UT4 – JEE y Servlets

Plataforma de desarrollo Java, JEE

JEE es una plataforma de programación para desarrollar y ejecutar software de aplicaciones en el lenguaje de programación Java. JEE está formado por varias especificaciones de APIs, que incluyen tecnologías que extienden la funcionalidad de las APIs de base de JSE

APIs aportadas por JEE

Servlets, JSP, JSTL: Se trata de APIs, para la construcción de páginas web dinámicas

Componentes EJB: Componentes para trabajar en sistemas distribuidos con transacciones distribuidas, ahora mismo desplazada esta técnica por Spring

Java Message Service API (JMS) Es el API para crear, enviar, recibir y leer mensajes entre aplicaciones heterogéneas, realizados en distintos lenguajes y plataformas.

Java Transaction API (JTA) Es el API para manejo de transacciones a través de sistemas heterogéneos.

JavaMail API: API para el envío y recepción de correo electrónico.

Java API for XML. Processing JAXP. Para el tratamiento de documentos XML

Java Database Connectivity API (JDBC) Para la ejecución de sentencias SQL desde clases Java

Java Persistence API: Para la relación entre entidades java y tablas de la base de datos.

Java Naming and Directory Interface (JNDI) Para acceso a servicios de nombres y directorios

Java Authentification and Authorization Service (JAAS) Permite la autentificación y autorización a usuarios o grupos de usuarios

Servidor de aplicaciones

Los programas Java se ejecutan sobre un servidor de aplicaciones, que puede manejar transacciones, seguridad, gestión de componentes desplegados, concurrencia...

Servidores de aplicaciones más conocidos

Tomcat: http://tomcat.apache.org/

JBOSS: Existe una versión de código abierto soportado por la comunidad y otra empresarial. Soporta EJB http://www.jboss.org/technology/

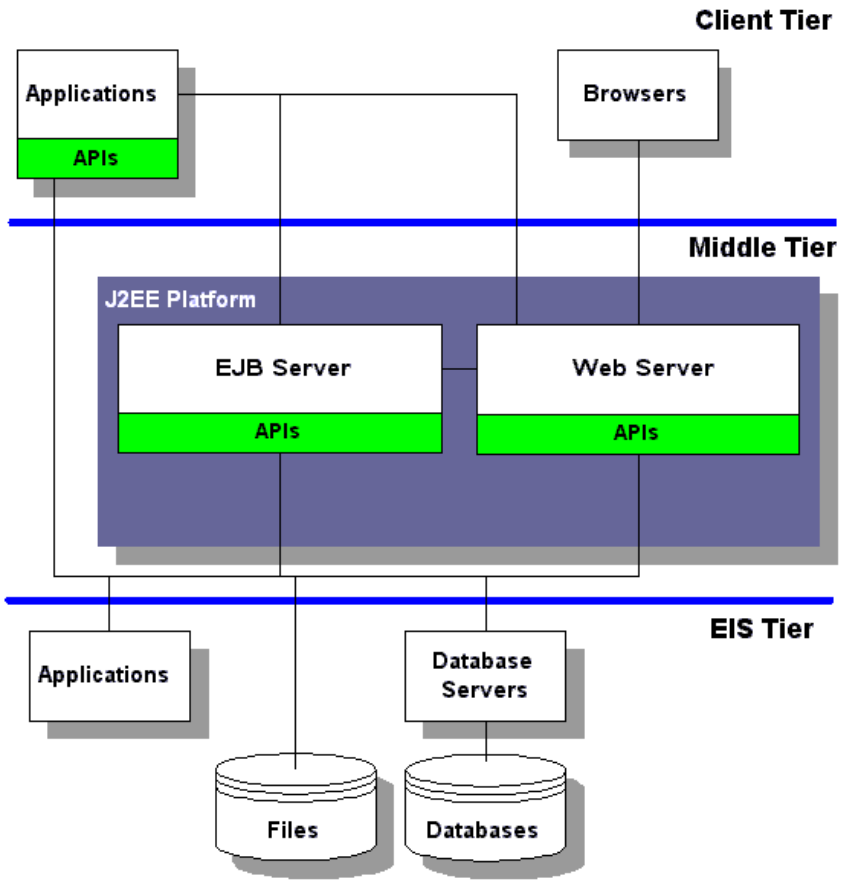
Oracle WebLogic Application:
http://www.oracle.com/technetwork/middleware/weblogic/overview/index-085209.html

Apache Geronimo http://geronimo.apache.org/

IBM WebSphere Application Server http://www.ibm.com/software/websphere

GLASSFISH: servidor de aplicaciones de código abierto de Sun
https://glassfish.java.net/

Jetty: http://www.eclipse.org/jetty/



Empaquetado de aplicaciones Java

Una aplicación Java EE se entrega en un archivo JAR (Java Archive), un archivo de archivos web (WAR), o un archivo EAR (Enterprise Archive).

JAR: Módulos de aplicación de escritorio, que contienen los archivos de clase y, opcionalmente, un descriptor de despliegue del cliente de aplicación, se empaquetan como archivos JAR con una extensión .jar.

WAR Es un archivo comprimido (con la extensión WAR) usado para distribuir una colección de archivos JSP, Servlets, clases Java, archivos XML y contenido web estático (HTML). En conjunto constituyen una aplicación Web.

EAR Este tipo de ficheros son desplegables en servidores de aplicaciones que soporten el stack completo de JEE , es decir, que contengan tanto un Servlet Container como un EJB Container.

Un EAR es capaz de albergar varios ficheros WAR cada uno de los cuales contiene una aplicación web completa. Aparte de albergar varias aplicaciones web también tiene la capacidad de contener EJBs que son empaquetados en archivos JAR para su utilización en las distintas aplicaciones web

Instalación de Tomcat

Descarga el Apache Tomcat del servidor de la web http://tomcat.apache.org/

Para la instalación en Windows simplemente descomprime en un directorio el fichero descargado. Para la instalación de Tomcat en otras plataformas consulta en Google.

Después de la instalación prueba a abrir un navegador http://localhost:8080/. Esto debería abrir la página principal de Tomcat, lo cual significa que Tomcat está bien instalado.

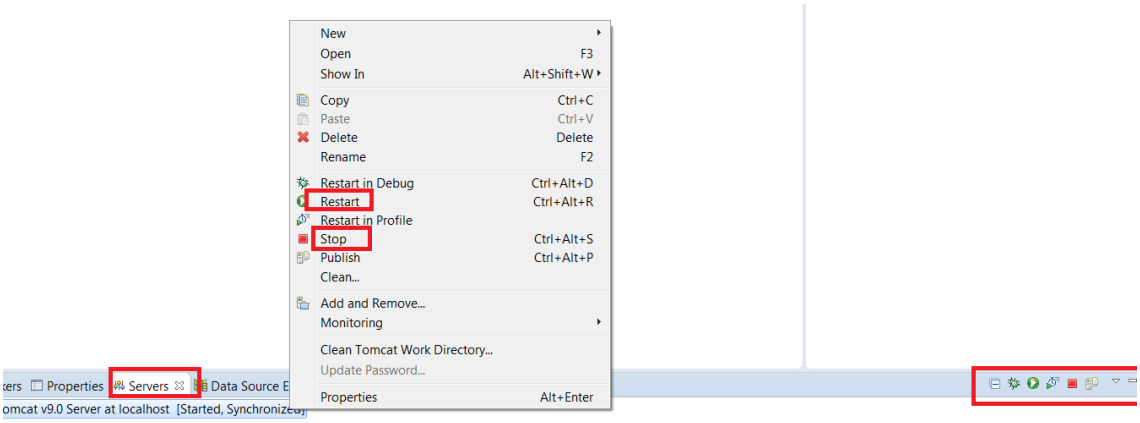
Despúes de verificar que Tomcat está instalado correctamente, páralo. Eclipse tratará de iniciar Tomcat por sí mismo.

Configuración de Eclipse: Selecciona Windows->Preferences->Server->Runtime Environment -> Add -> Selecciona la versión del Tomcat

Para compilar las páginas JSP en servlets es necesario utilizar el JDK. Normalmente viene seleccionado por defecto, en caso contrario pulsar sobre Installed JRE y seleccionar. Pulsar finalizar y Aceptar.

Servidor

Durante el desarrollo se va a crear un servidor. Podemos administrar el servidor a través de la vista Servers de Eclipse. Windows->Show View->Servers, desde donde podremos iniciar, detener y reiniciar el servidor



Qué es un servlet

Un servlet es un programa o una clase java que se ejecuta en un contenedor web de un servidor de aplicaciones y que sirve para crear aplicaciones web en Java. Para crear un servlet que responda a peticiones HTTP, solamente tenemos que crear un clase que herede de HttpServlet. Los Servlets siempre deben de implementar la interface javax.servlet.Servlet o heredar de la clase javax.servlet.http.HttpServlet.

API de Servlet

La clase javax.servlet.Servlet es la interfaz base del api de servlets, pero existen otras interfaces y clases que debemos tener en cuenta para trabajar con servlets.

Cómo se usa

Los clientes pueden invocarlo mediante el protocolo HTTP, por ejemplo a través de un enlace en una página web (Enlace al servlet), o bien a través de un formulario HTML, colocando el nombre del Servlet en la cláusula action del formulario.

El Servlet de este modo, acepta peticiones de un cliente, procesa la información relativa a la petición realizada por el cliente y le devuelve los resultados que podrán ser mostrados mediante páginas HTML, JSP, XSLT...

También pueden utilizarse para otras tareas como comunicarse con otros Servlets, acceder a bases de datos, gestionar la comunicación entre varias páginas html, jsp, realizar cualquier lógica de aplicación...

Ciclo de vida del servlet

Un Java Servlet tiene definido un ciclo de vida, forzado por la interface javax.servlet.Servlet, por tanto, siempre que queramos implementar un Java Servlet, nuestra clase deberá implementar dicho interface. Es una especie de contrato entre el Java Servlet y el contenedor web del Servidor de Aplicaciones Java EE para poder funcionar. El ciclo de vida viene determinado por los siguientes métodos, que son los que contienen un Servlet básico.

- init: se ejecuta una vez en la vida del Java Servlet y se suele utilizar para llevar a cabo todas las labores de inicialización que necesitemos para su posterior ejecución.
- service: se ejecuta cada vez que se recibe una petición para el Java Servlet. En su implementación, se coloca la lógica para la que se creó. Si se ha heredado de la clase javax.servlet.http.HttpServlet, se utilizan los métodos doGet o doPost.
- destroy: se ejecuta una vez en la vida del Java Servlet y se suele utilizar para liberar todos aquellos recursos que pudiéramos haber utilizado, con el afán de dejar el sistema lo más limpio posible.
- Los objetos de los tipos javax.servlet.http.HttpServletRequest y javax.servlet.http.HttpServletResponse. que reciben los métodos doGet y doPost, se utilizan para identificar la petición realizada y la respuesta dada. Contienen métodos para obtener de la petición parámetros, atributos, cabeceras y gestionar la respuesta del Servlet al navegador que ha hecho la petición.

Estados del Servlet

Dicho ciclo de vida tiene los siguientes estados:

- Cuando se recibe una petición de la ejecución de un Java Servlet en el Servidor de Aplicaciones Java EE y este no ha sido invocado nunca, el contenedor web instancia la clase del Java Servlet (llama a su constructor) y acto seguido invoca el método: public void init(ServletConfig config); Dicho método, solo se ejecuta una vez en la vida del Java Servlet y se suele utilizar para llevar a cabo todas las labores de inicialización que necesitemos para su posterior ejecución. Como se puede observar, recibe como parámetro una instancia de la clase ServletConfig, que representa información de configuración que el administrador del Servidor de Aplicaciones Java EE haya podido añadir. Veremos este método en más detalle en los siguientes apartados.

- Una vez está ya inicializado, entonces invoca el método: public void doPost(ServletRequest request, ServletResponse response) o el doGet(); Dicho método se ejecuta cada vez que se recibe una petición para este java Servlet. En su implementación, colocaremos la lógica para la que se creó dicho Java Servlet. Como se puede observar, recibe como parámetros un par de instancias de las clases ServletRequest y ServletResponse. Ambos dos parámetros representan la petición (la usaremos para entender qué nos están pidiendo) y la respuesta (la utilizaremos para devolver el resultado de la ejecución del java Servlet al peticionario).

- Si por alguna razón el contenedor web necesita eliminar al Java Servlet (se detiene el Servidor de Aplicaciones Java EE, se queda sin recursos de memoria y necesita liberar espacio, etc…), entonces invoca el método: public void destroy(); Dicho método sólo se ejecuta una vez en la vida del Java Servlet y se suele utilizar para liberar todos aquellos recursos que pudiéramos estar utilizando, para dejar el sistema lo más limpio posible.

Ejemplo 1

Desarrollo con Eclipse

Desde Eclipse es posible configurar una instancia del servidor Tomcat (o de cualquier otro servidor de aplicaciones), para probar la aplicación cuando estamos desarrollándola. Así que no tenemos que generar el fichero War y desplegarlo en el servidor Tomcat real para poder probar la aplicación.

Cuando ejecutamos un componente de la aplicación web, Eclipse hace las siguientes tareas de forma automática:

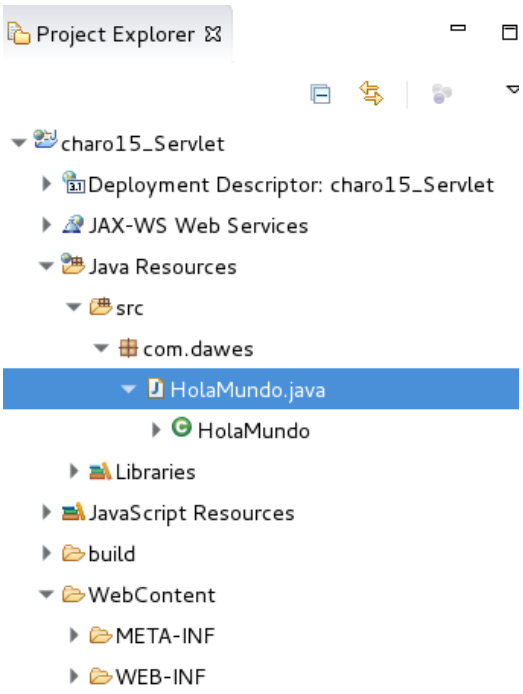
- Compila las clases
- Lanza una instancia del Tomcat (una copia del servidor original)
- Publica la aplicación en una carpeta temporal
- Abre un panel con un navegador
- Muestra el componente en el navegador

Primera aplicación web
Desde el menú del Eclipse for Java EE Developers: File->New->Web->Dynamic Web Project

En la carpeta del proyecto botón derecho->New->Servlet->Finish

En target Runtime, pulsamos el botón New Runtime, y le damos el directorio de instalación de Tomcat. Además activamos la pestaña create new server

Esto crea una estructura de directorios tal como la siguiente



Esto nos crea un servlet con un constructor y los métodos doGet() y doPost()

Si creamos el servlet como una clase normal, o si hacemos copy/paste de otro Servlet, el servidor Tomcat no encontrará el Servlet, ya que no actualizará el fichero web.xml.

Vamos a ver en qué partes se descompone un servlet, ciclo de vida y cómo pasar parámetros a un servlet desde un formulario HTML

Creamos el servlet cuyo código fuente es el siguiente:

```
package com.dawes;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/HolaMundo")
```

```
public class HolaMundo extends HttpServlet {
        private static final long serialVersionUID = 1L;

   public HolaMundo() {
      super();
   }

    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
                response.setContentType("text/html");
                PrintWriter out=response.getWriter();
                out.println("Hola Mundo");
        }

    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
                doGet(request, response);
        }

}
```

Comentamos un poco este primer servlet. Lo primero que nos aparece son las clases que utiliza javax.servlet.*, javax.servlet.http.* y java.io.*

Lo segundo es una annotatation @WebServlet que indica al contenedor web Tomcat que esta clase es un Servlet que tiene que atender a las peticiones de los clientes

Más información sobre anotaciones en Tomcat:

http://docs.oracle.com/javaee/6/api/javax/servlet/annotation/package-summary.html

Lo siguiente que vemos es la firma de la clase, que hereda de la clase HttpServlet.

Lo siguiente es redefinir el método doGet(), que recibe dos parámetros. El primero, de la clase HttpServletRequest, representa la petición del cliente y el segundo HttpServletResponse, representa la respuesta del servidor.

Como en este primer ejemplo no necesitamos ninguna información del cliente, no usaremos para nada el parámetros HttpServletRequest.

De la clase HttpServletResponse usamos dos métodos:

- void setContentType(java.lang.String type): nos sirve para especificar el tipo de contenido que vamos a enviar en la respuesta. Por ejemplo: para indicar que se trata de una página web, usamos el tipo text/html,
  Otras posibilidades son: text/plain, image/gif, etc… Los distintos tipos de contenido están estandarizados a través de las especificaciones MIME (Multipurpose Internet Mail Extensions). En la siguiente URL están documentados todos los posibles tipos: http://www.iana.org/assignments/media-types/index.html Esta información viaja en la cabecera de una respuesta HTTP.

```
response.setContentType("text/html");
```

- java.io.PrintWriter getWriter() throws java.io.IOException: con el que obtenemos una clase PrintWriter en donde iremos escribiendo los datos que queremos que el cliente reciba

```
PrintWriter out=response.getWriter();
```

Una vez que hemos establecido el tipo de respuesta (text/html) y tenemos el flujo de salida (variable out) solo nos queda utilizar el método println de la clase PrintWriter para ir escribiendo en dicho flujo de salida la página HTML que queremos que visualice el cliente

```
out.println("Hola Mundo");
```

Ejecutar el Servlet:botón derecho sobre el proyecto->Run as...->Run on server

- Nos aparece una ventana para elegir el servidor de aplicaciones que vamos a utilizar (elegimos Tomcat 8)
- Seguidamente se abre una ventana con un navegador y veremos el resultado de la ejecución del método doGet() del servlet

Hemos ejecutado un servlet de forma independiente, invocándolo directamente desde el navegador (http://localhost/8080/nombreProyecto/HolaMundo)

Ejemplo 2

Normalmente los servlets tendrán parámetros o fuentes de información que le darán un aspecto dinámico. Es decir, para generar una simple página HTML no nos complicamos tanto la vida, se escribe la página HTML y se ha terminado. Las fuentes de información de las que un servlet hace uso pueden ser varias: el propio servlet, el servidor web, bases de datos o ficheros o parámetros que le pase el cliente. Las más interesantes son los accesos a bases de datos y los parámetros que nos pasa el cliente mediante formularios HTML.

Cuando pasamos parámetros a través de un formulario, en los servlets a través de la clase HttpServletRequest, disponemos de los siguientes métodos para su tratamiento:

String getParameter(String nombre): Nos devuelve el valor del parámetro cuyo nombre le indicamos. Si la variable no existiera nos devuelve null

Enumerate getParameterNames(): nos devuelve una enumeración de los nombres de los parámetros que recibe el servlet

Enumerate getParameterValues(String): Nos devuelve los valores que toma un parámetro dado, esto es útil para listas de selección múltiple donde un parámetro tiene más de un valor.

Veamos un ejemplo de un pequeño formulario que tenga distintos tipos de parámetros, se los envie a nuestro servlet y éste los muestre por pantalla.

Creamos un fichero formulario.html dentro de la carpeta WebContent del proyecto con el siguiente código

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
```

```
<title>Formulario de ejemplo</title>
<body>
	<h1>Formulario</h1>
	<form method="POST" action="Formulario"">
		Nombre: <INPUT TYPE="TEXT" NAME="nombre"><BR> Primer
		Apellido:<INPUT TYPE="TEXT" NAME="apellido1"><BR> Segundo
		Apellido:<INPUT TYPE="TEXT" NAME="apellido2"><BR>
		<hr>
		Correo electronico: <INPUT TYPE="TEXT" NAME="email"><BR>
		Clave: <INPUT TYPE="PASSWORD" NAME="clave"><BR>
		<hr>
		Comentario:
		<TEXTAREA NAME="comenta" ROWS=3 COLS=40>
		  </TEXTAREA>
		<BR>
		<hr>
		Sexo:<BR> <INPUT TYPE="RADIO" NAME="sexo"
VALUE="hombre">Hombre<BR>
		<INPUT TYPE="RADIO" NAME="sexo" VALUE="mujer">Mujer<BR>

		Areas de interés:<br> <SELECT NAME="intereses" MULTIPLE>
			<OPTION>Informatica</OPTION>
			<OPTION>Derecho</OPTION>
			<OPTION>Matematicas</OPTION>
			<OPTION>Fisica</OPTION>
			<OPTION>Musica</OPTION>
		</SELECT>

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

Veamos en primer lugar un servlet llamado Formulario, que conociendo de antemano los distintos parámetros que va a recibir vaya mostrandolos en una página HTML.

```
import java.io.IOException;
import java.io.PrintWriter;
```

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/Formulario")
public class Formulario extends HttpServlet {
	private static final long serialVersionUID = 1L;

   public Formulario() {
      super();
   }

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

		  out.println("<html>");
		  out.println("<body>");
		  out.println("<h1>Parámetros del servlet desde un formulario
HTML</h1>");
		  out.println("<br> Nombre:"+request.getParameter("nombre") );
		  out.println("<br> Primer
apellido:"+request.getParameter("apellido1") );
		  out.println("<br> Segundo
apellido:"+request.getParameter("apellido2") );
		  out.println("<br> Correo
electrónico:"+request.getParameter("email") );
		  out.println("<br> Contraseña:"+request.getParameter("clave") );
		  out.println("<br> Comentario:"+request.getParameter("comenta")
);
		  out.println("<br> Sexo:"+request.getParameter("sexo") );
		  out.println("<br> Areas de
interés:"+request.getParameter("intereses") );
		  out.println("</body>");
		  out.println("</html>");

	}

	protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
		// TODO Auto-generated method stub
```

```
            doGet(request, response);
        }

}
```

Si en la lista de selección múltiple escogemos más de un valor en la solución actual mostraría únicamente la primera elección. Si quisiéramos mostrar todos los valores deberíamos usar getParameterValues("intereses") e ir recorriendo y mostrando cada uno de los valores seleccionados del parámetros intereses.

Botón derecho sobre el formulario.html en el explorador de paquetes de Eclipse-> run as ->run on server

Despliegue de la aplicación

Para poder desplegar una aplicación web, se puede realizar mediante un EAR o un WAR (dependiendo del servidor de aplicaciones). El descriptor de despliegue web.xml, aunque no es obligatorio si se utilizan anotaciones, aunque conviene que exista para poder determinar de una manera sencilla que es lo que se va a desplegar y el comportamiento asociado.

Una vez creado el servlet, se compila y se despliega, junto con el resto de componentes (páginas web, imágenes, jsp, ...) exportándolo como un fichero WAR o un EAR. Una aplicación Java EE está formada por un empaquetamiento de una o varias unidades conocidas con el nombre de módulos. Uno de los distintos tipos de módulos mencionados son los módulos Web, que contienen normalmente Java Servlets, JavaServer Pages (JSP), JavaServer Faces (JSF), contenidos estáticos como imágenes, HTMLs, CSSs… Su extensión del fichero empaquetado es WAR (Web ARchive).
Normalmente no nos preocuparemos mucho de la generación de un módulo Web u otro tipo de módulo, porque son los IDEs los que nos van a ayudar a generarlo. Por ejemplo, en nuestro caso, con Eclipse IDE for Java EE Developers basta con seleccionar el proyecto, pulsar el botón derecho del ratón y seleccionar las opciones "Export" -> "WAR file":
Este fichero WAR, sería el que un administrador de Servidores de Aplicación Java EE desplegaría y configuraría para que la aplicación estuviese disponible y dando servicio. Pero en nuestro caso, simplemente añadiremos o quitaremos aplicaciones

en el Apache Tomcat de Eclipse IDE for Java EE Developers mediante el interfaz de usuario: botón derecho del ratón y la opción "Add and Remove…", o directamente sobre el componente y "Run As" -> "Run on Server".
Una vez desplegada y arrancado el servidor Tomcat, abrimos el navegador y tecleamos la url del servlet: http://localhost:8080/charo15_Servlet/HolaMundo
Esto envía una petición HTTP get al Tomcat, que busca el Servlet en el archivo de despliegue (web.xml) y si lo encuentra, el servidor ejecutará el método doGet() del Servlet

Peticiones HTTP

Para acceder a una página web a través de un navegador (http://localhost:8080/recurso.html) se envía una petición HTTP al servidor en el que está alojada la página. Esta petición, en caso de que contenga un formulario, puede enviar información al servidor. La petición puede ser de dos formas: petición de tipo get o de tipo post.

Petición de tipo get

Cuando el tipo de petición HTTP es GET, los datos del formulario son enviados al servidor a continuación de la dirección URL especificada por el atributo action del formulario; por ejemplo:

http://localhost:8080/mantCuenta?numCuenta=1234&fecha=2015-10-14

En la URL podemos ver los parámetros que se envian al servidor. En este caso el numero de la cuenta y una fecha. Este tipo de peticiones provoca varios problemas:

1.- El tamaño de la cadena está limitado a 255 caracteres

2.- El tipo de codificación ASCII es legible y puede ser utilizada con intenciones fraudulentas

Petición de tipo post

Para poder utilizar el post, se utilizan los formularios de HTML. Los datos son enviados al servidor en el cuerpo de la petición. Es el método usual de enviar los datos de un formulario. La información del formulario se envía después del URL indicado por action dentro del cuerpo del la petición, por lo que no es accesible a simple vista. El tamaño de la información enviada no está limitado.

Por tanto, podemos utilizar la petición get si los datos son pocos y no importa su confidencialidad, pero si son largos, privados o importantes, utilizaremos post.

Atributos de Servlet

Debido a que HTTP no tiene estado, para asociar una solicitud a cualquier otra solicitud, necesita una forma de almacenar datos de usuario entre solicitudes HTTP.

Las cookies o los parámetros de URL (por ejemplo, como http://example.com/insertar?nombre=pepe&apellido=perez) son formas adecuadas de transportar datos entre 2 o más solicitudes. Sin embargo, no son buenos en caso de que no desee que los datos se puedan leer o editar en el lado del cliente.

La solución es almacenar ese lado del servidor de datos, darle una "identificación" y dejar que el cliente solo sepa (y devolver en cada solicitud http) esa identificación, es decir, sesiones implementadas. O puede usar el cliente como un almacenamiento remoto conveniente, pero cifraría los datos y mantendría el secreto del lado del servidor.

Los atributos de servlet se utilizan para la comunicación entre servlets y clases java. Podemos establecer, obtener y eliminar los atributos en la aplicación web. Hay tres ámbitos para los atributos: ámbito de petición, ámbito de sesión y ámbito de aplicación.

El objeto HttpServletRequest

La interfaz ServletRequest se utiliza para proporcionar información de solicitud del cliente al servlet. El contenedor de servlets crea el objeto ServletRequest desde la solicitud del cliente y lo pasa al método service() para su procesamiento.

Algunos de los métodos importantes de la interfaz ServletRequest son:

Object getAttribute(String name): este método devuelve el valor del atributo name y null si no existe. Podemos utilizar getAttributeNames() para obtener la enumeración de los nombres de los atributos de la petición.

String getParameter(String name): este método devuelve el parámetro de la petición como cadena. Podemos utilizar getParameterNames() para obtener la enumeración de los nombres de los parámetros de la petición.

La interfaz HttpServletRequest contiene también métodos para la gestión de la sesión, las cookies.

El objeto HttpServletResponse

Esta interfaz es utilizada por el servlet para enviar la respuesta al cliente. El contenedor de servlets, crea el objeto ServletResponse y lo pasa al método service() y posteriormente utiliza el objeto response para generar la respuesta HTML para el cliente.

Algunos de los métodos importantes son:

addCookie: se utiliza para añadir una cookie a la respuesta

addHeader (String name, String value): se utiliza para agregar un encabezado de respuesta con el nombre y el valor dado.

getHeader(String name):devuelve el valor para el encabezado especificado o null si esta cabecera no se ha establecido.

sendRedirect(url): se utiliza para enviar una respuesta de redirección temporal para el cliente, utilizando la dirección url.

El objeto HttpSession

Uno de los conceptos que más problemas produce cuando comenzamos a trabajar con aplicaciones web en Java es el concepto de java session (HttpSession) que sirve para almacenar información entre diferentes peticiones HTTP ya que este protocolo es stateless (sin estado). Así pues en muchas ocasiones nos encontraremos con el problema de compartir estado (datos usuario) entre un conjunto amplio de páginas de nuestra Aplicación.

Para solventar este problema en la plataforma Java EE se usa de forma muy habitual la clase HttpSession que tiene una estructura de HashMap (Dicccionario) y permite almacenar cualquier tipo de objeto en ella de tal forma que pueda ser compartido por las diferentes páginas que como usuarios utilizamos.

El funcionamiento del sistema de sesiones es relativamente sencillo. Cada vez que un usuario crea una session accediendo a una página (que la genere) se crea un objeto a nivel de Servidor con un HashMap vacío que nos permite almacenar la información que necesitamos relativa a este usuario. Realizado este primer paso se envía al navegador del usuario una Cookie que sirve para identificarle y asociarle el

HashMap que se acaba de construir para que pueda almacenar información en él. Este HashMap puede ser accedido desde cualquier otra página permitiéndonos compartir información.

El concepto de Session es individual de cada usuario que se conecta a nuestra aplicación y la información no es compartida entre ellos. Así pues cada usuario dispondrá de su propio HashMap en donde almacenar la información que resulte útil entre páginas.

Estudiaremos este concepto en el siguiente tema Seguimiento de la Sesión

ServletContext

El objeto ServletContext permite acceder a un servlet a la información de su contexto, es decir, la información asociada con la propia aplicación y que es común a todos los servlets que despleguemos dentro de esa aplicación. Por eso, el nombre adecuado quizás sería ApplicationContext….

Los desarrolladores del contenedor web, generan algún objeto que aporta información a nivel de api sobre la propia aplicación web. El objeto ServletContext, que pertenece al standard de JEE y por lo tanto todos los servidores lo incorporan accede a parte de la información de este otro objeto.

Ejemplo: leer una variable global de la aplicación web y que está declarada en el fichero web.xml

```
        <context-param>
        <param-name>carpeta</param-name>
        <param-value>imagenes</param-value>
        </context-param>
</web-app>
```

Como podemos ver hemos declarado una variable denominada carpeta, que es común para toda la aplicación y queremos compartirla entre los distintos servlets. Para ello utilizamos el objeto ServletContext y el método getInitParameter()

```
        protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
                response.setContentType("text/html");
                PrintWriter out=response.getWriter();
                ServletContext contexto=request.getServletContext();
                 String directorio=contexto.getInitParameter("carpeta");
                 out.println(directorio);
```

Algunos métodos importantes de ServletContext son:

Object getAttribute (String name): devuelve el atributo de objeto para el nombre dato. Podemos obtener la enumeración de todos los atributos suando getAttributeNames()

setAttribute(String name, Object value): este método se utiliza para establecer el atributo con el ámbito de aplicación. El atributo será accesible a todos los demás servlets que tengan acceso a este ServletContext. Podemos eliminar un atributo utilizando el método removeAttribute(String name)

getInitParameter(String name): este método devuelve el valor de cadena para el parámetro init definido con el nombre en web.xml. Devuelve null si el parámetro no existe. Podemos utilizar el método getInitParemeterNames() para obtener la enumeración de todos los nombres de los parámetros de inicio.

ServletContext tiene otros métodos que pueden resultar interesantes. Echales un vistazo

https://docs.oracle.com/javaee/6/api/javax/servlet/ServletContext.html

ServletConfig

Es un objeto que sirve para pasar información de configuración al servlet, que se declaran los parámetros en el web.xml, pero dentro de la etiqueta servlet. Cada servlet tiene su propio objeto ServletConfig y el contenedor de servlets es responsable de instanciar este objeto. Podemos proporcionar parámetros de inicialización del servlet en web.xml o mediante el uso de la anotación WebInitParam. Podemos utilizar el método getServletConfig() para obtener el objeto ServletConfig del servlet

```
<servlet>
                <servlet-name>Contador</servlet-name>
                <servlet-class>com.dawes.Contador</servlet-class>
    <init-param>
                        <param-name>correo</param-name>
                        <param-value>laboral@gmail.com</param-value>
          </init-param>

        </servlet>

</web-app>
```

Así para recuperar los valores del parámetro desde el servlet Contador haremos

String parametro=getInitParameter("correo");

Transferir el control a otro componente

En la mayoría de aplicaciones necesitaremos que un componente Web (servlet) simplemente procese las peticiones y transfiera el control a otros componentes (JSPs) para mostrar los resultados.

Para llamar a otro componente desde un Servlet podemos utilizar dos métodos:

El método forward() de RequestDispatcher.

```
RequestDispatcher dispatcher= request.getRequestDispatcher("/otrapagina.jsp");
if(dispatcher!= null)
        dispatcher.forward(request,response);
```

El método sendRedirect() de HttpResponse.

```
response.sendRedirect("/MiWeb/mipagina.jsp");
```

Forward es más rápido y además envía al jsp el mismo objeto request que recibe el servlet. Sin embargo, no se puede ir a direcciones externas a nuestro servidor (http://.....)

Pool de conexiones.

El pool de conexiones es una técnica usada en aplicaciones Web para mejorar el rendimiento de las aplicaciones Web.

Antes de ver en qué consiste , veamos cómo funciona la creación de conexiones en una aplicación clásica de escritorio cliente-servidor y luego veamos los problemas de seguir con dicha estructura en una aplicación web.

Problemas en la creación de conexiones

En una aplicación de escritorio se crea una conexión de base de datos al iniciar la aplicación y se cierra al finalizar la aplicación. Es decir que cada usuario que inicia la aplicación tiene una conexión en exclusiva para él. Y obviamente sería imposible compartirlas ya que cada aplicación estará en un ordenador independiente.

En una aplicación Web podríamos seguir un esquema similar , en el que cada usuario nuevo que se conecta a nuestra aplicación se le crea una conexión y al salir de la aplicación que se cierre su conexión.

Si siguiéramos el mismo patrón de creación de conexiones de aplicaciones de escritorio en aplicaciones web acabaríamos con una cantidad enorme de conexiones activas (debido al gran número de usuarios) y con gran cantidad de conexiones abiertas sin usar (debido a usuarios que abandonan el portal y no lo hemos detectado)

Consecuencia de los anterior al tener tantas conexiones a la base de datos se acabaría cayendo el servidor de base de datos debido a los recursos consumidos por todas las conexiones.

Podemos pensar que nuestro servidor puede aguantar todas esas conexiones ya que podemos tener pocos usuarios pero no suele ser así debido a:

- Si no se cierran las conexiones aun teniendo pocos usuarios es probable que acabemos saturando al servidor de conexiones sin usar.
- Las aplicaciones Web pueden tener picos de mucho tráfico donde sería normal que se excediera la capacidad de nuestro servidor.
- Sería muy sencillo hacernos un ataque de denegación de servicio y saturando el servidor haciendo que se crearan gran cantidad de conexiones que no se usen.

Una solución **ineficaz** que nos evitaría las conexiones sin usar sería que se creara la conexión al iniciar cada petición web y se cerrara al finalizar dicha petición web. ¿El problema de eso? Crear y cerrar una conexión es muy costoso. Lo que tendríamos es una aplicación lentísima.

La solución del pool de conexiones tiene que solucionar los siguientes problemas:

- No tener tantas conexiones como usuarios ya que el nº de usuarios es demasiado elevado.
- Cerrar la conexión.

El pool de conexiones consiste en tener un conjunto (pool) de conexiones ya conectadas a la base de datos que puedan ser reutilizadas entre distintas peticiones.

Veamos como funciona

- El servidor web tiene "n" conexiones ya creadas y conectadas a la base de datos (Se llaman conexiones esperando)
  - Cuando llegan "m" peticiones web , la aplicación pide "m" conexiones al pool de conexiones quedando esperando en el pool "n-m" conexiones. Esta operación es muy rápida ya que la conexión ya está creada y solo

hay que marcarla como que alguien la está usando. Ahora hay "m" conexiones activas que está usando la aplicación.

- Cuando las peticiones web finalizan, las conexiones no se cierran sino que se devuelven al pool indicándole que ya se han acabado de usar las conexiones. Ahora vuelve a quedar "n" conexiones esperando en el pool. Esta operación también es muy rápida ya que realmente no se cierra ninguna conexión sino que simplemente se marcan como que ya no las están usando nadie.

- Si se piden más conexiones de las que hay esperando en el pool se crearán en ese instante nuevas conexiones hasta el máximo de conexiones que permita el pool
- Al devolver una conexión al pool , ésta se queda esperando para que otra petición la pueda usar. Si hay ya demasiadas conexiones esperando a ser usadas se cerrarán para ahorrar recursos en el servidor de base de datos.

¿Que hemos conseguido con el pool?

- Ahora las conexiones ya no se quedarán abiertas cuando el usuario se marcha del portal ya que cada conexión se *pseudo-abre* y *pseudo-cierra* con cada petición.
- No tenemos tantas conexiones como usuarios usan la aplicación ya que solo se necesitan tantas como usuarios hay haciendo una petición en ese instante. Pensemos por un momento en facebook. ¿cuandos usuarios están conectados a facebook? Supongamos "x". Pero ¿cuantos están realmente haciendo una petición y no viendo los datos que se han servido? Supongamos "y". Obviamente "y" es mucho menor que "x". Con lo que nos hemos ahorrado "x-y" conexiones.

Configuración del pool de conexiones JDBC

Convertir el proyecto dinámico web en un proyecto maven. Para ello, desde el explorador de proyectos de eclipse, botón derecho sobre el proyecto->configure->convert to maven project

Añadimos al pom.xml la dependencia del driver de MySql, como ya sabemos hacer.

El primer paso es editar el archivo context.xml

Desde Eclipse: Project Explorer-> Servers-> Tomcat8...-> context.xml y añadir el datasource para después poder recuperarlo desde java con JNDI.

```
  <Resource name="ConexionMySql" auth="Container"
type="javax.sql.DataSource"
      maxActive="20" maxIdle="10" maxWait="5000"
      username="root" password="temporal"
driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost/BD_dawes"/>
```

Vamos a ver qué significan los parámetros que hemos usado, si quieres ver todos los posibles aqui tienes la lista completa en ingles.

- maxActive: Indica el número máximo de conexiones que pueden estar abiertas al mismo tiempo.
- maxIdle: El numero máximo de conexiones inactivas que permanecerán abiertas, si el numero de conexiones inactivas es muy bajo puede darse el caso de que las conexiones se cierren porque se llega al máximo de conexiones inactivas y se vuelvan a abrir inmediatamente reduciendo la eficiencia ya que se perdería la ventaja del uso de pool de conexiones en cuanto a que no hay que abrir una conexión cada vez que es necesario.
- maxWait: Es el tiempo máximo (en ms) que se esperará a que haya una conexión disponible (inactiva), si se supera este tiempo se lanza una excepción.

1. Ahora hay que añadir el recurso creado en el descriptor de despliegue de nuestra aplicación. Desde Eclipse: Project Explorer-> Servers-> Tomcat8...-> web.xml Añade la etiqueta <Resource-ref> con los detalles de la conexión a la base de datos dentro de la etiqueta <Web-app>

```
 <resource-ref>
  <description>Pool conexiones MySql</description>
  <res-ref-name>ConexionMySql</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
 </resource-ref>
```

Con estos pasos ya tenemos el pool listo para poder ser usado en nuestras aplicación ya solo queda incluir la librería del driver JDBC correspondiente en ${catalina.home}/lib/ o incluirla en la aplicación, en la documentación pone que hay que incluirla en la carpeta del Tomcat pero si la pones en la aplicación también funciona.

La conexión ahora se obtiene desde el DataSource que obtenemos haciendo lookup sobre InitialContext pasándole el nombre del DataSource que hemos creado, precedido de "java:/comp/env/".

Y una vez obtenida la conexión todo es igual Y finalmente devolver la conexión al pool para que pueda ser reutilizada, cerrando todos los objetos: resultset, statement, connection

```
public class Pool extends HttpServlet {
	private static final long serialVersionUID = 1L;
	Connection c;
	Statement st;
	Context initContext;
	ResultSet rs;
  /**
   * @see HttpServlet#HttpServlet()
   */
  public Pool() {
     super();
     // TODO Auto-generated constructor stub
  }
  public void conectar(){
	try {
			initContext=new InitialContext();
			DataSource ds=(DataSource)
	initContext.lookup("java:/comp/env/ConexionMySql");
			c=ds.getConnection();
			st=c.createStatement();
			rs=st.executeQuery("select * from ALUMNO");
			while (rs.next()){
				System.out.println(rs.getString(1));
				System.out.println(rs.getString(2));
			}
			System.out.println("Conexion OK");
		} catch (NamingException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (SQLException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
```

```
    }
        /**
         * @
```

Para utilizar el pool de conexiones con Hibernate podemos hacerlo de dos formas:

1.- Utilizando el pool del contenedor web

2.- Utilizando una aplicación, por ejemplo c3p0

La única diferencia es que en el primer caso podremos controlar la carga de trabajo del pool utilizando los interfaces estandar del contenedor (consola JBoss o Tomcat). De este modo, el administrador del sistema puede tomar decisiones para aumentar el tamaño del pool. La desventaja es que configurar JNDI es algo más engorroso.

Configurar el pool de conexiones con Hibernate por contenedor web

1. Abrir el fichero *server.xml* en Eclipse

2. Añade la etiqueta <Resource> con los detalles de la conexión a la base de datos dentro de la etiqueta <GlobalNamingResources>

```
<Resource name="jdbc/mydb"
        global="jdbc/mydb"
        auth="Container"
        type="javax.sql.DataSource"
        driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/mydb"
        username="root"
        password="temporal"
        maxActive="10"
        maxIdle="10"
        minIdle="5"
        maxWait="10000"/>
```

3. Guarda el fichero *server.xml*
4. Abre el fichero *context.xml* en Eclipse
5. Añade la siguiente etiqueta <ResourceLink> dentro de la etiqueta <Context> .

```
<ResourceLink name="jdbc/mydb"
            global="jdbc/mydb"
            auth="Container"
            type="javax.sql.DataSource" />
```

6. Salva el fichero *context.xml*
7. Abre el fichero *hibernate-cfg.xml* file y añade las siguientes propiedades.

Añade:

```
-------
<property name="connection.datasource">java:comp/env/jdbc/mydb</property>

Elimina:
--------
<!--<property name="connection.url">jdbc:mysql://localhost:3306/mydb</property>
-->
<!--<property name="connection.username">root</property> -->
<!--<property name="connection.password"></property> -->
```

8. Reinicia el tomcat para comprobar el funcionamiento.

Configurar el pool de conexiones por aplicación usando C3P0

1.- Para integrar c3p0 con Hibernate necesitas incorporar la dependencia hibernate-c3p0.jar del repositorio de Maven

2.- Para configurar c3p0 coloca los detalles de configuración en el fichero hibernate.cfg.xml como sigue:

```
<property name="hibernate.c3p0.min_size">5</property>
<property name="hibernate.c3p0.max_size">20</property>
<property name="hibernate.c3p0.timeout">300</property>
<property name="hibernate.c3p0.max_statements">50</property>
<property name="hibernate.c3p0.idle_test_period">3000</property>
```

Donde hibernate.c3p0.min_size son el mínimo de conexiones en el pool, por defecto 1.

hibernate.c3p0.max_size, son el máximo número de conexiones en el pool, por defecto 100

hibernate.c3p0.timeout, es el tiempo de inactividad despues del que es borrado del pool una conexión, por defecto 0, nunca expira

hibernate.c3p0.max_statements, numero de sentencias preparadas almacenadas en caché, lo que mejora el rendimiento. Por defecto 0.

hibernate.c3p0.idle_test_period, es el lapso del tiempo en segundos, antes de que la conexión sea automáticamente validada. Por defecto 0.

3.- El pool de conexiones ya está funcionando al ejecutar la aplicación web.