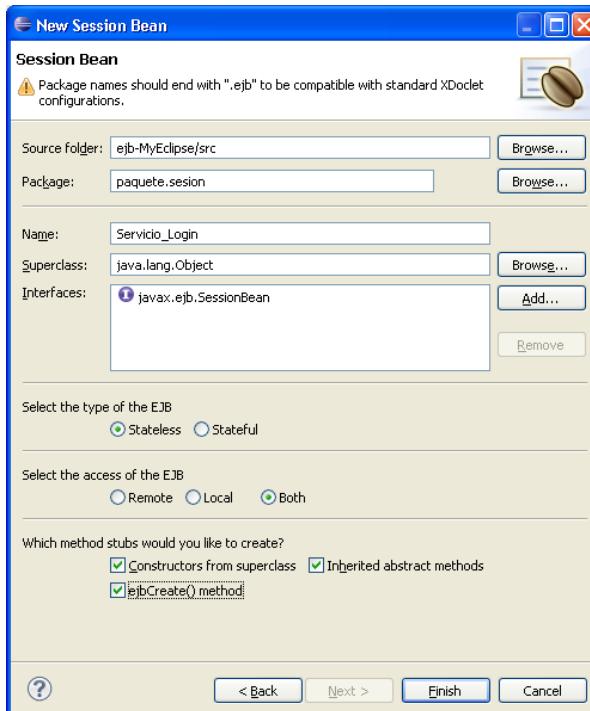
Elejimos las opciones adecuadas a nuestro proyecto. Pulsamos Finish.



Boton derecho en el proyecto y New – Other.



Elejimos la opción de EJB dentro de MyEclipse que vamos a usar.

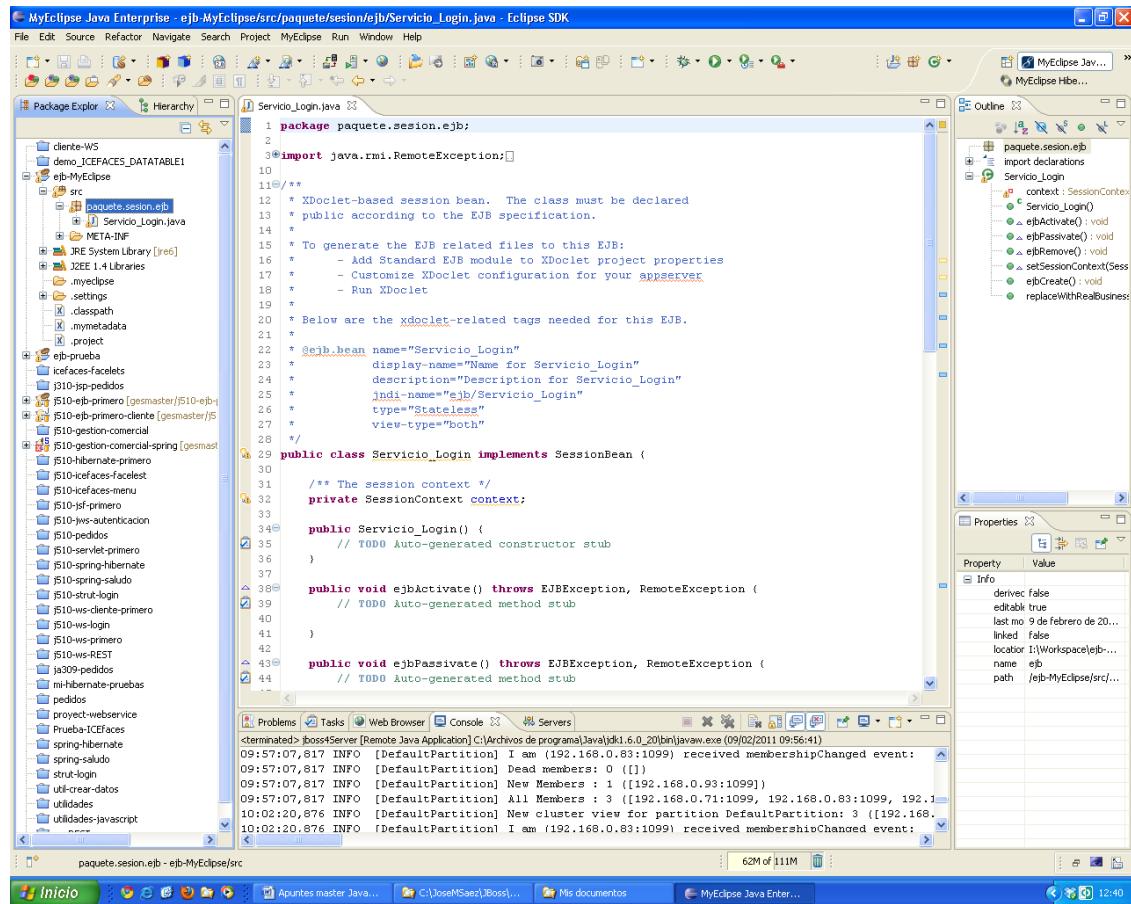


Rellenamos los datos del EJB según nos interese en nuestro proyecto

Nos informa de que el nombre del paquete debe terminar en .ejb en caso de que no lo hayamos hecho así, lo necesitaremos para que el siguiente paso funcione.

Finish

Nos creara una clase según le hemos indicado



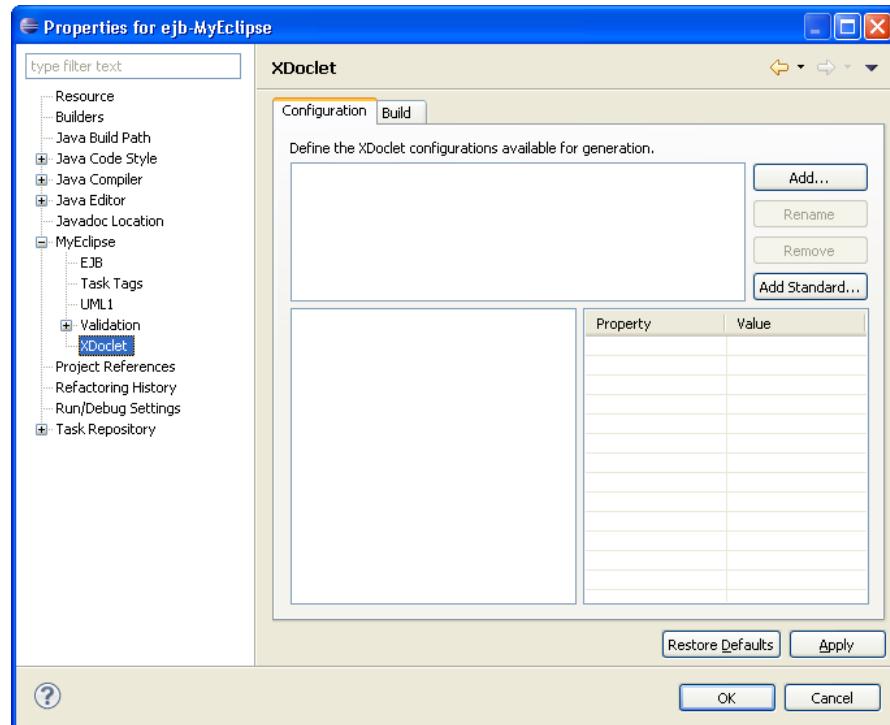
Esa clase ya lleva los métodos implementados de su interfaz según vimos en la parte teórica de este capítulo. Además escribe las anotaciones a partir de los parámetros que le hemos indicado.

Por otro lado, existe una herramienta llamada **XDoclet** que genera el código automáticamente. XDoclet es el paso intermedio entre Doclet y las anotaciones. Aunque hoy en día la tendencia es a no usar esta herramienta, EJB 2 es una tecnología algo antigua y puede usarse XDoclet.

Para usarla iríamos a la clase que se nos ha generado y la adaptaremos a nuestras necesidades, podemos quitar alguno de los métodos generados que no vamos a necesitar como **replaceWithRealBusinessMethod** que existe con el único propósito de que lo usemos para cambiarle el nombre y le demos el uso personalizado que queramos.

A los proyectos EJB como este podemos darles capacidades de Hibernate tal como hemos visto en otros proyectos no cambia nada. Así mismo podemos darle capacidades de Spring, en ese orden, primero capacidades de Hibernate y luego de Spring. Solo hay que tener en cuenta que al darle capacidades de Hibernate + Spring le digamos a hibernate que no cree el SessionFactoryClass.

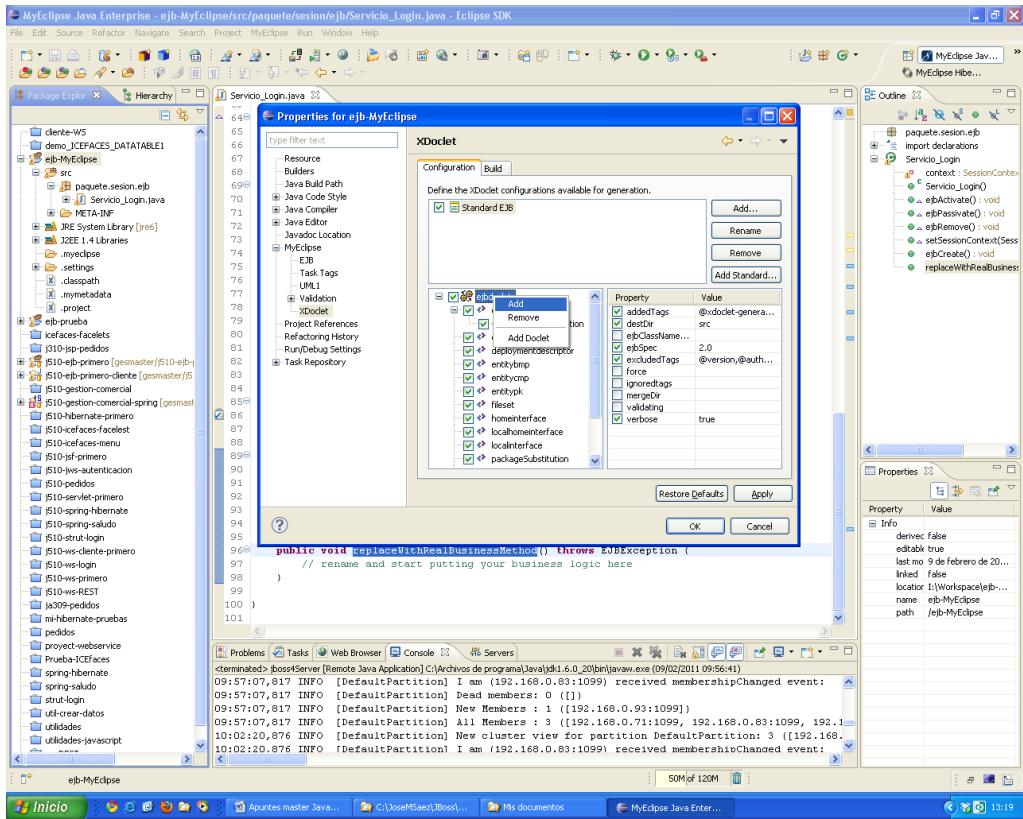
Nos crearemos las fachadas que vayamos a necesitar y para usar **XDoclet**, como esta herramienta sirve para crear cualquier tipo de código hay que indicarle el tipo de código que vamos a usar, en este caso vamos a el proyecto damos botón derecho – Properties – My Eclipse – XDoclet



Pulsamos en el botón Add Standard.. y se abre esta ventana:



Elejimos Standard EJB para este caso concreto, pulsamos OK

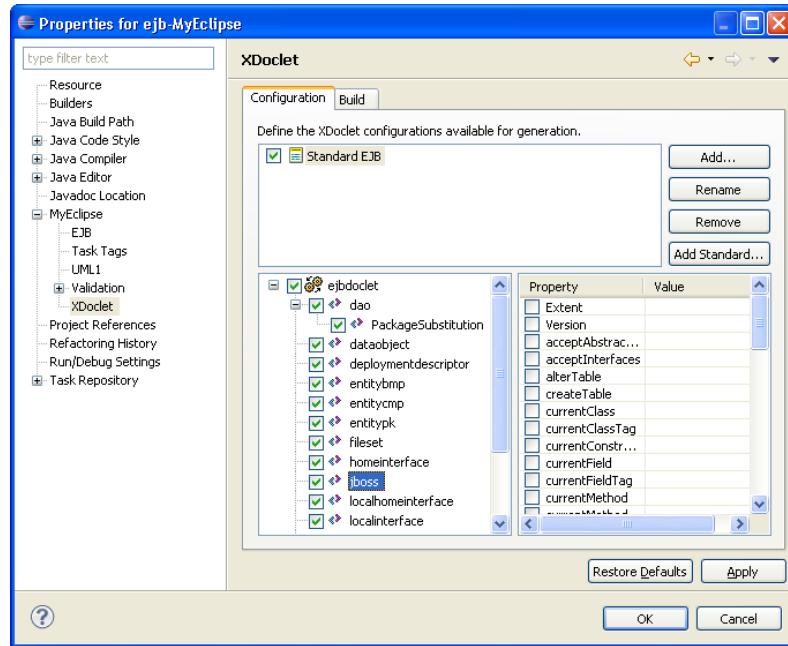


Nos añade la entrada a la ventana anterior, si marcamos esa entrada, y luego con el botón derecho pulsamos sobre ejbdoclet, sale un menú contextual, pulsando la opción add sale un menú:

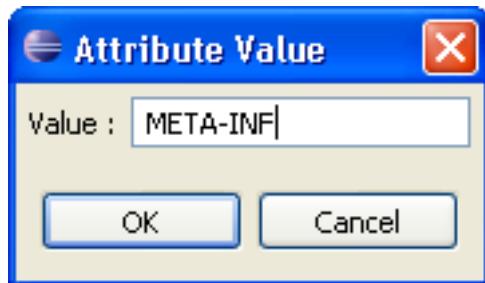


En nuestro caso elegimos jboss y pulsamos OK, eso añade JBoss a las opciones de la ventana anterior.

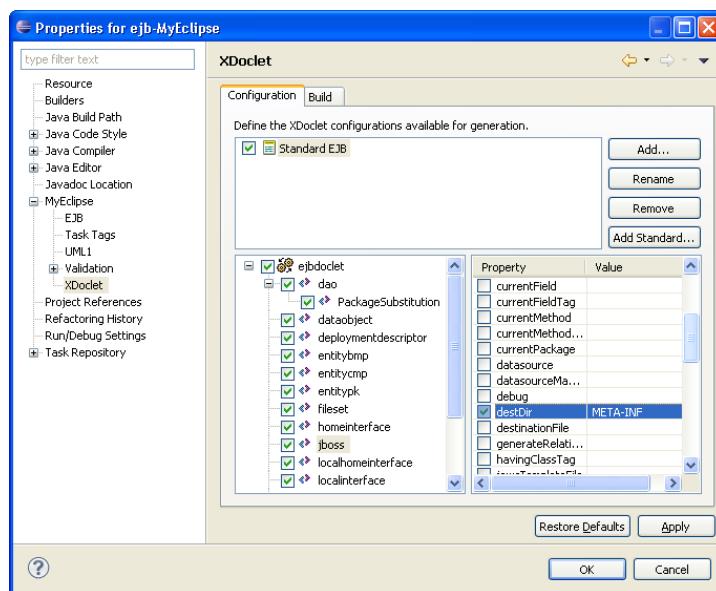
Pulsamos en la opción jboss de la ventana anterior.



Y rellenamos las opciones del lado derecho, en concreto versión (le damos valor 4.0) y el fichero de destino en dest-DIR le damos valor META-INF.

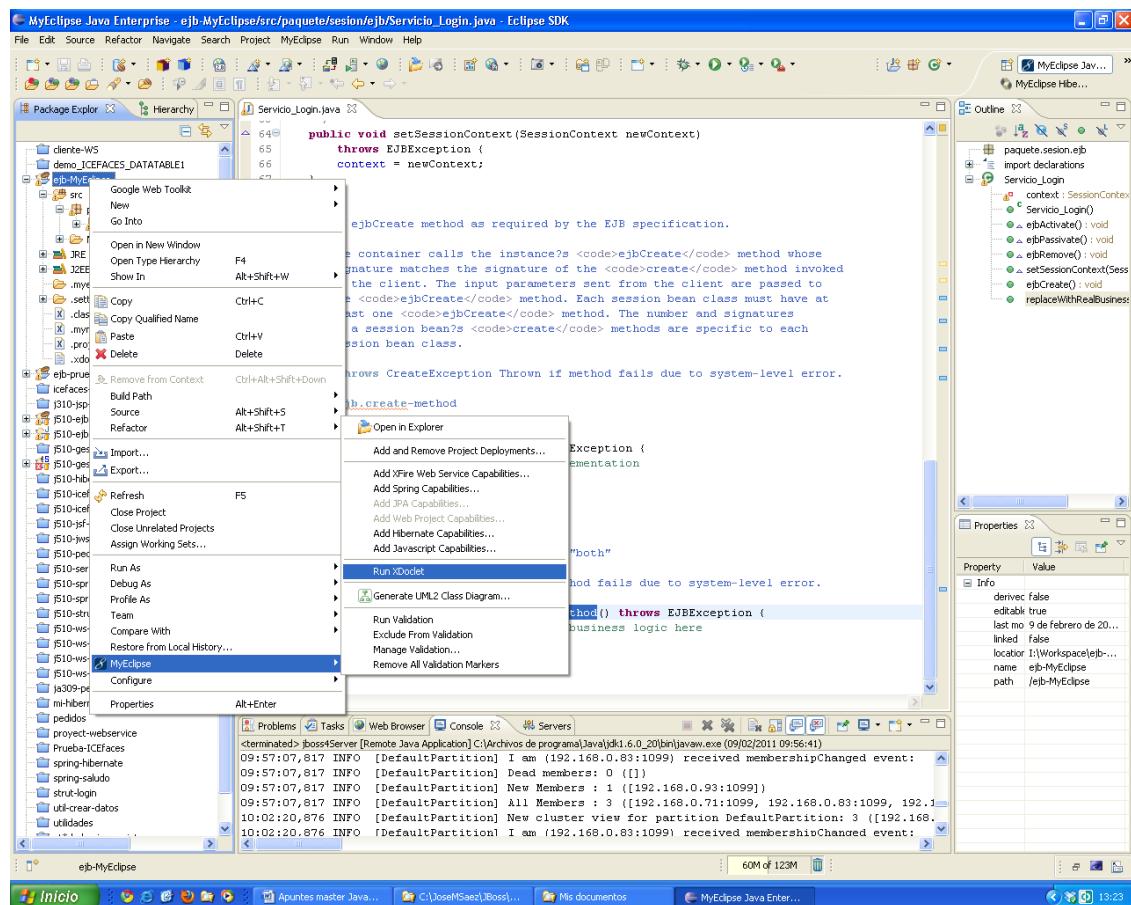


Quedando así:

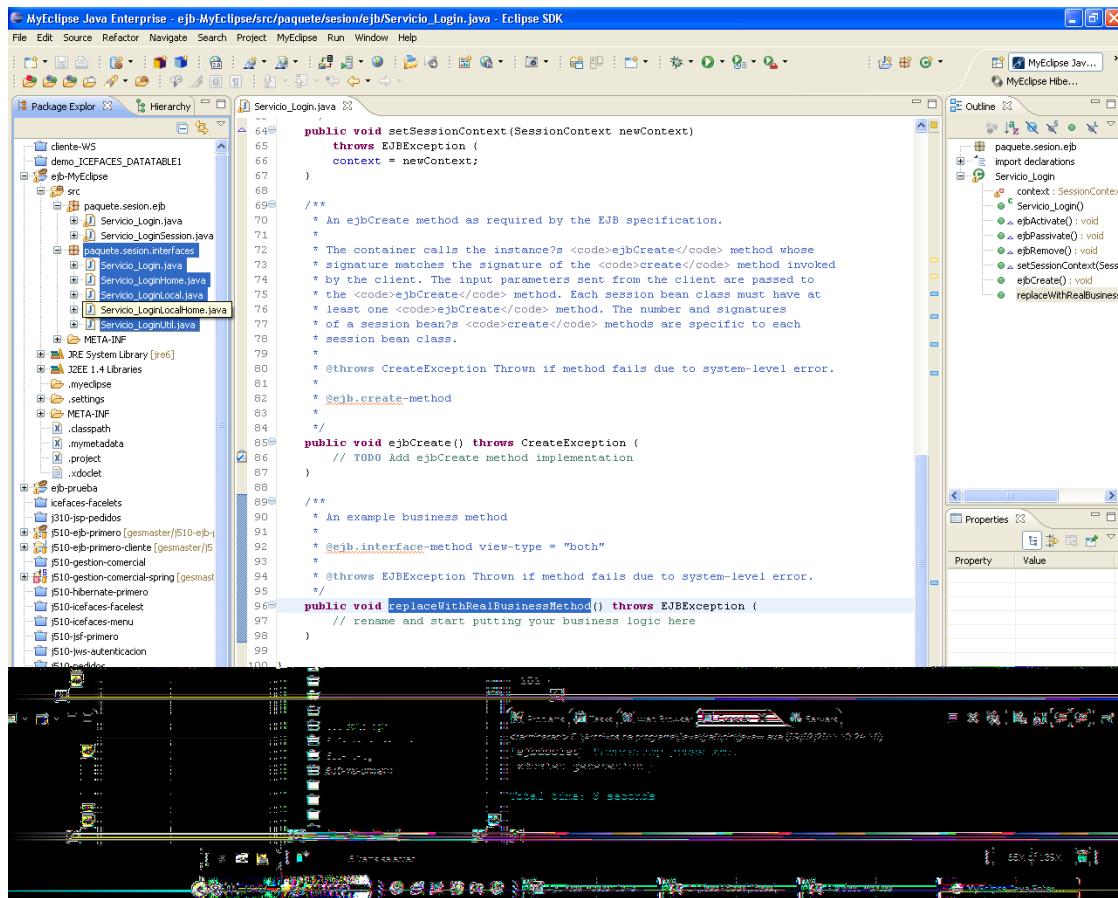


Le damos OK y volvemos a MyEclipse

Para generar el proyecto usamos la opción Run XDoclet



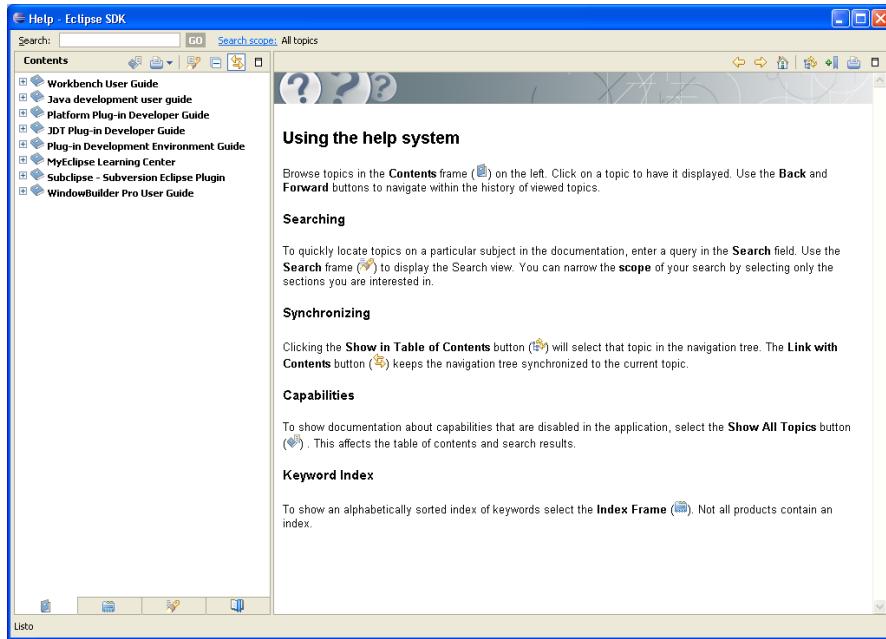
Al hacer eso se generan automáticamente las cuatro interfaces.



Como se puede ver, además de las cuatro interfaces hay una clase `Servicio_LoginUtil` que sirve para instanciarse el servicio y la conexión que en nuestro caso era de Login. Lo que antes habíamos hecho a mano.

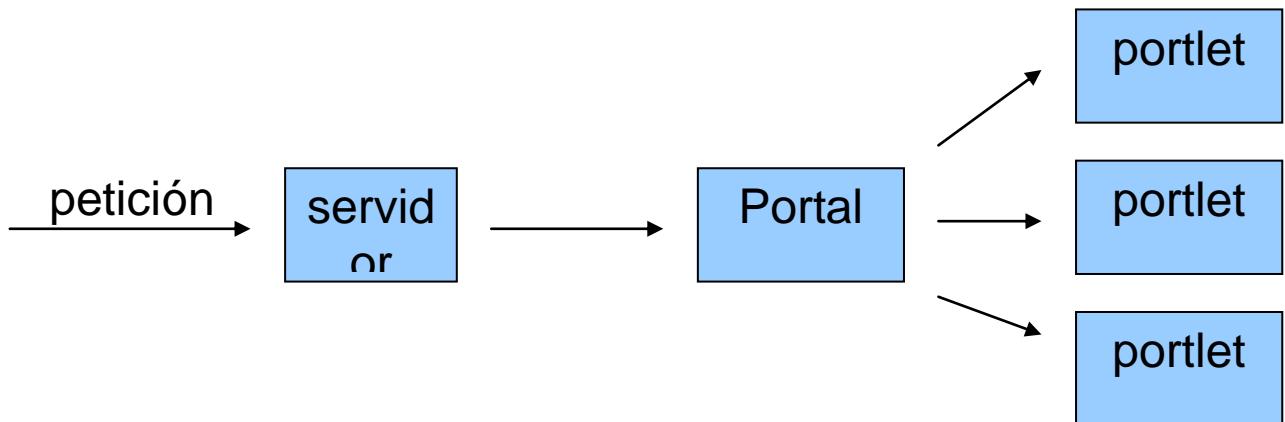
Además el solo genera el ejb.jar, el fichero descriptor también.

MyEclipse tiene una ayuda muy completa, que tanto para este tema de los servicios EJB, como para cualquier otro de los ya vistos en el master puede ser de gran utilidad. Para acceder a dicha ayuda hay que ir a Help – Help contents y aparece esta ventana:



Desde esta venta se pueden buscar o se puede navegar por las opciones que nos ofrece. En concreto en el menu de la izquierda hay una opcion **MyEclipse Learning Center** que contiene una enorme variedad de temas. Este tema en concreto de EJB y Doclet esta tratado en profundidad.

Portlets



Un portlet es un fragmento de una página, que se verá solo o acompañado de otros portlets. De forma que son independientes y reutilizables por cualquier otra aplicación. Están respaldados por Sun. Los portlets tienen versión 1 y 2. La primera viene de 2005, pero en los dos últimos años han tenido un crecimiento porque, los portlets, reducen costes en el desarrollo de las aplicaciones con reutilización.

Los portlets tienen que funcionar dentro de un gestor de portlets, es decir, un portal. Un portal es una aplicación que se monta dentro de un servidor web.

Los portlets están pensados para ser independientes y para tener una gran interacción con el usuario final.

Como todo componente tiene un ciclo de vida que será gestionado por el portal, no por el servidor.

Tiene dos métodos para inicio y fin

El método `processAction` cuando es llamado, implica que haya una lógica de negocio y recibe los contextos. No tiene los contextos de J2EE, sino el `ActionRequest` y `ActionResponse`.

El método `render()` es llamado para repintar la vista. Puede ser llamado sin haber invocado al `processAction`. Recibe `RenderRequest` (que no trae los parámetros de la petición) y `RenderResponse`. `RenderRequest` y `RenderResponse`.

Nos permite definir estados o modos: la especificación nos habla de 3, view, edit y help. Y todo portlet puede tener cualquier combinación de los tres o los tres. Esto se traduce en

una vista y operativa distinta.

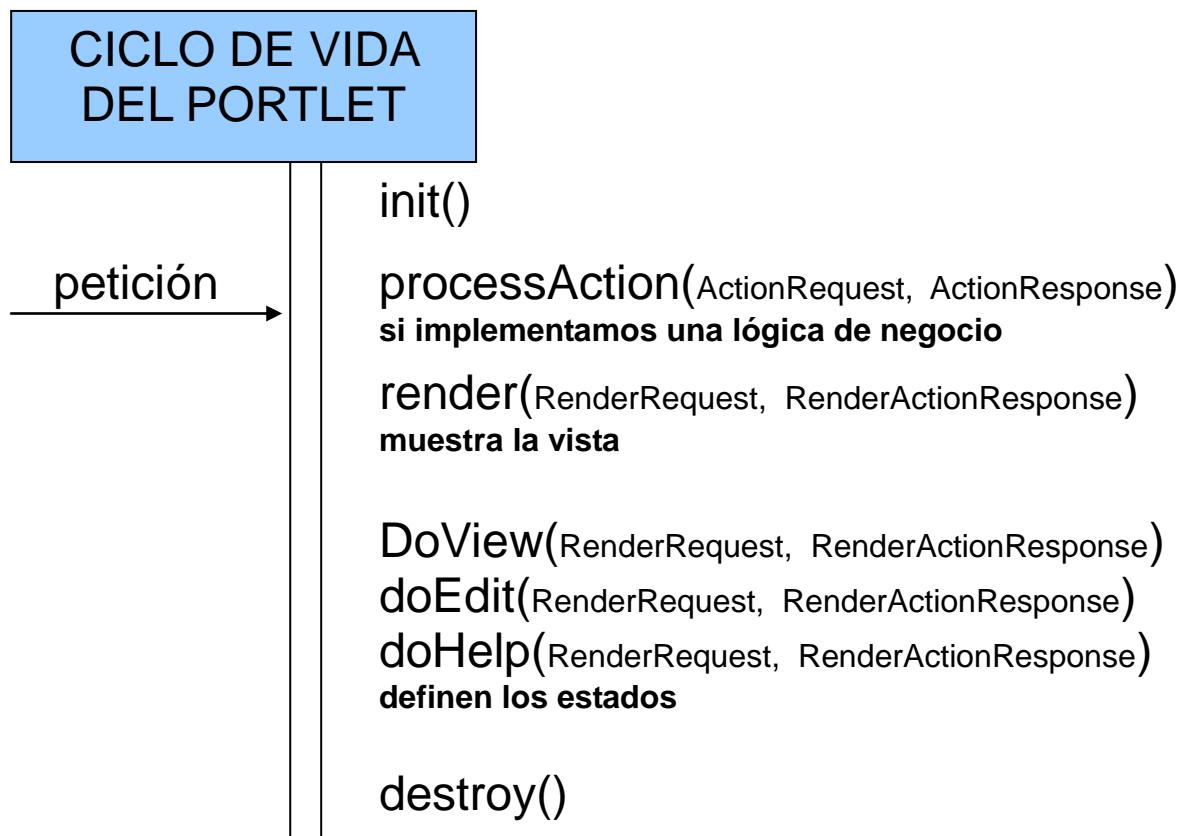
El modo **view** tiene que ser la operativa normal. Por ejemplo mostrar la temperatura del dia o el estado de la bolsa.

El modo **edit**, debe usarse solo para que el usuario cambie las preferencias del portlet que el programador decida dejarle modificar.

El modo **help** se debe utilizar para dar cualquier tipo de asistencia o ayuda al usuario.

Como tenemos 3 posibles estados, tras el render habrá de llamarse a uno de los 3 métodos (doView(), doEdit(), doHelp()). Estos métodos reciben como argumentos el renderRequest, renderResponse. Desde estos tres métodos se llamará a la vista que se quiera mostrar a continuación.

Este proceso completo se repite por cada portlet. Se pueden establecer comunicación interportlet mediante el uso de escuchadores.



El portlet no se define en el web.xml, sino que tiene un portlet.xml

```
<portlet>
    <portlet-name>
```

```

<portlet-class>
    <support>
        <mime-type>text/HTML</mime-type>
        <portlet-mode>view</portlet-mode>
            <window-state>normal/
                maximize/minimize</window-
state>
        <portlet-mode>edit</portlet-mode>
        <portlet-mode>help</portlet-mode>
    </support>

</portlet-class>
</portlet-name>
<portlet-preferences name="" value="" read-
only="true/false">
    </portlet-preferences>
    <portlet-info>
        <title></title>
    </portlet-info>
</portlet>

```

portlet-name: nombre del mismo

portlet-class: clase de la instancia del portlet

support

mime-type: tipo de documento que es.

window-state: estado de ventana

- **normal:** con el tamaño definido
- **maximize:** cubre toda la pantalla, tapando los demás portlets
- **minimize:** queda oculto

portlet-preferences: si implementamos el modo edit, el usuario podrá modificar las preferencias modificadas aquí.

- **Name="”** nombre de la preferencia
- **value="”** valor asignado
- **read-only="true/false"** permiso para que el usuario la modifique

portlet-info:

- **title:** titulo que aparecerá en la ventana

Al definir una clase, lo haremos como en los servlets, heredando de una clase (GenericPortlet) y a partir de ahí, reescribimos los métodos que queramos o necesitemos.

Public class xxx extends GenericPortlet.

Las páginas pueden ser de cualquier tipo, lo que si necesitará serán las etiquetas personalizadas, que nos permitirán acceder al contexto de la página y a los objetos

cargados en él.

```
<%@ taglib uri="" prefix="portlet" %>
<portlet: defineobjectcts> nos carga todos los objetos en el
contexto de la pagina.
    <%= renderRequest.getPreference() %>
    <form action="renderResponse.createActionUrl">
        //urls dinamicas, necesito un objeto que lo calcule

    <form action="renderResponse.createRenderUrl">
        //opcion para llamar a una vista
```

Pluto

El servlet pluto es de Apache. Hay que definirlo en el web.xml.

Web.xml

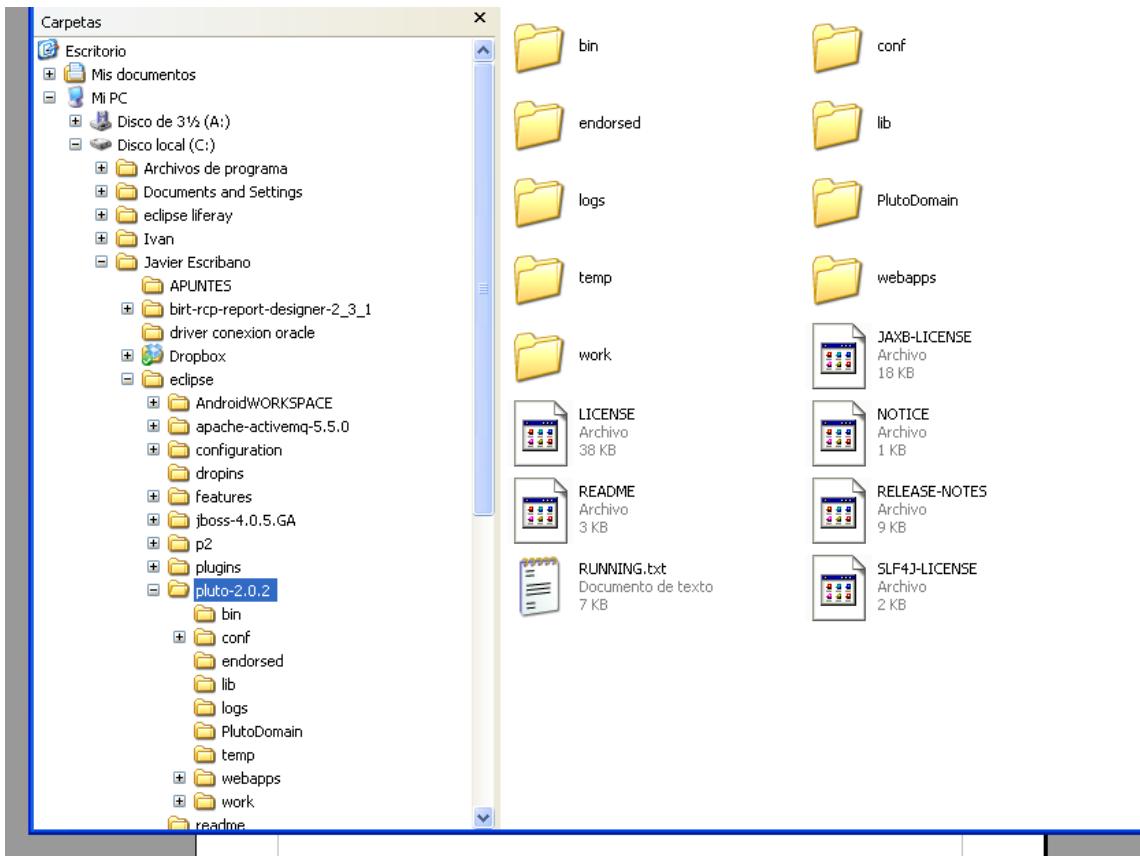
```
<servlet>
    <servlet-name>nombre del servlet</servlet-name>
    <servlet-class>
        org.apache.pluto.container.driver.PortletServlet
    <servlet-class>

    <init-param>
        <param-name>nombre del parametro en el
web.xml</param-name>
        <param-value>nombrePortlet</param-value>
    </init-param>
</servlet>

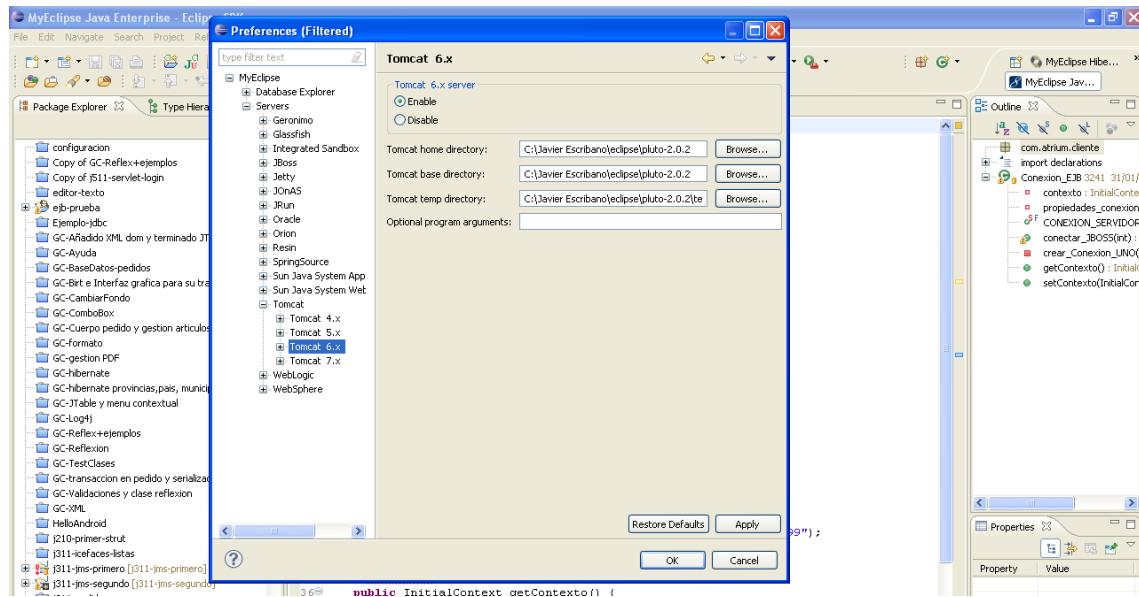
<servlet-mapping>
    <servlet-name>nombre del servlet</servlet-name>
    <url-pattren>/PlutoInvoker/nombrePortlet</url-pattren>
</servlet-mapping>
```

Instalacion de Pluto

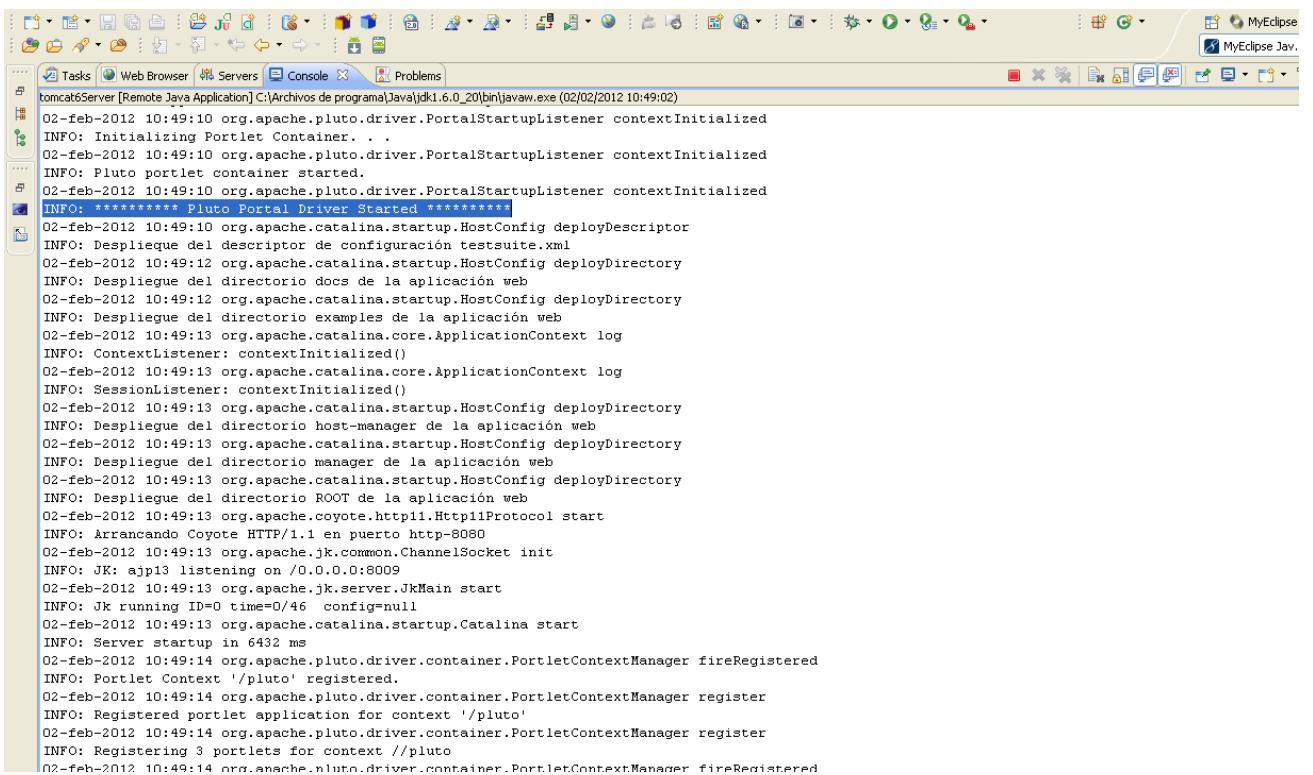
Instalamos el archivo de apache pluto-2.0.2.zip que está instalado sobre tomcat. Para ello solo tenemos que descomprimir el archivo en la carpeta que queramos, como en otras ocasiones, lo haremos en la misma carpeta que la instalación de eclipse



Como ya estamos utilizando un tomcat en eclipse, tendremos que darle la carpeta del “nuevo tomcat” en la carpeta de configuración de servidores de eclipse.



Al arrancar el servidor vermos como la consola nos muestra que Pluto ha arrancado.



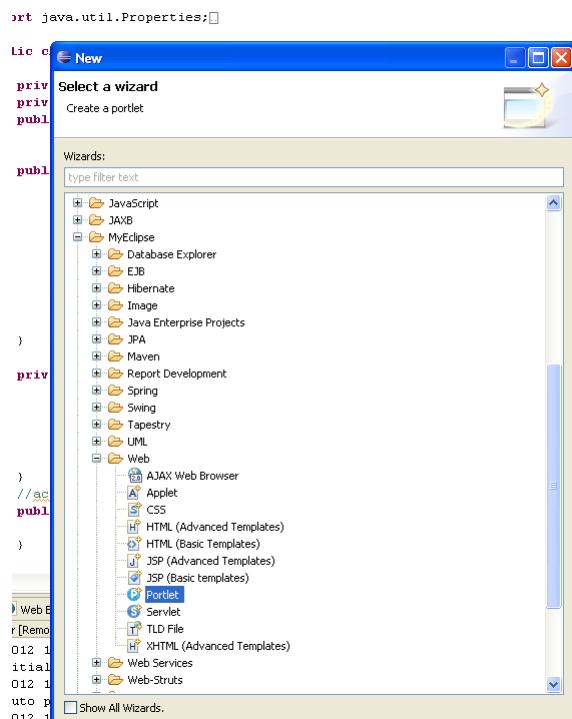
The screenshot shows the MyEclipse Java IDE interface. The title bar says "MyEclipse Java". The main window has several tabs: Tasks, Web Browser, Servers, Console, and Problems. The "Console" tab is selected and displays the log output from the "tomcat6Server [Remote Java Application]". The log shows the startup process of the Tomcat 6 server, including the initialization of the Pluto portlet container and the deployment of the web application. The log ends with the message "INFO: Registering 3 portlets for context //pluto".

```
tomcat6Server [Remote Java Application] C:\Archivos de programa\Java\jdk1.6.0_20\bin\javaw.exe (02/02/2012 10:49:02)
02-feb-2012 10:49:10 org.apache.pluto.driver.PortalStartupListener contextInitialized
INFO: Initializing Portlet Container. .
02-feb-2012 10:49:10 org.apache.pluto.driver.PortalStartupListener contextInitialized
INFO: Pluto portlet container started.
02-feb-2012 10:49:10 org.apache.pluto.driver.PortalStartupListener contextInitialized
INFO: ***** Pluto Portal Driver Started *****
02-feb-2012 10:49:10 org.apache.catalina.startup.HostConfig deployDescriptor
INFO: Despliegue del descriptor de configuración testsuite.xml
02-feb-2012 10:49:12 org.apache.catalina.startup.HostConfig deployDirectory
INFO: Despliegue del directorio docs de la aplicación web
02-feb-2012 10:49:12 org.apache.catalina.startup.HostConfig deployDirectory
INFO: Despliegue del directorio examples de la aplicación web
02-feb-2012 10:49:13 org.apache.catalina.core.ApplicationContext log
INFO: ContextListener: contextInitialized()
02-feb-2012 10:49:13 org.apache.catalina.core.ApplicationContext log
INFO: Sessionlistener: contextInitialized()
02-feb-2012 10:49:13 org.apache.catalina.startup.HostConfig deployDirectory
INFO: Despliegue del directorio host-manager de la aplicación web
02-feb-2012 10:49:13 org.apache.catalina.startup.HostConfig deployDirectory
INFO: Despliegue del directorio manager de la aplicación web
02-feb-2012 10:49:13 org.apache.catalina.startup.HostConfig deployDirectory
INFO: Despliegue del directorio ROOT de la aplicación web
02-feb-2012 10:49:13 org.apache.coyote.http11.Http11Protocol start
INFO: Arrancando Coyote HTTP/1.1 en puerto http-8080
02-feb-2012 10:49:13 org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
02-feb-2012 10:49:13 org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/46 config=null
02-feb-2012 10:49:13 org.apache.catalina.startup.Catalina start
INFO: Server startup in 6432 ms
02-feb-2012 10:49:14 org.apache.pluto.driver.container.PortletContextManager fireRegistered
INFO: Portlet Context '/pluto' registered.
02-feb-2012 10:49:14 org.apache.pluto.driver.container.PortletContextManager register
INFO: Registered portlet application for context '/pluto'
02-feb-2012 10:49:14 org.apache.pluto.driver.container.PortletContextManager register
INFO: Registering 3 portlets for context //pluto
02-feb-2012 10:49:14 org.apache.pluto.driver.container.PortletContextManager fireRegistered
```

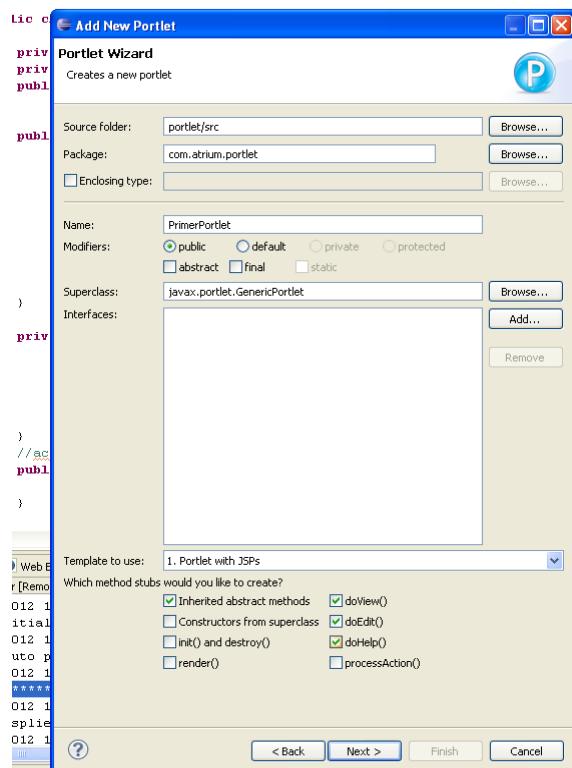
Creacion de un portlet en Eclipse

Creamos un nuevo Web Proyect y dentro de src, un paquete com.atrium.portlet.

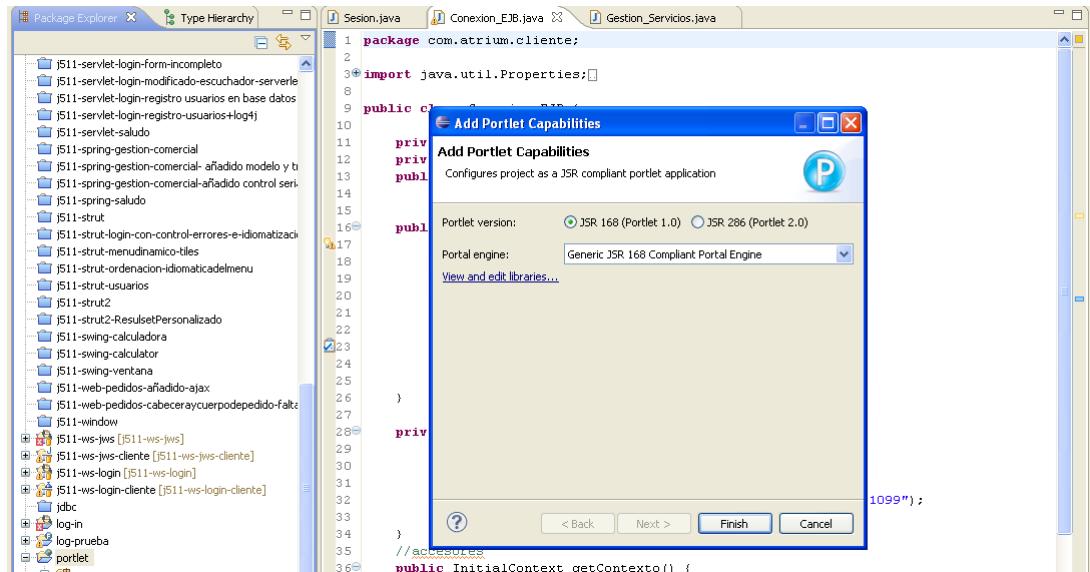
Tenemos que añadir capacidades de portlet al proyecto. Creamos un portlet
Elegimos el tipo de archivo portlet



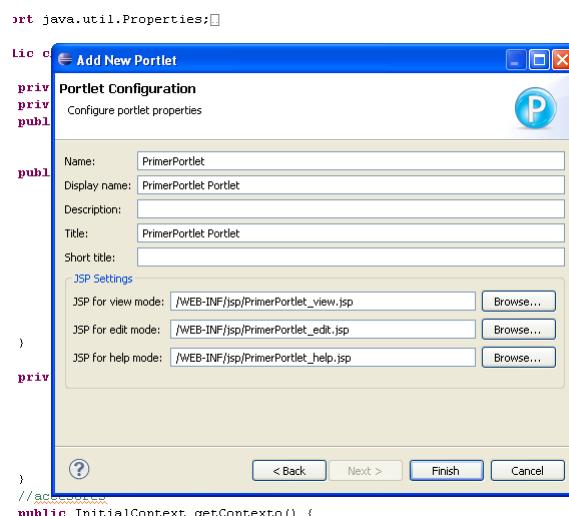
Elegiremos los tres modos



Elegimos la version 2



Siguiente



Damos a Siguiente y nos crea la clase. En esta clase FALTA PONER EL MENSAJE.

```
public class PrimerPortlet extends GenericPortlet {

    /**
     * Helper method to serve up the mandatory view mode.
     */
    protected void doHelp(RenderRequest request, RenderResponse
    response)
        throws PortletException, IOException {
        response.setContentType("text/html");
        PortletRequestDispatcher dispatcher = getPortletContext()
            .getRequestDispatcher("/WEB-
```

```

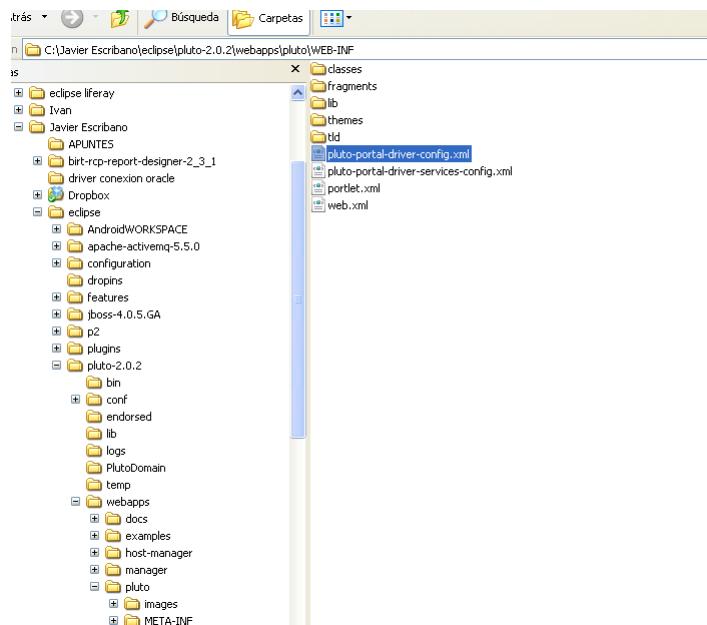
INF/jsp/PrimerPortlet_help.jsp");
    dispatcher.include(request, response);
}

/**
 * Helper method to serve up the mandatory view mode.
 */
protected void doEdit(RenderRequest request, RenderResponse
response)
    throws PortletException, IOException {
    response.setContentType("text/html");
    PortletRequestDispatcher dispatcher = getPortletContext()
        .getRequestDispatcher("/WEB-
INF/jsp/PrimerPortlet_edit.jsp");
    dispatcher.include(request, response);
}

/**
 * Helper method to serve up the mandatory view mode.
 */
protected void doView(RenderRequest request, RenderResponse
response)
    throws PortletException, IOException {
    response.setContentType("text/html");
    PortletRequestDispatcher dispatcher = getPortletContext()
        .getRequestDispatcher("/WEB-
INF/jsp/PrimerPortlet_view.jsp");
    dispatcher.include(request, response);
}
}

```

tendremos que ir al archivo de configuración para crear una nueva página de configuración.



The screenshot shows a Windows Notepad window displaying an XML configuration file. The file defines a portal driver named 'Pluto Portal Driver' with version 2.0.2. It specifies supported portlet modes (view, edit, help, config) and window states (normal, maximized, minimized). The 'render-config' section contains definitions for various pages and portlets, including 'About Apache Pluto', 'Test Page', 'JSR 286 Tests', 'Pluto Admin', and a custom page for 'Pruebaportlet'. The XML uses namespaces for pluto-portal-driver and pluto-portal-config.

```
See the License for the specific language governing permissions and
limitations under the License.
-->

<pluto-portal-driver
    xmlns="http://portals.apache.org/pluto/xsd/pluto-portal-driver-config.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://portals.apache.org/pluto/xsd/pluto-portal-driver-config.xsd
        http://portals.apache.org/pluto/pluto-portal/2.0/pluto-portal-driver-config.xsd"
    version="1.1">

<portal-name>pluto-portal-driver</portal-name>
<portal-version>2.0.2</portal-version>
<container-name>Pluto Portal Driver</container-name>

<supports>
    <portlet-mode>view</portlet-mode>
    <portlet-mode>edit</portlet-mode>
    <portlet-mode>help</portlet-mode>
    <portlet-mode>config</portlet-mode>

    <window-state>normal</window-state>
    <window-state>maximized</window-state>
    <window-state>minimized</window-state>
</supports>

<!-- Render configuration which defines the portal pages. -->
<render-config default="About Apache Pluto">
    <page name="About Apache Pluto" uri="/WEB-INF/themes/pluto-default-theme.jsp">
        <portlet context="/pluto" name="AboutPortlet"/>
        <portlet context="/testsuite" name="TestPortlet1"/>
    </page>
    <page name="Test Page" uri="/WEB-INF/themes/pluto-default-theme.jsp">
        <portlet context="/testsuite" name="TestPortlet1"/>
        <portlet context="/testsuite" name="TestPortlet2"/>
    </page>
    <page name="JSR 286 Tests" uri="/WEB-INF/themes/pluto-default-theme.jsp">
        <portlet context="/testsuite" name="286TestPortlet"/>
        <portlet context="/testsuite" name="286TestCompanionPortlet"/>
    </page>
    <page name="Pluto Admin" uri="/WEB-INF/themes/pluto-default-theme.jsp">
        <portlet context="/pluto" name="PlutoPageAdmin"/>
        <portlet context="/pluto" name="AboutPortlet"/>
    </page>
    <page name="Pruebaportlet" uri="/WEB-INF/themes/pluto-default-theme.jsp">
        <portlet context="/j511-primer-portlet" name="Primer Portlet"/>
    </page>
</render-config>

Para obtener Ayuda, presione F1
```

Liferay

Instalación de Liferay

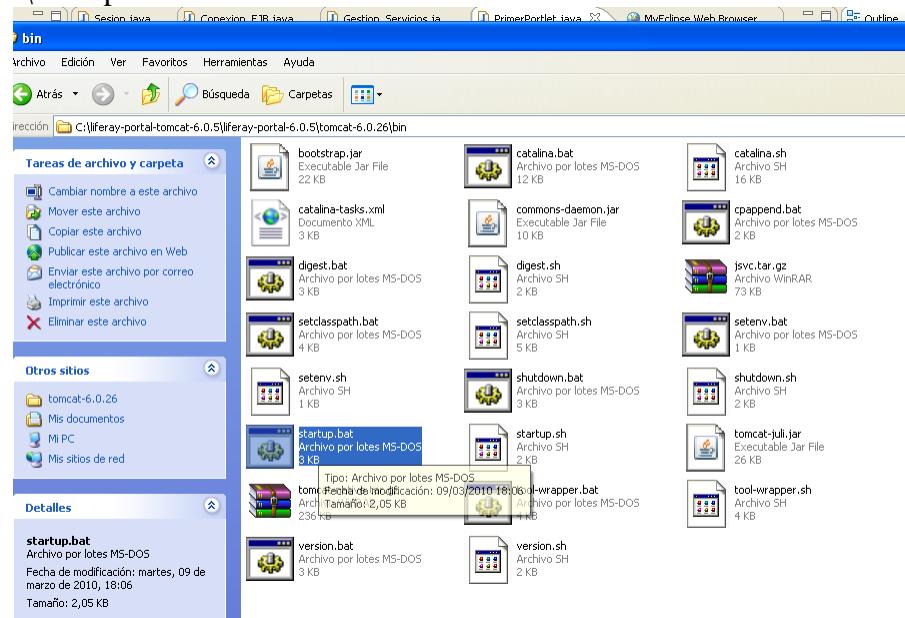
Descargamos Liferay de la web oficial

The screenshot shows the Liferay website's download section. At the top, there's a message in Spanish: "Esta página está escrita en inglés ▾ ¿Quieres traducirla? Traducir No". Below the header, there's a navigation bar with links: Inicio, Productos, Servicios, Partners, Documentación, Comunidad, Descargas, Quiénes somos, and Marketplace. The "Descargas" menu is expanded, showing categories: Liferay Portal, Liferay Social Office, Liferay Sync, and Liferay Projects (Alloy UI, Liferay IDE). A sidebar on the left has a "Try Liferay EE for 30 days FREE!" button. The main content area is titled "Get Liferay Portal" and describes the two editions: "Liferay Portal comes in two versions offering market-leading innovations and features. Learn more about our Community (CE) and Enterprise (EE) Editions to determine which best suits your needs." Below this, there are two download cards. The first card is for "Liferay Portal 6.1 Community Edition" (6.1 GA 1), which is bundled with Tomcat. The second card is for "Liferay Portal 6.0 Community Edition" (6.0 GA 4), also bundled with Tomcat. Both cards have a "Download" button.

La versión 6.1 todavía no es completamente estable. Nosotros usaremos la versión liferay-portal-tomcat-6.0.5.zip

Es recomendable instalarlo en la carpeta raíz del SO, como dicta el manual, ya que algunas rutas son muy largas y pueden perderse por exceder el maximo de windows de rutas de mas de 200 caracteres.

Vamos a la carpeta C:\liferay-portal-tomcat-6.0.5\liferay-portal-6.0.5\tomcat-6.0.26\bin\satrtup



Ejecuta

```
primpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at org.apache.catalina.startup.Bootstrap.load(Bootstrap.java:261)
    at org.apache.catalina.startup.Bootstrap.main(Bootstrap.java:413)
2-feb-2012 12:29:01 org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 4377 ms
2-feb-2012 12:29:02 org.apache.catalina.core.StandardService start
INFO: Arrancando servicio Catalina
2-feb-2012 12:29:02 org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/6.0.26
2-feb-2012 12:29:02 org.apache.catalina.startup.HostConfig deployDescriptor
INFO: Despliegue del descriptor de configuraci n en ROOT.xml
padding jar file: C:/liferay-portal-tomcat-6.0.5/liferay-portal-6.0.5/tomcat-6.0.5/webapps/ROOT/WEB-INF/lib/portal-impl.jar!/system.properties
padding jar file: C:/liferay-portal-tomcat-6.0.5/liferay-portal-6.0.5/tomcat-6.0.5/webapps/ROOT/WEB-INF/lib/portal-impl.jar!/portal.properties
2012-02-02 02:29:06 INFO [DialectDetector:89] Determining dialect for HSQL Database Engine
in 1 seconds
2:29:12.299 WARN [DialectDetector:84] Liferay is configured to use Hypersonic as its database. DO NOT use Hypersonic in production. Hypersonic is an embedded database useful for development and demo'ing purposes. The database settings can be changed in portal.properties.
2:29:12.876 INFO [DialectDetector:49] Using dialect org.hibernate.dialect.HSQL
ialect
```

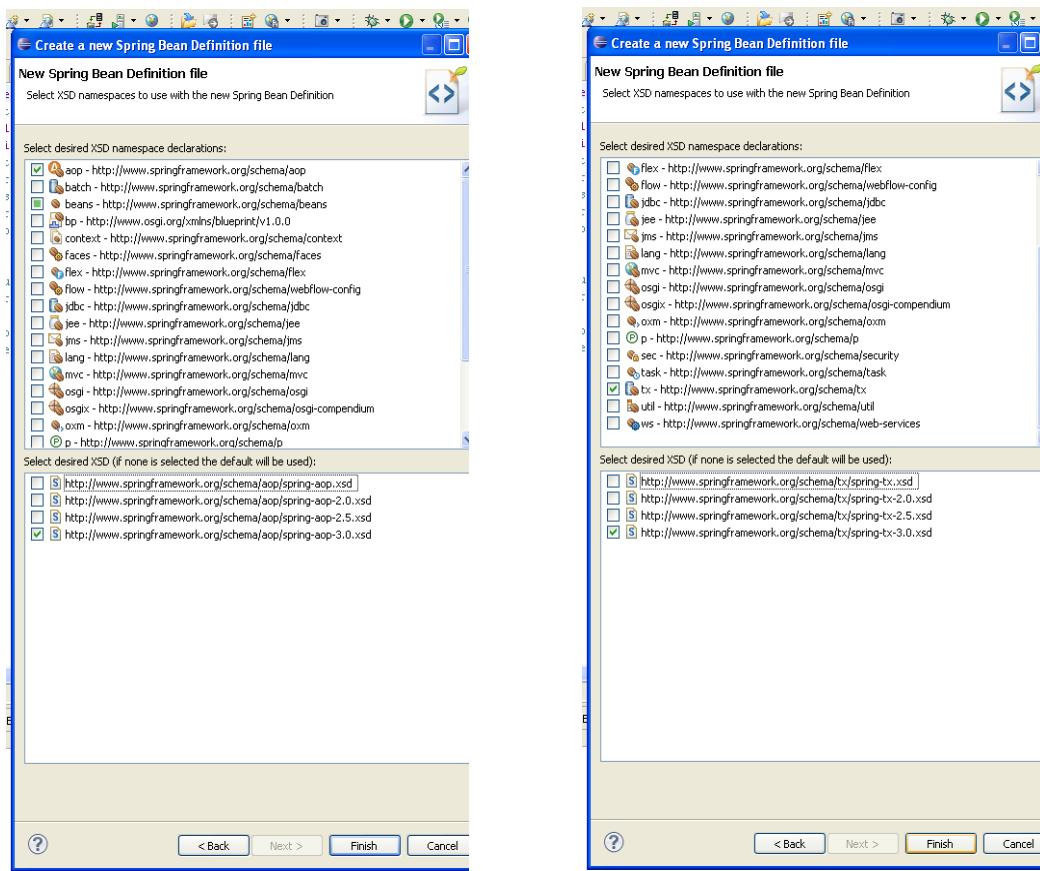
Una vez terminado se abre el navegador y muestra el mensaje de instalacion y arranque exitoso.

Creacion de Portlet con JSF y Iceface

Creamos el proyecto, y sus paquetes. Es importante que en el portlet.xml se declaren los parametros del view, edit y help para que iceface los recoja de la vista. Es decir PONER j511-portlet-pluto-iceface.

En el faces-config.xml especificamos que spring sera el que creara los managedBeans colocando el DelegatingVariableResolver.

Para poder definir menos beans en el applicationContext.xml, creamos un nuevo Spring Bean Definition File con el asistente y le damos nuevos XSD name space declarations en este caso AOP y tx.



Comunicación entre distintos portlets. Eventos

Trabajaremos con la versión 2 de portlets. En la comunicación entre portlets hay un intermediario. Un portlet lanza un mensaje a un portal y el portal se lo pasa a otro portlet.

Portlet.xml

Portlet emisor

```
<supported-publishing-event>
    <Qname xmlns:mio="-----">prefijo:nombreEvento</Qname>
<supported>
```

Al producirse el evento, el portlet va a crear un xml y se lo manda al portal. El QName que se debe usar es el de javax.

P.Receptor

```

<supported-processing-event>
    <Qname xmlns:mio="-----">prefijo:nombreEvento</Qname>
<supported_processing-event>
<event-definitions>
    <Qname>-----</Qname>
    <value-type>java.lang.String</value-type>
</event-definitions>

```

ya solo quedarían las clases, que se harán con anotaciones.

En el metodo que me interese del portlet (lo mas normal es el processAction), lanzaré el evento.

Emitir evento

```

Metodo(){
    QName objEvento = new QName("Qname      ", "nombreEvento");
    ActionResponse.setEvent(objEvento, "informacionEvento");
}

```

`QName` es una clase de java, que crea el fichero xml que se enviará al portal.

Recibir evento

Para indicar que ese metodo será el escuchador del evento colocamos una anotación. Como argumento, le pasamos el nombre o identificador del evento.

```

@processEvent(Qname={"nombreEvento"})
public void receptor(EventRequest eventRequest, EventResponse eventResponse){
    Event obj = eventRequest.getEvent();
    obj.getValue();
}

```

También podemos usar un parámetro compartido para la comunicación entre portlets. Esta opción nos permitiría que una vez comunicado del emisor al receptor el evento, el receptor lance otro evento que capture el emisor

ver ejercicio j511-portlet-pluto-eventos2