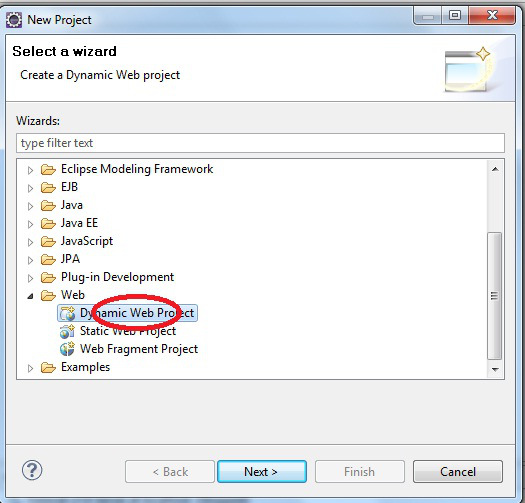
**SERVLETS**

Un servlet es un componente Web y  se ejecuta en un contenedor Web. Los servlets son componentes que forman parte de una aplicación Web.

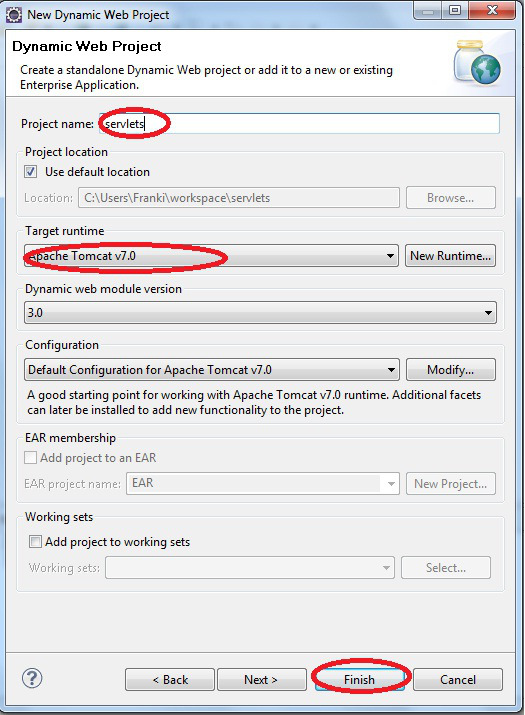
Por lo tanto, todos los ejemplos de servlets los desplegaremos sobre una aplicación web creada en **Eclipse**.

**Crear un proyecto web en Tomcat 7 y desplegarlo desde Eclipse**

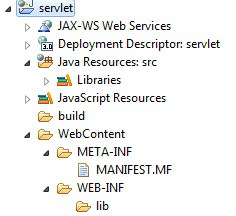
Para crear un proyecto web vacío en Eclipse pinchamos en **File –> New –> Project**, desplegamos Web y seleccionamos**Dynamic Web Project**.



Le damos un nombre a nuestro proyecto, **servlet**por ejemplo y seleccionamos el servidor**Apache Tomcat 7.0** que configuramos en la entrada anterior. Pulsamos en **Finish** y Eclipse nos creará un nuevo proyecto Web en nuestro **workspace**.

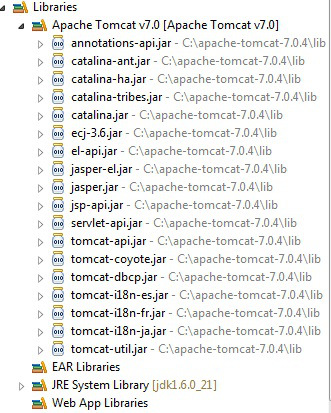


Si desplegamos el proyecto desde el explorador de proyectos, podemos ver la estructura por defecto que crea Eclipse para los proyectos Web de la plataforma Java EE.



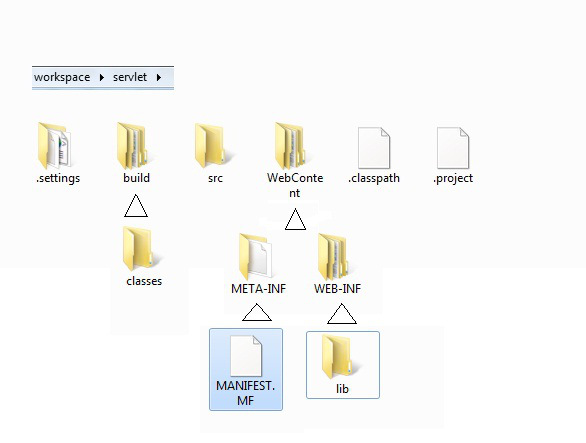
* En la carpeta **Java Resources:src** guardaremos el código fuente de las clases Java empaquetadas en packages.
* En la carpeta **WebContent** guardaremos los archivos Web (HTML, JavaScript, CSS, JSP, imágenes, documentos, etc).
* En la carpeta **WebContent/WEB-INF** guardaremos el descriptor de despliegue**web.xml**.
* En la carpeta **WebContent/WEB-INF/lib** guardaremos las librerías externas que utilizemos en la aplicación Web Java.

En la carpeta **Java Resources:src**, dentro de **Libraries**, podemos encontrar las librerías de Apache Tomcat 7, donde se encuentra las librerías necesarias para desarrollar servlets, en concreto **servlet-api.jar**.



Como comentamos en la entrada anterior, la plataforma Java EE es un conjunto de especificaciones y la tecnología de los servlets es una tecnología que cuenta con su propia especificación, de tal forma que cualquier empresa pueda desarrollar un producto que las cumpla. Estamos en esa situación ya que la organización Apache ha desarrollado Tomcat 7, un contenedor Web que nos permite gestionar y ejecutar los componentes servlets, es decir, podemos utilizar ese archivo JAR para desarrollar componentes servlets y desplegarlos en el contenedor Tomcat 7.

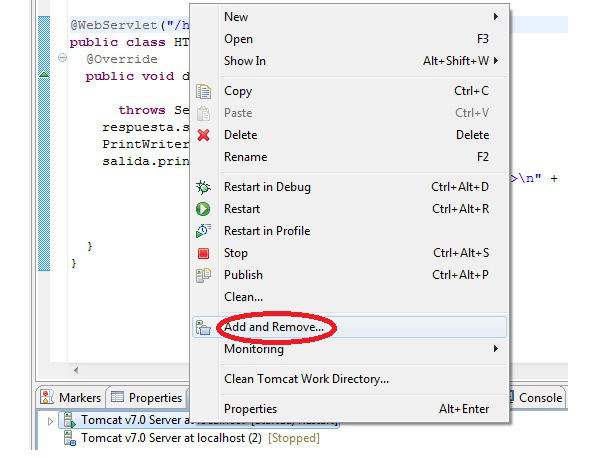
Si seguimos siendo curiosos podemos entrar dentro de la carpeta donde eclipse guarda todo el espacio de trabajo para ver qué es lo que está creando por debajo de lo que nosotros estamos haciendo en el IDE. Podemos observar que crea una nueva carpeta por cada proyecto creado y dentro de ella almacena las carpetas **src** y **WebContent** comentadas anteriormente, además de una carpeta extra **/build/classes**, donde almacena las clases Java compiladas empaquetadas en sus respectivos paquetes.

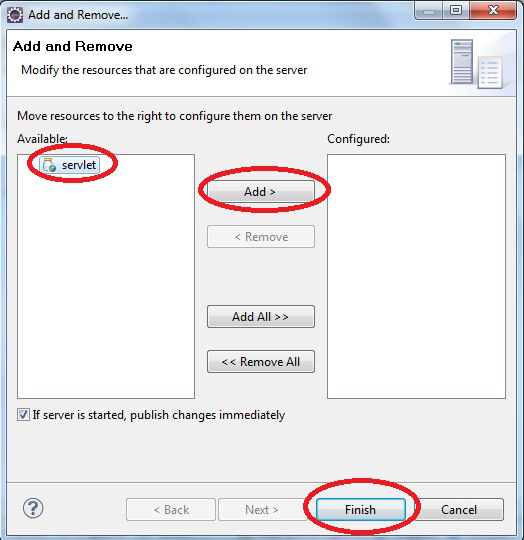


Realmente esta estructura de directorios no es la estructura estándar de directorios de una aplicación web Java. Si seguimos curioseando podemos acceder a la carpeta **workspace/.metadata/.plugins/ org.eclipse.wst.server.core/tmp0/wtpwebapps/** y encontraremos una carpeta llamada igual que el nombre de nuestro proyecto. Todos los directorios que conforman esta carpeta si que forman la estructura estándar de una aplicación web Java EE y es la carpeta real que es desplegada dentro del contenedor Tomcat 7.

Si copiáramos esta carpeta dentro del directorio **tomcat\_install\_dir/web-apps/** estaríamos desplegando la aplicación en Tomcat manualmente.

Para desplegar nuestra aplicación web en Tomcat 7 desde Eclipse tenemos que pinchar con el botón derecho del ratón sobre el servidor Tomcat definido en la pestaña Servers, y seleccionar la opción Add and Remove.

Seleccionamos nuestro proyecto y pulsamos en el botón **Add** para añadirlo a Tomcat 7. Pulsamos **Finish** y ya tenemos nuestra aplicación web desplegada en Tomcat 7.



Podríamos iniciar Tomcat y probarla pero la aplicación de ejemplo aún está vacía, así que ese paso lo vamos a dejar para más adelante.

**Crear un proyecto web y desplegarlo en Tomcat 7 manualmente**

Es buena idea, aunque sea solo la primera vez, crear una aplicación manualmente y desplegarla en Tomcat. Una aplicación Web Java tiene una estructura estándar que no es la misma que utiliza Eclipse en su espacio de trabajo.

Lo primero es crear un **directorio raíz con el nombre de nuestro proyecto** donde colocaremos la aplicación web.

**appName/** : El directorio raíz de nuestra aplicación puede contener los elementos típicos de un sitio web, es decir, documentos estáticos HTML, hojas de estilo CSS, archivos JavaScript, imágenes, documentos pdf, etc. En este directorio residen también las páginas JSPs de la aplicación, pero no los servlets.

**appName/WEB-INF/web.xml** : El descriptor de despliegue de nuestra aplicación. Podemos ignorar este archivo si utilizamos anotaciones en nuestros servlets. Esto es una de las nuevas características de la plataforma Java EE 6 y de la especificación 3.0 de los servlets. Si te has fijado Eclipse no crea un web.xml por defecto.

**appName/WEB-INF/classes/** : Este directorio contiene las clases Java utilizadas dentro de la aplicación. En este directorio residen los servlets. Es una buena práctica empaquetar todas las clases en paquetes segun su función dentro de la aplicación.

**appName/WEB-INF/lib/** : Este directorio contiene los archivos JAR que serán utilizados por la aplicación, como por ejemplo el conector utilizado para conectarse a una base de datos.

El siguiente paso es configurar la variable de entorno**CLASSPATH** para que el compilador encuentre el archivo JAR de los servlets. Establecer incorrectamente esta variable es lo que más problemas ocasiona. Introducimos el siguiente comando en la líena de comandos:

**SET CLASSPATH = .;install\_tomcat\_dir\lib\servlet-api.jar;**

Si estamos en Windows 7 podemos modificar el CLASSPATH desde*Equipo, botón derecho, Configuración avanzada del sistema, Variables de entorno*.

Para desplegar el proyecto en Tomcat 7 simplemente tenemos que copiar el directorio raíz al directorio de despliegue de Tomcat, el directorio **install\_tomcat\_dir\webapps**. Nuestra aplicación se autodesplegará automáticamente.

Veremos cómo compilar servlets manualmente y como deben organizarse más adelante.

**¿Qué es un servlet?**

Los servlets son programas Java que se ejecutan en un contenedor Web dentro de un servidor de aplicaciones y actúan como una capa intermedia entre la petición de un cliente y la aplicación del servidor o la base de datos.

Un servlet puede:

1. Leer los datos explícitos enviados por el cliente. Suele ser información introducida en un formulario HTML.
2. Leer los datos implícitos de la petición HTTP enviada por el navegador. La información HTTP incluye las cookies, y toda la información que se incluye dentro de la cabecera de una petición HTTP.
3. Generar resultados. Este proceso puede requerir comunicarse con la base de datos, ejecutar una llamada RMI o CORBA, invocar un servicio Web o generar la respuesta directamente.
4. Enviar datos explícitos al cliente. El documento puede ser enviado en una gran variedad de formatos, ya sea HTML o XML, imágenes GIF o JPG, archivos EXCEL, etc.
5. Enviar datos ímplicitos en la respuesta HTTP al cliente. La información HTTP incluye el establecimiento de cookies y cualquier cabecera que pueda enviarse dentro de la respuesta HTTP, como el almacenamiento en caché, etc.

Podemos resumir que un **servlet se dedica a responder peticiones HTTP de los navegadores clientes y a generar resultados dinámicos en respuesta a estas peticiones.**Los servlets no solo se limitan a responder peticiones HTTP. Existen servlets que pueden ser embebidos en un servidor FTP o en servidores de correo, pero en la práctica solamente se utilizan los servlets que responden peticiones HTTP.

**Estructura básica de un servlet**

Los servlets normalmente extienden la clase **HttpServlet**y sobreescriben el método **doPost** o **doGet,** dependiendo de si los datos han sido enviados por el método POST o GET. Podemos realizar la misma acción tanto para peticiones GET y POST simplemente llamando a doGet desde doPost o viceversa.

Ambos métodos, doPost y doGet, toman dos argumentos, un objeto **HttpServletRequest**que encapsula la petición HTTP, y un objeto **HttpServletResponse** que encapsula la respuesta HTTP. El objeto HttpServletRequest nos permite acceder tanto a los datos como a las cabeceras de la petición HTTP. El objeto HttpServletResponse nos permite modificar la información de salida, como los códigos de estado HTTP o las cabeceras de respuesta.

Además podemos obtener un objeto**PrintWriter** sobre el que generar el contenido dinámico del documento al cliente.

Los métodos doGet y doPost pueden lanzar las excepciones **ServletException** e **IOexception**. Por último, se deben de importar los paquetes **java.io** (para el PrintWriter),**javax.servlet** (para el HttpServlet) y **javax.servlet.http** (para el HttpServletRequest y HttpServletResponse).

Vamos a ver una serie de ejemplos muy básicos de servlets.

**Primer ejemplo: Un servlet que genera texto plano:**

Vamos a ver cómo crear nuestro primer servlet. En este caso el servlet generará como salida texto plano con la cadena Hola Mundo!

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** HolaMundoServlet **extends** HttpServlet {

 @Override

**public** **void** doGet(HttpServletRequest peticion,

                   HttpServletResponse respuesta)

**throws** ServletException, IOException {

   PrintWriter salida = respuesta.getWriter();

   salida.println("Hola Mundo!");

 }

}

**Segundo ejemplo: Un servlet que genera HTML:**

Para que el servlet genere HTML tendremos que decirle al navegador que le vamos a enviar texto HTML. Esto se consigue estableciendo la cabecera **Content-Type** a **text/html.** Generalmente modificamos las cabeceras HTTP desde el objeto **HttpServletResponse** utilizando el método**setHeader**pero como establecer el tipo de contenido es una tarea tan común en los servlets, existe el método **setContentType**. Utilizando este método podemos decirle al navegador el tipo de los datos que le estamos enviando. Por ejemplo si le estuviéramos enviando una imagen gif estableceríamos el Content-Type a image/gif. Al igual que sucede en PHP por ejemplo, las cabeceras HTTP de la respuesta deben enviarse antes del contenido generado.

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** HTMLServlet **extends** HttpServlet {

 @Override

**public** **void** doGet(HttpServletRequest peticion,

                   HttpServletResponse respuesta)

**throws** ServletException, IOException {

   respuesta.setContentType("text/html");

   PrintWriter salida = respuesta.getWriter();

   salida.println("<html>**\n**" +

               "<head><title>Hola Mundo!</title></head>**\n**" +

               "<body>**\n**" +

               "<h1>Hola Mundo!</h1>**\n**" +

               "</body></html>");

 }

}

Generar contenido HTML desde un servlet no es una buena opción, imagínate tener que estar imprimiendo cada cadena que forma parte del contenido de una página web. Se hace realmente tedioso y difícil de mantener. Por eso surgió la tecnología JSP, para hacernos la vida más fácil. Hablaremos de ella más adelante.

**Tercer ejemplo: Empaquetando servlets**

Cuando desarrollamos servlets no es buena idea colocar todos en el mismo directorio dentro de la jerarquía de directorios de una aplicación Web. Es una buena práctica empaquetar los servlets en **packages** para evitar conflictos y poder gestionarlos más fácilmente.

Para empaquetar una clase a nivel de código fuente basta con utilizar la sentencia:**package nombreDePackage;** como primera línea del archivo.

Para empaquetar un servlet en una aplicación Web desde Eclipse basta con pinchar con el botón derecho del ratón sobre la carpeta **Java Resources:src** y elegir la opción **New –> Package**. Después situamos todas las clases de nuestros ejemplos en este paquete. El IDE Eclipse ya se encarga a la hora de compilar y desplegar de crear el directorio que representa ese paquete **tanto en el código fuente** como en la carpeta **classes**.

Si no estamos utilizando un IDE como Eclipse tendremos que crear a mano el directorio con el nombre del paquete dentro del directorio donde estemos desarrollando los ejemplos como en el directorio **classes**. Además recordar introducir la sentencia package dentro del código fuente de la clase del servlet.

**package** paquete;

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** HTMLServlet **extends** HttpServlet {

 @Override

**public** **void** doGet(HttpServletRequest peticion,

                   HttpServletResponse respuesta)

**throws** ServletException, IOException {

   respuesta.setContentType("text/html");

   PrintWriter salida = respuesta.getWriter();

   salida.println("<html>**\n**" +

               "<head><title>Hola Mundo!</title></head>**\n**" +

               "<body>**\n**" +

               "<h1>Hola Mundo!</h1>**\n**" +

               "</body></html>");

 }

}

**Cuarto ejemplo: Un servlet que utiliza otras clases Java**

Tal como sucede con los servlets, es una buena práctica empaquetar todas las clases Java de la aplicación, ya sean servlets, clases de ayuda, beans (los veremos más adelante), etc.

En este ejemplo vamos a crear una clase con un método estático que se encargue de generar el HTML del ejemplo anterior. Es un ejemplo muy básico y soso ya que no tiene mucha miga, pero basta para demostrar que podemos utilizar todo tipo de clases dentro de un servlet. Si la clase esta en otro paquete dentro de la aplicación deberemos importarlo con la sentencia **import**, de todas formas esto no hace falta que lo diga, porque vuelvo a repetir que eso tiene que estar muy masticado ya.

**package** paquete;

**public** **class** GeneradorHTML {

**public** **static** String generar() {

**return** "<html>**\n**" +

          "<head><title>Hola Mundo!</title></head>**\n**" +

          "<body>**\n**" +

          "<h1>Hola Mundo!</h1>**\n**" +

          "</body></html>";

 }

}

**package** paquete;

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** HTMLServletConGenerador **extends** HttpServlet {

 @Override

**public** **void** doGet(HttpServletRequest peticion,

                   HttpServletResponse respuesta)

**throws** ServletException, IOException {

   respuesta.setContentType("text/html");

   PrintWriter salida = respuesta.getWriter();

   salida.println(GeneradorHTML.generar());

 }

}

**Compilar los servlets y las clases Java que forman nuestra aplicación**

Desde Eclipse no hace falta que compilemos ninguna clase, el IDE se encarga por si mismo de ir compilando al vuelo y notificándonos los errores de sintaxis o de programación que cometemos en el código fuente. Esta es una de las ventajas de utilizar un IDE.

Sino hemos utilizado un IDE mientras seguíamos los ejemplos tendremos que compilar todas las clases Java con el compilador javac mediante la línea de comandos para generar los **.class**que se situarán dentro de la carpeta**WEB-INF\classes**de la aplicación.

Voy a suponer que se tiene creada la estructura de directorios estándard de una aplicación web Java, y además está situada en el directorio donde se despliegan las aplicaciones en Tomcat.

Ya habíamos establecido la variable de entorno**CLASSPATH** para que apuntara al JAR del servlet. Quién no haya utilizado un IDE se habrá creado un directorio de trabajo donde desarrollar las clases Java de la aplicación. Este directorio al menos tiene que tener un directorio llamado **paquete** que contiene cuatro archivos**.java** con las clases que hemos desarrollado en los ejemplos.

Mediante la línea de comandos nos situamos en el directorio de trabajo que nos hayamos creado, es decir el directorio que contiene el package principal y compilamos todas las clases Java:

**javac paquete\\*.java**

Si todo ha funcionado bien nos habrá creado cuatro archivos .class que hay que copiar dentro de la carpeta **WEB-INF\classes** de la aplicación en un directorio llamado**paquete**. Si nos ha dado algun error seguramente sea porque la variable CLASSPATH está mal configurada.

Como veis es un auténtico coñazo tener que compilar todo a mano mediante la línea de comandos, estar configurando a mano el CLASSPATH y tener que copiar archivos de un lado para otro. Sin mencionar la necesidad de ejecutar los .bat para iniciar y parar Tomcat 7.

Es bueno realizarlo la primera vez, más que nada para saber porque funciona todo, pero de ahora en adelante todos los ejemplos los crearemos en algún IDE como Eclipse o NetBeans.

**El descriptor de despliegue**

El descriptor de despliegue de una aplicación web (**web.xml**) es un archivo escrito en XML que describe diversas características de la aplicación web.

Vamos a ver como describir los servlets de la aplicación y poco más. Iremos viendo más adelante como configurar otras características de la aplicación en este archivo.

Si utilizamos un contendor de servlets que implemente la especificación 3.0 de los servlets no es necesario el uso del descriptor de despliegue. Podemos configurar los servlets mediante anotaciones en el propio código fuente.

Una plantilla para la versión 3.0 de la especificación puede ser la siguiente:

**<web-app** version="3.0"

        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\_3\_0.xsd"**>**

**<servlet>**

**<servlet-name>**Nombre del servlet**</servlet-name>**

**<servlet-class>**Paquete del servlet.Clase del servlet**</servlet-class>**

**</servlet>**

**<servlet-mapping>**

**<servlet-name>**Nombre del servlet**</servlet-name>**

**<url-pattern>**Patron de ruta del servlet**</url-pattern>**

**</servlet-mapping>**

**</web-app>**

El primer elemento **<web-app>** indica el inicio de la aplicación, es dentro de este elemento donde se definen todos los elementos restantes.

El elemento **<servlet>** define las características de un servlet y a su vez está compuesto por los elementos **<servlet-name>** y **<servlet-class>** que indican un nombre corto para el servlet así como el nombre de la clase Java que contiene el servlet, respectivamente. La clase Java se indica con la ruta completa de packages.

Posteriormente se define el elemento **<servlet-mapping>** para establecer la ubicación en términos de URL. Está compuesto por los elementos **<servlet-name>** y **<url-pattern>** que especifican el nombre del servlet que será accedido a través del un patrón URL .

Vamos a ver como definir el descriptor de despliegue para describir los servlets que hemos utilizado en nuestra aplicación de ejemplos. Creamos un archivo nuevo en **WEB-INF** llamado**web.xml**con el siguiente contenido, tanto si estamos realizando los ejemplos desde Eclipse como si lo estamos realizando manualmente:

**<web-app** version="3.0"

        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\_3\_0.xsd"**>**

**<servlet>**

**<servlet-name>**ServletEjemplo1**</servlet-name>**

**<servlet-class>**paquete.HolaMundoServlet**</servlet-class>**

**</servlet>**

**<servlet-mapping>**

**<servlet-name>**ServletEjemplo1**</servlet-name>**

**<url-pattern>**/ejemplo1**</url-pattern>**

**</servlet-mapping>**

**<servlet>**

**<servlet-name>**ServletEjemplo2**</servlet-name>**

**<servlet-class>**paquete.HTMLServlet**</servlet-class>**

**</servlet>**

**<servlet-mapping>**

**<servlet-name>**ServletEjemplo2**</servlet-name>**

**<url-pattern>**/ejemplo2**</url-pattern>**

**</servlet-mapping>**

**<servlet>**

**<servlet-name>**ServletEjemplo3**</servlet-name>**

**<servlet-class>**paquete.HTMLServletConGenerador**</servlet-class>**

**</servlet>**

**<servlet-mapping>**

**<servlet-name>**ServletEjemplo3**</servlet-name>**

**<url-pattern>**/ejemplo3**</url-pattern>**

**</servlet-mapping>**

**</web-app>**

En él hemos definido los tres servlets que hemos explicado en los ejemplos y les hemos establecido la ruta URL de acceso.

El siguiente paso es desplegar la aplicación en Tomcat 7, iniciar Tomcat 7 y acceder a los servlets de ejemplos desde el navegador utilizando las rutas URLs que hemos definidio en el descriptor de despliegue.

Accedemos a los servlets desde el navegador utilizando las siguientes URLs:

<http://localhost/servlet/ejemplo1>

<http://localhost/servlet/ejemplo2>

<http://localhost/servlet/ejemplo3>

Podemos observar que la URL se conforma de la siguiente manera:

[**http://hostName/appName/servletPattern**](http://hostname/appName/servletPattern)

También podemos describir los servlets utilizando anotaciones en la clase del servlet. Para poder utilizar las anotaciones debemos de utilizar un contenedor servlet que implemente la especificación 3.0, sino es así debemos de utilizar el descriptor de despliegue.

Para poder utilizar las anotaciones debemos de importar el paquete **annotations** y utilizar la anotación **@WebServlet(“URL-Pattern”)**. Vamos a ver nuestros tres ejemplos utilizando las anotaciones:

**package** paquete;

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**import** javax.servlet.annotations.\*;

@WebServlet("/ejemplo1")

**public** **class** HolaMundoServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest peticion,

                   HttpServletResponse respuesta)

**throws** ServletException, [IOException](http://www.google.com/search?hl=en&q=allinurl%3Aioexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[PrintWriter](http://www.google.com/search?hl=en&q=allinurl%3Aprintwriter+java.sun.com&btnI=I%27m%20Feeling%20Lucky) salida = respuesta.getWriter();

   salida.println("Hola Mundo!");

 }

}

**package** paquete;

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**import** javax.servlet.annotations.\*;

@WebServlet("/ejemplo2")

**public** **class** HTMLServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest peticion,

                   HttpServletResponse respuesta)

**throws** ServletException, [IOException](http://www.google.com/search?hl=en&q=allinurl%3Aioexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

   respuesta.setContentType("text/html");

[PrintWriter](http://www.google.com/search?hl=en&q=allinurl%3Aprintwriter+java.sun.com&btnI=I%27m%20Feeling%20Lucky) salida = respuesta.getWriter();

   salida.println("<html>**\n**" +

               "<head><title>Hola Mundo!</title></head>**\n**" +

               "<body>**\n**" +

               "<h1>Hola Mundo!</h1>**\n**" +

               "</body></html>");

 }

}

**package** paquete;

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**import** javax.servlet.annotations.\*;

@WebServlet("/ejemplo3")

**public** **class** HTMLServletConGenerador **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest peticion,

                   HttpServletResponse respuesta)

**throws** ServletException, [IOException](http://www.google.com/search?hl=en&q=allinurl%3Aioexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

   respuesta.setContentType("text/html");

[PrintWriter](http://www.google.com/search?hl=en&q=allinurl%3Aprintwriter+java.sun.com&btnI=I%27m%20Feeling%20Lucky) salida = respuesta.getWriter();

   salida.println(GeneradorHTML.generar());

 }

}