

Servidores Web

Índice

1	Protocolo HTTP. Introducción a las aplicaciones web JavaEE.....	3
1.1	Protocolo HTTP.....	3
1.2	Introducción a las aplicaciones web en Java.....	11
1.3	Despliegue de archivos WAR.....	15
2	Ejercicios de protocolo HTTP e introducción a las aplicaciones web.....	18
2.1	Pruebas con protocolo HTTP (1 punto).....	18
2.2	Recursos estáticos en Tomcat (0,5 puntos).....	19
2.3	Desplegar una aplicación web sencilla sin Eclipse (1 punto).....	19
3	Configuración de Aplicaciones Web en Tomcat. Desarrollo con Eclipse WTP.....	21
3.1	Desarrollo y despliegue con WebTools.....	21
3.2	Configuración de Tomcat.....	25
3.3	Configuración de aplicaciones web en Tomcat.....	28
3.4	Servicios en JavaEE.....	34
4	Ejercicios de configuración de Tomcat y aplicaciones web.....	37
4.1	Uso de Eclipse WTP (1 punto).....	37
4.2	Configuración a través del web.xml (0.5 puntos).....	38
4.3	Configuración de fuentes de datos (1.5 puntos)	38
4.4	Configuración con el contexto (1 punto).....	39
5	Desarrollo, despliegue y pruebas de aplicaciones web con Maven.....	40
5.1	Creación y empaquetado de un proyecto web con Maven.....	40
5.2	Despliegue de aplicaciones web con maven.....	43
5.3	Pruebas en aplicaciones web.....	46
6	Ejercicios de desarrollo, despliegue y pruebas de aplicaciones web con Maven.....	52
6.1	Creación de un proyecto web en Eclipse basado en Maven (1 pto).....	52
6.2	Despliegue de la aplicación en Maven usando Cargo (1 pto).....	53
6.3	Pruebas funcionales con JWebUnit (1 pto).....	53

7 Seguridad en aplicaciones web.....	56
7.1 Seguridad del servidor.....	56
7.2 Seguridad en aplicaciones web.....	59
8 Ejercicios de seguridad en Tomcat.....	67
8.1 Creación de un Realm de base de datos (1 pto).....	67
8.2 Seguridad basada en formularios (1 pto).....	67
8.3 Seguridad del servidor (1 punto).....	67

1. Protocolo HTTP. Introducción a las aplicaciones web JavaEE

1.1. Protocolo HTTP

El protocolo HTTP especifica el modo de comunicación entre una máquina cliente y una máquina servidor, de modo que el cliente solicita un documento del espacio de direcciones del servidor, y éste se lo sirve.

HTTP es un protocolo que no tiene estado: un cliente realiza una petición al servidor, que contesta y la transacción acaba, con lo que en la siguiente petición que pueda realizar el mismo cliente se deben proporcionar de nuevo todos los datos necesarios para que el servidor sirva correctamente la nueva petición, no habiendo ninguna relación entre las peticiones.

1.1.1. Peticiones del cliente

En el protocolo HTTP el cliente realiza una **petición** que se descompone en:

- Un comando HTTP, seguido de una dirección de documento o URI (*Uniform Resource Identifier*), y un número de versión HTTP, de forma que se tiene una línea con el formato:

Comando	URI	Protocolo
---------	-----	-----------

Por ejemplo:

GET	/index.html	HTTP/1.1
-----	-------------	----------

- Tras la petición, el cliente puede enviar información adicional de **cabeceras** (*headers*) con las que se da al servidor más información sobre la petición (tipo de software que ejecuta el cliente, tipo de contenido (`content-type`) que entiende el cliente, etc). Esta información puede utilizarla el servidor para generar la respuesta apropiada. Las cabeceras se envían una por línea, donde cada una tiene el formato:

Clave:	Valor
--------	-------

Por ejemplo:

Accept-Encoding:	gzip, deflate
User-Agent:	Mozilla/4.0 (compatible;MSIE5.0;Windows 98)

Tras las cabeceras, el cliente envía una línea en blanco (`\r\n\r\n`) para indicar el final de la sección de cabeceras.

- Finalmente, de forma opcional, se pueden enviar **datos adicionales** si el comando HTTP solicitado lo requiere (por ejemplo, el método POST que veremos a continuación).

METODO GET

El comando `GET` permitía al principio solicitar al servidor un documento estático, existente en su espacio de direcciones. Luego se vio que esto no era suficiente, y se introdujo la posibilidad de solicitar búsquedas al servidor, de forma que el documento no tuviera que ser necesariamente estático, sino que la búsqueda estuviera condicionada por unos determinados parámetros. Así, el comando `GET` tiene la forma:

```
GET direccion ? parametros version HTTP
```

Por ejemplo:

```
GET /cgi-bin/pagina.cgi?IDIOMA=C&MODELO=a+b HTTP/1.1
```

Los parámetros se indican con pares *nombre=valor*, separados por '&', y reciben el nombre de **datos de formulario**. El URI no puede contener espacios ni algunos caracteres, por lo que se utilizan códigos especiales, como el '+' para indicar espacio en blanco, u otros códigos %xx para representar otros caracteres. Uno de los trabajos más duros de los programas CGI es procesar esta cadena de parámetros para extraer la información necesaria.

OTROS METODOS

En la versión 1.1 de HTTP se definen otros métodos además de `GET`:

- `OPTIONS`: para consultar al servidor acerca de las funcionalidades que proporciona
- `HEAD`: el servidor responde de forma idéntica a un comando `GET`, pero no devuelve el cuerpo del documento respuesta, sólo las cabeceras. Suele emplearse para comprobar características del documento.
- `POST`: se emplea para enviar al servidor un bloque de datos en el cuerpo de la petición
- `PUT`: solicita que el cuerpo de la petición que envía se almacene en el espacio de direcciones del servidor, con el identificador URI solicitado (guarda un documento en el servidor)
- `DELETE`: solicita borrar un documento específico del servidor
- `TRACE`: se utiliza para seguir el camino de la petición por múltiples servidores y proxies (útil para depurar problemas de red).

GET Y POST

Los dos métodos más comúnmente usados son `GET` y `POST`. Veremos las diferencias entre uno y otro con un ejemplo:

- Un ejemplo de petición `GET` es:

```
GET /dir/cargaPagina.php?id=21&nombre=Pepe HTTP/1.1
<cabeceras>
```

- Este ejemplo, convertido a petición `POST` es:

```
POST /dir/cargaPagina.php HTTP/1.1
<cabeceras>
```

```
id=21&nombre=Pepe
```

Vemos que los parámetros se pasan en el cuerpo de la petición, fuera de la línea del comando.

Comúnmente existen 3 formas de enviar una petición GET:

- Teclear la petición directamente en la barra del navegador:

```
http://www.xx.com/pag.html?id=123&nombre=pepe
```

- Colocar la petición en un enlace y pinchar el enlace para realizarla:

```
<a href="http://www.xx.com/pag.html?id=123&nombre=pepe">Pulsa  
Aqui</a>
```

- Enviar la petición tras rellenar un formulario con METHOD=GET (o sin METHOD) con los dos parámetros a enviar:

```
<html>  
<body>  
    <form action="http://www.xx.com/pag.html">  
        <input type="text" name="id" value="123">  
        <input type="text" name="nombre" value="pepe">  
        <input type="submit" value="Enviar">  
    </form>  
</body>  
</html>
```

Para enviar una petición POST, normalmente se utiliza un formulario con METHOD=POST:

```
<html>  
<body>  
    <form action="http://www.xx.com/pag.html" METHOD=POST>  
        <input type="text" name="id" value="123">  
        <input type="text" name="nombre" value="pepe">  
        <input type="submit" value="Enviar">  
    </form>  
</body>  
</html>
```

1.1.2. Respuestas del servidor

Las respuestas del servidor también tienen tres partes:

- Una **línea de estado** con la versión del protocolo HTTP utilizado en el servidor, un código de estado y una breve descripción del mismo:

```
HTTP/1.0 200 OK
```

- Información de **cabeceras**, donde se envía al cliente información sobre el servidor y sobre el documento solicitado. El formato de estas cabeceras es el mismo que el visto para las peticiones del cliente, terminando en una línea en blanco.
- Finalmente, se envía el **documento solicitado**. Para marcar el final del mismo se

envía también otra línea en blanco.

1.1.3. Cabeceras

Vamos a poder implementar programas que lean las cabeceras que envía un cliente (un navegador, por ejemplo) y que modifiquen el documento servido en función de dichas cabeceras (por ejemplo, enviar una página en función del idioma que se especifique). Por otra parte, podremos utilizar las cabeceras que envíe el servidor como respuesta para obligar al navegador a hacer determinadas acciones, como saltar a otra URL.

Veremos a continuación las cabeceras más comunes tanto en las peticiones de los clientes como en las respuestas de los servidores. La RFC donde se especifican estas cabeceras es la 2616.

CABECERAS DE PETICION (HTTP/1.1)

- **Accept:** Tipos MIME que puede manejar el cliente
- **Accept-Charset:** Conjunto de caracteres que el cliente puede manejar
- **Accept-Encoding:** Define si el navegador puede aceptar datos codificados
- **Accept-Language:** Idiomas aceptados
- **Authorization:** Para identificarse cuando se accede a páginas protegidas
- **Cache-Control:** Opciones relacionadas con el servidor proxy. Esta cabecera se llamaba *Pragma* en HTTP 1.0
- **Connection:** Define si el cliente es capaz de realizar conexiones persistentes (*keep-alive*, valor por defecto), o no (*close*). Nueva en HTTP 1.1
- **Content-Length:** Longitud de los datos enviados. Aplicable a peticiones POST
- **Content-Type:** Tipo MIME de los datos enviados. Aplicable a peticiones POST
- **Cookie:** Para las cookies que se manejen
- **From:** Dirección de correo electrónico responsable de la petición
- **Host:** Unica cabecera requerida por HTTP 1.1. Indica el host y el puerto tal y como se especifica en la URL original.
- **If-Modified-Since:** El cliente sólo desea el documento si ha sido modificado después de la fecha indicada en esta cabecera.
- **Referer:** URL origen de la petición. Si estamos en la página 1 y pinchamos en un enlace a la página 2, la URL de la página 1 se incluye en esta cabecera cuando se realiza la petición de la página 2.
- **User-Agent:** Cliente que está realizando la petición (normalmente muestra datos del navegador, como nombre, etc).

CABECERAS DE RESPUESTA

- **Allow:** Métodos disponibles (GET, POST, etc) a los que puede responder el recurso que se está solicitando
- **Cache-Control:** Dice al cliente en qué circunstancias puede hacer una caché del documento que está sirviendo:
 - `public`: el documento puede almacenarse en una caché

- **private:** el documento es para un solo usuario y sólo puede almacenarse en una caché privada (no compartida)
- **no-cache:** el documento nunca debe ser almacenado en caché
- **no-store:** el documento no debe almacenarse en caché ni almacenarse localmente de forma temporal en el disco duro
- **must-revalidate:** el cliente debe revalidar la copia del documento con el servidor original, no con servidores proxy intermedios, cada vez que se use
- **max-age=xxx:** el documento debe considerarse caducado después de *xxx* segundos.

Esta cabecera se llamaba **Pragma** en HTTP 1.0

- **Content-Encoding:** Tipo de compresión (*gzip*, etc) en que se devuelve el documento solicitado
- **Content-Language:** Idioma en que está escrito el documento. En la RFC 1766 están los idiomas disponibles
- **Content-Length:** Número de bytes de la respuesta
- **Content-MD5:** Una forma de fijar el *checksum* (verificación de integridad) del documento enviado
- **Content-Type:** Tipo MIME de la respuesta
- **Date:** Hora y fecha, en formato GMT, en que la respuesta ha sido generada
- **Expires:** Hora y fecha, en formato GMT, en que la respuesta debe considerarse caducada
- **Last-Modified:** Fecha en que el documento servido se modificó por última vez. Con esto, el documento se sirve sólo si su *Last-Modified* es mayor que la fecha indicada en el *If-Modified-Since* de la cabecera del cliente.
- **Location:** Indica la nueva URL donde encontrar el documento. Debe usarse con un código de estado de tipo 300. El navegador se redirigirá automáticamente a la dirección indicada en esta cabecera.
- **Refresh:** Indica al cliente que debe recargar la página después de los segundos especificados. También puede indicarse la dirección de la página a cargar después de los segundos indicados:

```
Refresh: 5; URL=http://www.unapagina.com
```

- **Set-Cookie:** Especifica una cookie asociada a la página
- **WWW-Authenticate:** Tipo de autorización y dominio que debería indicar el cliente en su cabecera *Authorization*.

Para colocar estas cabeceras en un documento se tienen varios métodos, dependiendo de cómo estemos tratando las páginas (mediante servlets, HTML, etc). Por ejemplo, con HTML podemos enviar cabeceras mediante etiquetas **META** en la cabecera (<HEAD>) de la página HTML:

```
<META HTTP-EQUIV="Cabecera" CONTENT="Valor">
```

Por ejemplo:

```
<META HTTP-EQUIV="Location" CONTENT="http://www.unapagina.com">
```

1.1.4. Códigos de estado

El código de estado que un servidor devuelve a un cliente en una petición indica el resultado de dicha petición. Se tiene una descripción completa de los mismos en el RFC 2616. Están agrupados en 5 categorías:

- **100 - 199:** códigos de información, indicando que el cliente debe responder con alguna otra acción.
- **200 - 299:** códigos de aceptación de petición. Por ejemplo:

200	OK	Todo está bien
204	No Content	No hay documento nuevo

- **300 - 399:** códigos de redirección. Indican que el documento solicitado ha sido movido a otra URL. Por ejemplo:

301	Moved Permanently	El documento está en otro lugar, indicado en la cabecera <i>Location</i>
302	Found	Como el anterior, pero la nueva URL es temporal, no permanente.
304	Not Modified	El documento pedido no ha sufrido cambios con respecto al actual (para cabeceras If-Modified-Since)

- **400 - 499:** códigos de error del cliente. Por ejemplo:

400	Bad Request	Mala sintaxis en la petición
401	Unauthorized	El cliente no tiene permiso para acceder a la página. Se debería devolver una cabecera WWW-Authenticate para que el usuario introduzca login y password
403	Forbidden	El recurso no está disponible
404	Not Found	No se pudo encontrar el recurso
408	Request Timeout	El cliente tarda demasiado en enviar la petición

- **500 - 599:** códigos de error del servidor. Por ejemplo:

500	Internal Server Error	Error en el servidor
501	Not Implemented	El servidor no soporta la petición realizada
504	Gateway Timeout	Usado por servidores que actúan como proxies o gateways, indica que el servidor no obtuvo una respuesta a tiempo de un servidor remoto

1.1.5. Cookies

Las **cookies** son un mecanismo general mediante el que los programas de un servidor web pueden almacenar información en la parte del cliente de la conexión. Es una forma de añadir estado a las conexiones HTTP, aunque el manejo de cookies no es parte del protocolo HTTP, pero es soportado por la mayoría de los clientes.

Las cookies son objetos de tipo: *nombre = valor*, donde se asigna un *valor* determinado (una cadena de texto) a una variable del *nombre* indicado. Dicho objeto es almacenado y recordado por el servidor web y el navegador durante un período de tiempo (indicado como un parámetro interno de la propia *cookie*). Así, se puede tener una lista de *cookies* con distintas variables y distintos valores, para almacenar información relevante para cada usuario (se tienen listas de cookies independientes para cada usuario).

El funcionamiento es: el servidor, con la cabecera *Set-Cookie*, envía al cliente información de estado que éste almacenará. Entre la información se encuentra la descripción de los rangos de URLs para los que este estado es válido, de forma que para cualquier petición HTTP a alguna de esas URLs el cliente incluirá esa información de estado, utilizando la cabecera *Cookie*.

La sintaxis de la cabecera *Set-Cookie* es:

```
Set-Cookie: CLAVE1=VALOR1;...;CLAVEN=VALORN [OPCIONES]
```

donde OPCIONES es una lista opcional con cualquiera de estos atributos:

```
expires=FECHA;path=PATH;domain=DOMINIO;secure
```

- Las parejas de *CLAVE* y *VALOR* representan la información almacenada en la cookie
- Los atributos *domain* y *path* definen las URL en las que el navegador mostrará la cookie. *domain* es por defecto el *hostname* del servidor. El navegador mostrará la cookie cuando acceda a una URL que se empareje correctamente con ambos atributos. Por ejemplo, un atributo *domain="eps.ua.es"* hará que el navegador muestre la cookie cuando acceda a cualquier URL terminada en *"eps.ua.es"*. *path* funciona de forma similar, pero con la parte del path de la URL. Por ejemplo, el path *"/foo"* hará

que el navegador muestre la cookie en todas las URLs que comiencen por `"/foo"`.

- `expires` define la fecha a partir de la cual la cookie caduca. La fecha se indica en formato GMT, separando los elementos de la fecha por guiones. Por ejemplo:

```
expires=Wed, 09-Nov-1999 23:12:40 GMT
```

- `secure` hará que la cookie sólo se transmita si el canal de comunicación es seguro (tipo de conexión HTTPS).

Por otra parte, cuando el cliente solicita una URL que empareja con el dominio y path de alguna cookie, envía la cabecera:

```
Cookie: CLAVE1=VALOR1;CLAVE2=VALOR2;...;CLAVEN=VALORN
```

El número máximo de cookies que está garantizado que acepte cualquier navegador es de 300, con un máximo de 20 por cada servidor o dominio. El tamaño máximo de una cookie es de 4096 bytes.

1.1.6. Autenticaciones

Veremos ahora algunos mecanismos que pueden emplearse con HTTP para autenticar (validar) al usuario que intenta acceder a un determinado recurso.

Autenticaciones elementales

El protocolo HTTP incorpora un mecanismo de autenticación básico (**basic**) basado en cabeceras de autenticación para solicitar datos del usuario (el servidor) y para enviar los datos del usuario (el cliente), de forma que comprobando la exactitud de los datos se permitirá o no al usuario acceder a los recursos. Esta autenticación no proporciona confidencialidad ni integridad, sólo se emplea una codificación Base64.

Una variante de esto es la autenticación **digest**, donde, en lugar de transmitir el password por la red, se emplea un password codificado. Dicha codificación se realiza tomando el login, password, URI, método HTTP y un valor generado aleatoriamente, y todo ello se combina utilizando el método de encriptado MD5, muy seguro. De este modo, ambas partes de la comunicación conocen el password, y a partir de él pueden comprobar si los datos enviados son correctos. Sin embargo, algunos servidores no soportan este tipo de autenticación.

Certificados digitales y SSL

Las aplicaciones reales pueden requerir un nivel de seguridad mayor que el proporcionado por las autenticaciones *basic* o *digest*. También pueden requerir confidencialidad e integridad aseguradas. Todo esto se consigue mediante los **certificados digitales**.

- **Criptografía de clave pública:** La clave de los certificados digitales reside en la **criptografía de clave pública**, mediante la cual cada participante en el proceso tiene dos claves, que le permiten encriptar y desencriptar la información. Una es la clave

pública, que se distribuye libremente. La otra es la clave privada, que se mantiene secreta. Este par de claves es asimétrico, es decir, una clave sirve para descryptar algo codificado con la otra. Por ejemplo, supongamos que A quiere enviar datos encriptados a B. Para ello, hay dos posibilidades:

- A toma la clave pública de B, codifica con ella los datos y se los envía. Luego B utiliza su clave privada (que sólo él conoce) para descryptar los datos.
- A toma su clave privada, codifica los datos y se los envía a B, que toma la clave pública de A para descodificarlos. Con esto, B sabe que A es el remitente de los datos.

El encriptado con clave pública se basa normalmente en el algoritmo RSA, que emplea números primos grandes para obtener un par de claves asimétricas. Las claves pueden darse con varias longitudes; así, son comunes claves de 1024 o 2048 bits.

- **Certificados digitales:** Lógicamente, no es práctico teclear las claves del sistema de clave pública, pues son muy largas. Lo que se hace en su lugar es almacenar estas claves en disco en forma de **certificados digitales**. Estos certificados pueden cargarse por muchas aplicaciones (servidores web, navegadores, gestores de correo, etc).

Notar que con este sistema se garantiza la **confidencialidad** (porque los datos van encriptados), y la **integridad** (porque si los datos se descryptan bien, indica que son correctos). Sin embargo, no proporciona **autenticación** (B no sabe que los datos se los ha enviado A), a menos que A utilice su clave privada para encriptar los datos, y luego B utilice la clave pública de A para descryptarlos. Así, B descodifica primero el mensaje con su clave privada, y luego con la pública de A. Si el proceso tiene éxito, los datos se sabe que han sido enviados por A, porque sólo A conoce su clave privada.

- **SSL:** SSL (*Secure Socket Layer*) es una capa situada entre el protocolo a nivel de aplicación (HTTP, en este caso) y el protocolo a nivel de transporte (TCP/IP). Se encarga de gestionar la seguridad mediante criptografía de clave pública que encripta la comunicación entre cliente y servidor. La versión 2.0 de SSL (la primera mundialmente aceptada), proporciona autenticación en la parte del servidor, confidencialidad e integridad. Funciona como sigue:

- Un cliente se conecta a un lugar seguro utilizando el protocolo HTTPS (HTTP + SSL). Podemos detectar estos sitios porque las URLs comienzan con `https://`
- El servidor envía su clave pública al cliente.
- El navegador comprueba si la clave está firmada por un certificado de confianza. Si no es así, pregunta al cliente si quiere confiar en la clave proporcionada.

SSL 3.0 proporciona también soporte para certificados y autenticación del cliente. Funcionan de la misma forma que los explicados para el servidor, pero residiendo en el cliente.

1.2. Introducción a las aplicaciones web en Java

1.2.1. Qué es una aplicación web

Una aplicación web es una aplicación a la que accedemos mediante protocolo HTTP utilizando un navegador web. Hemos visto el protocolo HTTP, pero no cómo utilizarlo para implementar una aplicación.

Aplicaciones en el lado del servidor

En el lado del servidor, tenemos que conseguir que nuestro servidor HTTP sea capaz de ejecutar programas de aplicación que recojan los parámetros de peticiones del cliente, los procesen y devuelvan al servidor un documento que éste pasará a su vez al cliente.

Así, para el cliente el servidor no habrá hecho nada distinto a lo estipulado en el protocolo HTTP, pero el servidor podrá valerse de herramientas externas para procesar y servir la petición solicitada, pudiendo así no limitarse a servir páginas estáticas, sino utilizar otras aplicaciones (servlets, JSP, PHP, etc) para servir documentos con contenido dinámico.

Los programas de aplicación son típicamente programas que realizan consultas a bases de datos, procesan la información resultante y devuelven la salida al servidor, entre otras tareas.

Vamos a centrarnos en las aplicaciones web JavaEE, en las que los componentes dinámicos que recibirán las peticiones HTTP en el servidor serán los servlets y JSPs. Estos componentes podrán analizar esta petición y utilizar otros componentes Java para realizar las acciones necesarias (beans, EJBs, etc).

Aplicaciones en el lado del cliente

Se tienen muchas tecnologías relacionadas con extensiones del lado del cliente (entendiendo cliente como un navegador que interpreta código HTML). El código HTML es un código estático que sólo permite formatear la apariencia de una página y definir enlaces a otras páginas o URLs. Esto no es suficiente si queremos que el navegador realice funciones más complicadas: validar entradas de formularios, mostrar la evolución del precio de unas acciones, etc.

Para ampliar las funcionalidades del navegador (respetando el protocolo HTTP), se utilizan tecnologías como JavaScript, Applets, Flash, etc. Estas se basan en hacer que el navegador ejecute código que le pasa el servidor, bien embebido en documentos HTML (como es el caso de JavaScript), o bien mediante ficheros compilados multiplataforma (como es el caso de los Applets Java o los ficheros Flash).

1.2.2. Estructura de una aplicación web JavaEE

Una aplicación web JavaEE que utilice servlets o páginas JSP debe tener una estructura de ficheros y directorios determinada:

- En el directorio raíz de la aplicación se colocan las páginas HTML o JSP (podemos

- dividir las también en directorios si queremos)
- Colgando del directorio inicial de la aplicación, se tiene un directorio **WEB-INF**, que contiene la información Web relevante para la aplicación. Esta información se divide en:
 - Fichero **descriptor de despliegue** de la aplicación: es el fichero descriptor de la aplicación web. Es un fichero XML (llamado `web.xml`) que contiene información genérica sobre la aplicación. Lo veremos con más detalle más adelante
 - Subdirectorio **classes**: en él irán todas las clases Java utilizadas en la aplicación (ficheros `.class`), es decir, clases externas a la API de Java que se utilicen en las páginas JSP, servlets, etc. Las clases deberán mantener la estructura de paquetes, es decir, si queremos colocar la clase `paquetel.subpaquetel.MiClase` dentro de `classes`, se quedará almacenada en el directorio `classes/paquetel/subpaquetel/MiClase`.
 - Subdirectorio **lib**: aquí colocaremos las clases Java que estén empaquetadas en ficheros JAR (es decir, colocaremos los ficheros JAR de nuestra aplicación Web, y las librerías ajenas a la API de JDK o de servlets y JSP que se necesiten)
 - El resto de elementos de la aplicación (imágenes, etc), podemos estructurarlos como nos convenga.

Notar que se separan los ficheros `.class` de los ficheros JAR, colocando los primeros en el directorio `classes` y los segundos en `lib`.

Esta estructura estará contenida dentro de algún directorio, que será el directorio correspondiente a la aplicación Web, y que podremos, si lo hacemos convenientemente, copiar en el servidor que nos convenga. Es decir, cualquier servidor Web JavaEE soporta esta estructura en una aplicación Web, sólo tendremos que copiarla en el directorio adecuado de cada servidor.

Cada aplicación web JavaEE es un contexto, una unidad que comprende un conjunto de recursos, clases Java y su configuración. Cuando hablemos de contexto, nos estaremos refiriendo a la aplicación web en conjunto. Por ello utilizaremos indistintamente los términos aplicación web y contexto.

Rutas relativas al contexto

Cada contexto (aplicación web) instalado en el servidor tendrá asociado una ruta para acceder a él desde la web. Por ejemplo, podemos asociar nuestro contexto la ruta `/aplic`, de forma que accediendo a la siguiente URL:

```
http://localhost:8080/aplic/index.htm
```

Estaremos leyendo el recurso `/index.htm` de nuestro contexto.

Supongamos que tenemos alguna imagen o recurso al que queremos acceder desde otro, en nuestra aplicación Web. Por ejemplo, supongamos que colgando del directorio raíz de la aplicación tenemos la imagen `miImagen.jpg` dentro de la carpeta `imagenes` (es decir,

`imagenes/miImagen.jpg`).

Podemos acceder a esta imagen de varias formas, aunque a veces podemos tener problemas con alguna, porque luego el contenedor Web tome la ruta relativa al lugar desde donde queremos cargar la imagen (o recurso, en general). Este problema lo podemos tener accediendo a elementos desde servlets, sobre todo.

Una solución para evitar esto es acceder a todos los elementos de la aplicación a partir de la ruta del contexto. Por ejemplo, si nuestro contexto tiene la ruta `/aplic` asociada, para acceder a la imagen desde una página HTML, pondríamos:

```

```

1.2.3. Empaquetamiento de aplicaciones web: ficheros WAR

Una forma de distribuir aplicaciones Web es empaquetar toda la aplicación (a partir de su directorio inicial) dentro de un fichero WAR (de forma parecida a como se hace con un TAR o un JAR), y distribuir dicho fichero. Podemos crear un fichero WAR de la misma forma que creamos un JAR, utilizando la herramienta JAR.

Estos ficheros WAR son un estándar de JavaEE, por lo que podremos utilizarlos en los diferentes servidores de aplicaciones JavaEE existentes.

Por ejemplo, si tenemos en el directorio `web/ejemplo` los siguientes ficheros:

```
web/ejemplo/
    index.html
    WEB-INF/
        web.xml
        classes/
            ClaseServlet.class
```

Para crear la aplicación WAR se siguen los pasos:

- Crear el WAR colocándonos en dicho directorio `web/ejemplo` y escribiendo:

```
jar cMvf ejemplo.war *
```

Las opciones `c`, `v` y `f` son para crear el WAR como un JAR comprimido normal. La opción `M` (mayúscula) es para que no se añada el fichero `MANIFEST`.

También es **IMPORTANTE** destacar que no debe haber subdirectorios desde la raíz de la aplicación, es decir, la estructura del fichero WAR debe ser:

```
index.html
WEB-INF/
    web.xml
    classes/
        ClaseServlet.class
```

sin ningún subdirectorio previo (ni `ejemplo/` ni `web/ejemplo/` ni nada por el estilo).

- Copiar el fichero WAR al servidor web para poner en marcha la aplicación. Veremos

esto con detalle en el siguiente apartado.

1.3. Despliegue de archivos WAR

Vamos a ver las diferentes formas de desplegar un archivo WAR en Tomcat. El empaquetamiento en archivos WAR es algo estándar, pero no así el proceso de despliegue, que es dependiente del servidor. No obstante, la mayoría de servidores JavaEE funcionan en este aspecto de modo similar: permiten desplegar las aplicaciones desde una consola de administración y también "dejando caer" el fichero en determinado directorio.

1.3.1. Copiar el .WAR a Tomcat

En Tomcat la forma más simple de desplegar un archivo .war consiste en "dejarlo caer" en el directorio `webapps`. Con la configuración por defecto, Tomcat examina periódicamente el directorio y si aparece en él algún archivo nuevo lo descomprime automáticamente en el directorio y "activa" la aplicación en el servidor. Esto se conoce en Tomcat como `autodeploy`, y se puede desactivar en la configuración si así lo deseamos.

La URL por defecto para acceder a la aplicación será el nombre del .war. Es decir, que si nuestro fichero se llama `ejemplo.war`, Tomcat lo descomprimirá creando un subdirectorio llamado `ejemplo` y desde el navegador accederemos con URLs del tipo `http://nombre_del_host:8080/ejemplo/...`

Nota

Con esta forma de despliegue, Tanto la URL como otros aspectos pueden configurarse en un archivo `context.xml` que colocaremos en determinado directorio dentro del .war. En temas posteriores veremos detalladamente esta configuración.

Podemos eliminar una aplicación sin más que borrar el directorio donde tomcat la ha descomprimido.

Aviso

Bajo Windows no se puede borrar directamente el directorio de la aplicación porque el sistema operativo interpreta que Tomcat lo está usando y no lo permite. Para permitir el borrado es necesario activar el *anti-locking* en la configuración de Tomcat.

1.3.2. La aplicación "manager" de Tomcat

Tomcat incluye una aplicación llamada `manager`, que nos permitirá desplegar y gestionar las aplicaciones web instaladas en el servidor en tiempo de ejecución. Con el *manager* podremos subir y desplegar una aplicación, ver la lista de aplicaciones desplegadas, y detener, recargar, reanudar o desinstalar estas aplicaciones.

El manager de Tomcat cuenta con una interfaz HTML desde la cual podremos desplegar aplicaciones y gestionar las aplicaciones instaladas. Para acceder a esta interfaz HTML del manager introduciremos la siguiente URL en cualquier navegador:

```
http://localhost:8080/manager/html
```



Nos pedirá usuario y password. Los autorizados para acceder a esta aplicación deben tener rol de manager. Las versiones de Tomcat existentes en el momento de imprimir estas páginas ya tienen asignado este rol al usuario admin.

En versiones antiguas de Tomcat no había un usuario con rol de manager creado por defecto, por tanto si usamos una de estas versiones tendremos que crearlo a mano. Para ello editaremos el fichero `${tomcat.home}/conf/tomcat-users.xml` e introduciremos las siguientes líneas:

```
<role rolename="manager"/>
<user username="admin" password="JavaEE" roles="manager"/>
```

Con esto ya podremos acceder al manager con nuestro usuario. En este caso el usuario tendrá el nombre admin y el password JavaEE.

Una vez accedamos al manager veremos una página como la que se muestra a continuación:

Gestor de Aplicaciones Web de Tomcat

Mensaje: OK

Gestor

[Listar Aplicaciones](#)
 [Ayuda HTML de Gestor](#)
 [Ayuda de Gestor](#)
 [Estado de Servidor](#)

Aplicaciones

Trayectoria	Nombre a Mostrar	Ejecutiéndose	Sesiones	Comandos
/	Welcome to Tomcat	true	0	Arrancar Parar Recargar Replegar <input type="button" value="Expire sessions"/> with idle ≥ <input type="text" value="30"/> minutes
/docs	Tomcat Documentation	true	0	Arrancar Parar Recargar Replegar <input type="button" value="Expire sessions"/> with idle ≥ <input type="text" value="30"/> minutes
/host-manager	Tomcat Manager Application	true	0	Arrancar Parar Recargar Replegar <input type="button" value="Expire sessions"/> with idle ≥ <input type="text" value="30"/> minutes
				Arrancar Parar Recargar Replegar

Vista del manager de Tomcat

Aquí podemos ver las aplicaciones instaladas en el servidor y podemos gestionarlas. Podemos detener (*Stop*) las aplicaciones para que dejen de estar disponibles, pero sin borrarlas del servidor, y posteriormente reanudar su ejecución con *Start*. También

podemos recargar las aplicaciones con *Reload*. Esto será útil cuando hayamos modificado la aplicación y queramos que Tomcat reconozca estos cambios, por ejemplo si hemos cambiado la configuración de la aplicación (`web.xml`) o hemos añadido o modificado clases Java. Por último, con *Remove* podremos desinstalar la aplicación del servidor. Al hacer esto se eliminarán todos los ficheros de la aplicación y ya no podrá reanudarse.

En la parte inferior de esta página encontramos los siguientes formularios:

Desplegar

Desplegar directorio o archivo WAR localizado en servidor

Trayectoria de Contexto (opcional):

URL de archivo de Configuración XML:

URL de WAR o Directorio:

Archivo WAR a desplegar

Seleccione archivo WAR a cargar

Desde aquí podremos desplegar aplicaciones web en el servidor. Con el formulario superior podremos desplegar una aplicación que ya se encuentre en un directorio de la máquina en la que está el servidor.

Con el formulario inferior será muy sencillo desplegar una aplicación web. Simplemente necesitamos tener el fichero WAR de la aplicación en nuestra máquina. Pulsamos sobre *Examinar...* para buscar y seleccionar este fichero WAR, y una vez seleccionado pulsaremos sobre *Desplegar* para que suba y despliegue la aplicación al servidor web.

2. Ejercicios de protocolo HTTP e introducción a las aplicaciones web

2.1. Pruebas con protocolo HTTP (1 punto)

En las plantillas de la sesión tenéis un par de archivos JAR que implementan un cliente y un servidor HTTP de "pruebas". Estos únicamente establecen la conexión y nos permiten escribir manualmente los comandos HTTP y ver la respuesta del otro lado. Crea un fichero `ejer1_1.txt` con lo que vayas haciendo en este ejercicio (las respuestas que obtengas del servidor o las peticiones del cliente, y las respuestas a las preguntas que se te plantean).

Utilizando el programa que simula ser un cliente HTTP (`ClienteHTTP.jar`) vamos a abrir la página principal de la Universidad de Alicante.

1. Ejecutar el programa desde un terminal colocándose en el directorio en que reside y tecleando

```
java -jar ClienteHTTP.jar
```

2. En el campo `host` deberéis indicar `www.ua.es`, y pulsar el botón "Conectar"
3. Como petición teclear

```
GET / HTTP/1.0
(línea en blanco)
```

NOTA: es importante respetar la línea en blanco al final de la petición

4. Comprobad que se recibe el HTML de la página principal de la UA comparando con el código fuente de la misma página vista desde un navegador.
5. Forzad a que el servidor os envíe la página en valenciano repitiendo la petición pero ahora adjuntando la cabecera `Accept-language` con el valor `ca` (correspondiente al locale catalán/valenciano, según la [lista de códigos ISO 639-1](#))

Utilizando el programa que simula ser un servidor HTTP que tenéis en la plantilla (`ServidorHTTP.jar`), hacer lo siguiente:

1. Ejecutar el programa desde un terminal con

```
java -jar ServidorHTTP.jar
```

2. Escribir como número de puerto 1080, 8080 o cualquier valor por encima de 1024 (ya que en linux los usuarios comunes no tienen permiso para crear sockets de servidor por debajo de este puerto). Pulsar sobre "Esperar petición".
3. Utilizar un navegador para hacer un petición HTTP a nuestro servidor de prueba. Pondremos una URL del estilo: `http://localhost:1080/index.htm` (evidentemente hay que usar el mismo puerto que hemos puesto en el servidor).
4. Observar en el servidor espía la petición que ha hecho el navegador. ¿Cómo se identifica el navegador (versión + sistema operativo)?

5. Abrir la página HTML `form_get.htm` con cualquier navegador web. Introducir datos en el formulario y enviar una petición. Esta petición se estará realizando al servidor espía instalado. Observar en este servidor la petición realizada. ¿Dónde se han enviado los datos del formulario?
6. Vamos a hacer lo mismo que en el apartado anterior, pero con la página `form_post.htm`. ¿Qué diferencia hay entre esta petición y la realizada en el caso anterior? ¿Dónde se envían los datos introducidos en el formulario?

2.2. Recursos estáticos en Tomcat (0,5 puntos)

Vamos a probar el servidor web Tomcat y a subir recursos estáticos a él para comprobar que funciona. Nota: en la entrega debes adjuntar

- Tomcat está instalado en el directorio `/opt/apache-tomcat-<nº de version>`. Ponerlo en marcha ejecutando el script `startup.sh` de su directorio `bin`. Aparecerán mensajes indicando las variables de entorno que está usando:

```
Using CATALINA_BASE:   /opt/apache-tomcat-6.0.29
Using CATALINA_HOME:   /opt/apache-tomcat-6.0.29
Using CATALINA_TMPDIR: /opt/apache-tomcat-6.0.29/temp
Using JRE_HOME:        /opt/jdk1.6.0_21
Using CLASSPATH:       /opt/apache-tomcat-6.0.29/bin/bootstrap.jar
```

Comprobar que Tomcat ha arrancado accediendo a `http://localhost:8080/` desde cualquier navegador. Debería aparecer la página principal de Tomcat

- Copiar el fichero `pagina.htm` (podrás encontrarlo en las plantillas de la sesión) a la aplicación raíz (`webapps/ROOT` dentro de Tomcat), que es la que se ejecuta por defecto (la página principal de Tomcat es parte de ella). Comprobar que se puede acceder correctamente a este recurso introduciendo la siguiente URL

```
http://localhost:8080/pagina.htm
```

. Incluye un volcado de la pantalla del navegador en la entrega con el nombre `ejer1_2.jpg`

2.3. Desplegar una aplicación web sencilla sin Eclipse (1 punto)

En las plantillas de la sesión hay una sencilla aplicación web que permite mantener una página y que los visitantes puedan añadir comentarios. Vamos a empaquetar y desplegar la aplicación en Tomcat

Nota: en la entrega incluye todos los ficheros que generes dentro de una carpeta `ejer1_3` (el `.war` y el `tomcat-users.xml`)

Empaquetamiento en .war. Observar la estructura de directorios que cuelga del directorio `comentarios`. Podemos empaquetar la aplicación mediante la herramienta `jar` incluida en el JDK, como se especifica en los apuntes:

```
jar cvf comentarios.war *
```

La operación anterior hay que hacerla **desde dentro del directorio comentarios**, es decir, el .war creado no debe contener la carpeta "comentarios" propiamente dicha, sino lo que contiene ésta.

Si no tenemos la herramienta `jar` en el *path*, podemos comprimir los archivos en un .zip convencional y cambiarle manualmente la extensión.

Despliegue manual en Tomcat Para desplegar el .war manualmente basta con dejarlo en la carpeta `webapps` del directorio de instalación de Tomcat. Observar que transcurridos unos segundos el .war se descomprime automáticamente en una carpeta con el mismo nombre. Para probar la aplicación abre un navegador y accede a la URL

```
http://localhost:8080/comentarios/comentarios.jsp
```

Repliegue de la aplicación: para eliminar la aplicación del servidor basta con borrar el .war y **también** el directorio descomprimido.

Despliegue a través del *manager* de Tomcat: Esta sería la forma de trabajar si no estuviéramos en la misma máquina que el servidor. Para ello, exportamos a un .WAR en cualquier directorio (que no sea de Tomcat).

Crear un nuevo usuario con rol "manager" en el fichero de usuarios de Tomcat (`conf/tomcat-users.xml`): introducir las líneas:

```
<role rolename="manager" />
<user username="manager" password="manager" roles="manager" />
```

IMPORTANTE: tendrás que rearrancar Tomcat para que el cambio tenga efecto. Páralo ejecutando el script `bin/shutdown.sh` y vuelve a ejecutarlo con `bin/startup.sh`

El acceso al *manager* se hace a través de la URL

```
http://localhost:8080/manager/html
```

Tomcat nos pedirá un usuario con rol manager. Introducir el usuario de administración. El .war se despliega con el formulario que aparece al final de la página. Una vez desplegado, comprobar que la aplicación aparece en la lista de aplicaciones activas. Comprobar también que si la aplicación se recarga a través del manager, los comentarios se pierden (ya que por el momento la aplicación los guarda en memoria del servidor).

3. Configuración de Aplicaciones Web en Tomcat. Desarrollo con Eclipse WTP

En esta sesión veremos primero cómo es el ciclo de desarrollo de aplicaciones web con Eclipse WTP, que es el conjunto de plugins para desarrollar aplicaciones web que es "estándar" de Eclipse. Luego veremos cómo configurar Apache Tomcat para ejecutar nuestras aplicaciones. Una vez configuradas las características comunes a todas las aplicaciones veremos cómo configurar los aspectos propios de cada aplicación. De éstos hay algunos que son estándares e independientes del servidor, pero hay otros particulares a Tomcat.

3.1. Desarrollo y despliegue con WebTools

WebTools es un plugin ya integrado con Eclipse, que permite gestionar aplicaciones web como proyectos, teniéndolo todo integrado y a mano para poder empaquetar, probar y desplegar la aplicación de forma cómoda. Para ello, desde WebTools podremos, entre otras cosas:

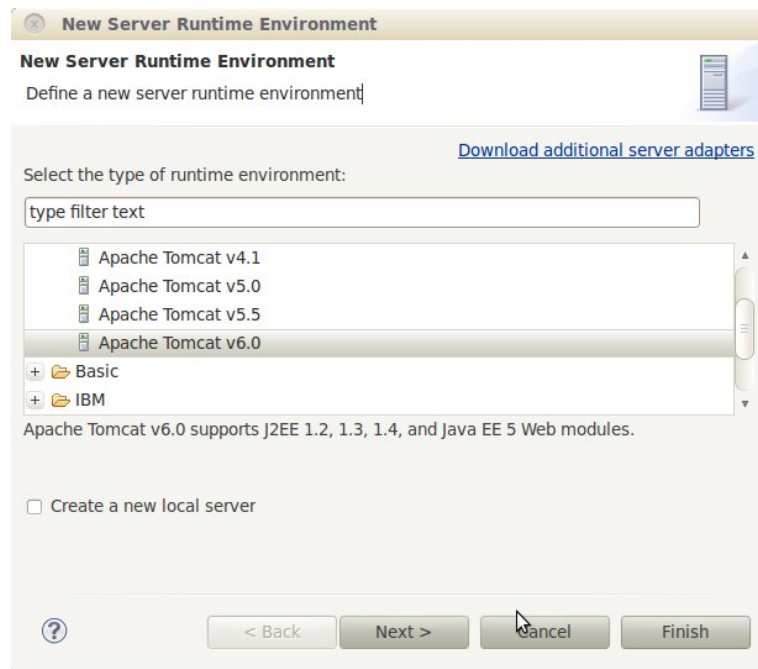
- Gestionar el servidor web sobre el que queremos desplegar y/o probar la aplicación, pudiendo editar su configuración, y pararlo y reanudarlo cuando lo necesitemos, desde el propio Eclipse
- Crear y desarrollar las aplicaciones web que vayamos a desplegar. Para cada una, Eclipse nos organizará los ficheros según la estructura que digamos (carpeta de fuentes Java, carpeta WEB-INF, librerías, etc)
- Desplegar y testear las aplicaciones sobre el servidor tantas veces como queramos, manteniendo siempre el control sobre su ejecución desde Eclipse

Para trabajar con este tipo de elementos, trabajaremos desde la perspectiva Java EE de Eclipse. Veremos cómo realizar estas tareas a continuación.

3.1.1. Gestión de servidores web con WebTools

En WTP hay que dar de alta tanto el "Server runtime" que es como el tipo genérico de servidor que vamos a usar como el "Server", que es una instancia concreta de ese runtime. Podemos tener varias instancias del mismo runtime.

Para dar de alta en Eclipse el servidor web sobre el que queremos trabajar, vamos a *Window - Show View - Servers*, para abrir la vista de Servidores, y después, pinchando sobre ella con el botón derecho, vamos a *New - Server*. Podemos llegar igualmente desde el menú *File > New > Server*. Después, en la ventana que aparece, simplemente rellenamos los datos del nombre del host (por defecto, *localhost*), y el tipo de servidor web que usamos. Si no hemos dado de alta ningún runtime, podemos hacerlo ahora. En el caso de tomcat se nos pedirá el directorio de instalación, entre otros datos.

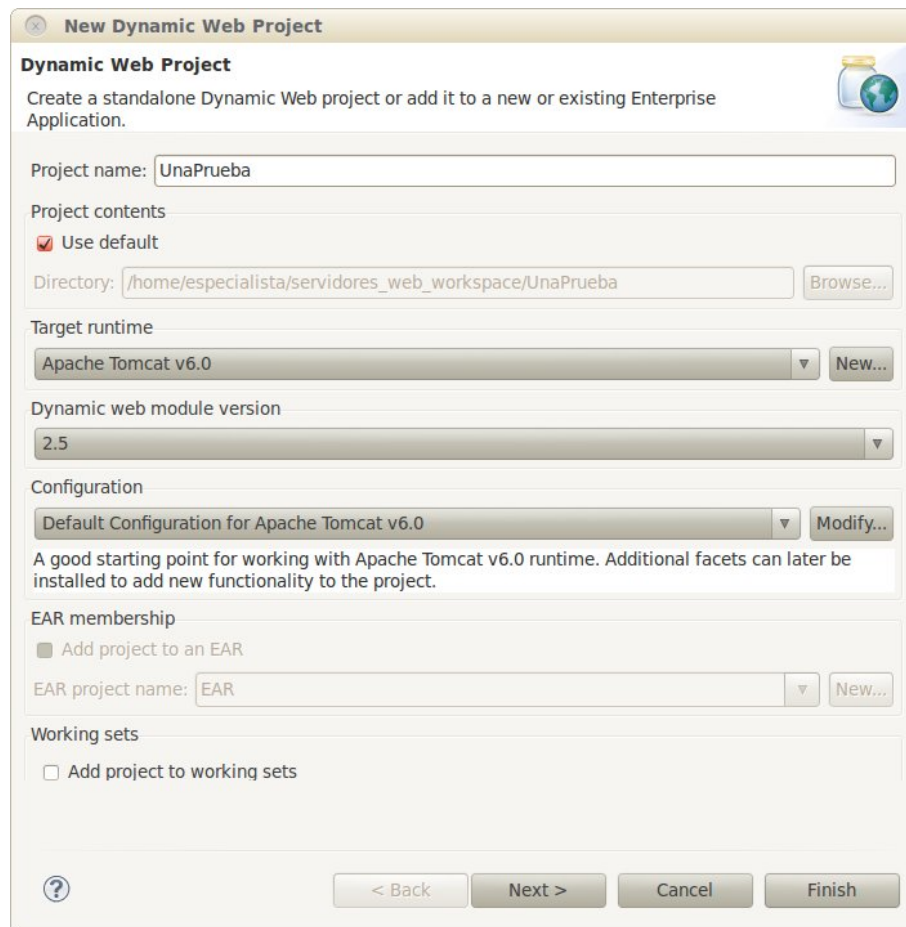


Configuración del servidor web en WebTools

Después, en la vista *Servers* (*Show > View > Servers*) ya tendremos añadido el nuevo servidor. Haciendo click con el botón derecho sobre él, accedemos a las opciones para pararlo, reanudarlo, asociarle aplicaciones, etc

3.1.2. Creación y desarrollo de una aplicación web con WebTools

Para crear un proyecto de aplicación web desde WebTools, vamos al menú *File - New - Project...* - *Web - Dynamic Web Project*



Creación del proyecto web

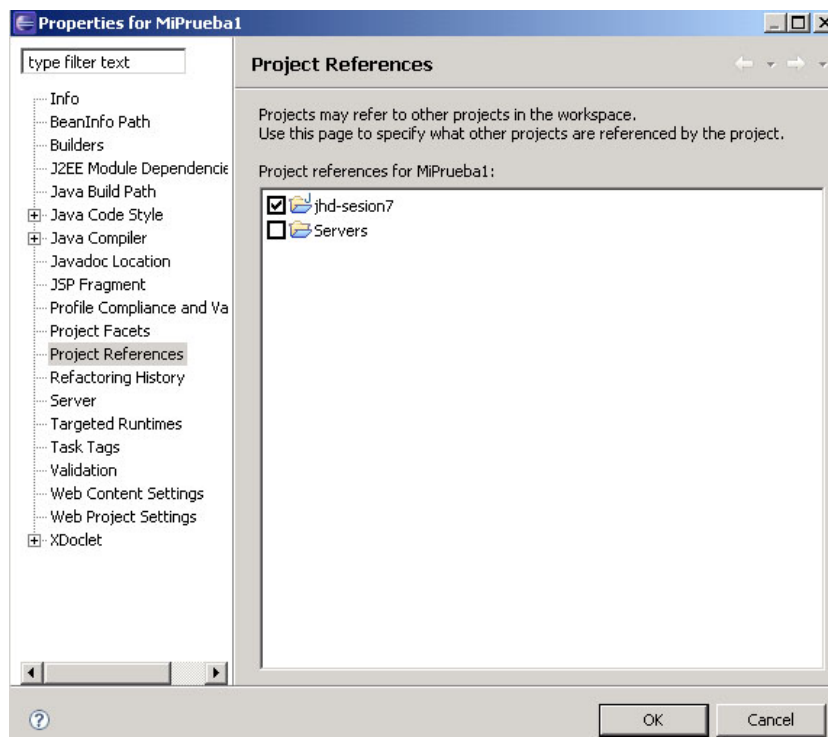
En los siguientes pasos del asistente podemos elegir su configuración, y las carpetas donde alojar contenido web y fuentes Java. La **estructura de carpetas** que crea WTP será la siguiente:

- Carpeta **src**: carpeta de fuentes, con todas las clases Java y paquetes que hagamos para nuestra aplicación (clases auxiliares, servlets, etc)
- Carpeta **build**, donde se colocará el código compilado.
- Carpeta **WebContent**: con la estructura de la aplicación web. Como ya vimos en la sesión anterior esta estructura es estándar. En su carpeta raíz podremos colocar las páginas HTML / JSP (organizadas en subcarpetas si queremos)
 - Subcarpeta **WebContent/WEB-INF**: con el fichero descriptor de la aplicación (`web.xml`)
 - Subcarpeta **WebContent/WEB-INF/lib**: con las librerías JAR que necesite nuestra aplicación
 - Subcarpeta **WebContent/WEB-INF/classes**: inicialmente vacía, en ella se copiarán después automáticamente los ficheros fuente de `src`

- Además verás una subcarpeta **WebContent/META-INF** que es propia de Tomcat, y no encontrarás en otros proyectos web que vayas a desplegar en otros servidores. Luego veremos qué se puede configurar en dicha carpeta.

Interdependencias entre proyectos

En ocasiones deberemos relacionar algún proyecto de clases Java que tengamos hecho, con un nuevo proyecto web que vayamos a hacer, para poder utilizar e incorporar las clases del proyecto Java en nuestro proyecto web. Para ello, vamos con el botón derecho del ratón sobre el proyecto, y vamos a sus *Properties*. En ellas, vamos a *Project References*, y marcamos con el ratón los proyectos que queramos asociar al actual:



Asociaciones entre proyectos

3.1.3. Despliegue de aplicaciones web con WebTools

Una vez tengamos la aplicación web lista para probar en el servidor web, tenemos dos alternativas para desplegarla y probarla:

- Utilizar las facilidades de despliegue sobre el servidor que ofrece WebTools. Esta opción la utilizaremos en fase de desarrollo y depuración, para probar de forma cómoda y rápida los cambios que vayamos haciendo sobre nuestra aplicación.
- Usar alguna herramienta adicional como Ant o Maven, como veremos en sesiones posteriores. Esto será necesario cuando el servidor sea remoto

Para poder desplegar la aplicación utilizando WebTools, simplemente tenemos que pinchar con el botón derecho sobre el proyecto web y elegir la opción *Run As - Run on Server*. Inicialmente, ya le hemos asignado el servidor al proyecto, cuando lo creamos, así que WebTools ya sabe sobre qué servidor desplegar los cambios.

3.2. Configuración de Tomcat

Para comprender mejor la configuración de Tomcat antes hay que ver cuál es su arquitectura, tanto desde el punto de vista lógico como físico.

3.2.1. Estructura física y lógica de Tomcat

Estructura física

La distribución de Tomcat está dividida en los siguientes directorios:

- `bin`: ejecutables y scripts para arrancar y parar Tomcat.
- `common`: clases y librerías compartidas entre Tomcat y las aplicaciones web. Las clases se deben colocar en `common/classes`, mientras que las librerías en formato JAR se deben poner en `common/lib`.
- `conf`: ficheros de configuración.
- `logs`: directorio donde se guardan por defecto los logs.
- `server`: las clases que componen Tomcat.
- `shared`: clases compartidas por todas las aplicaciones web.
- `webapps`: directorio usado por defecto como raíz donde se colocan todas las aplicaciones web.
- `work y temp`: directorios para almacenar información temporal

Los logs del servidor

Además del *log* de accesos, en Tomcat podemos tener otros ficheros de *log* cuya finalidad primordial es registrar eventos del servidor y de las aplicaciones para poder realizar una depuración en caso de que se produzca algún error.

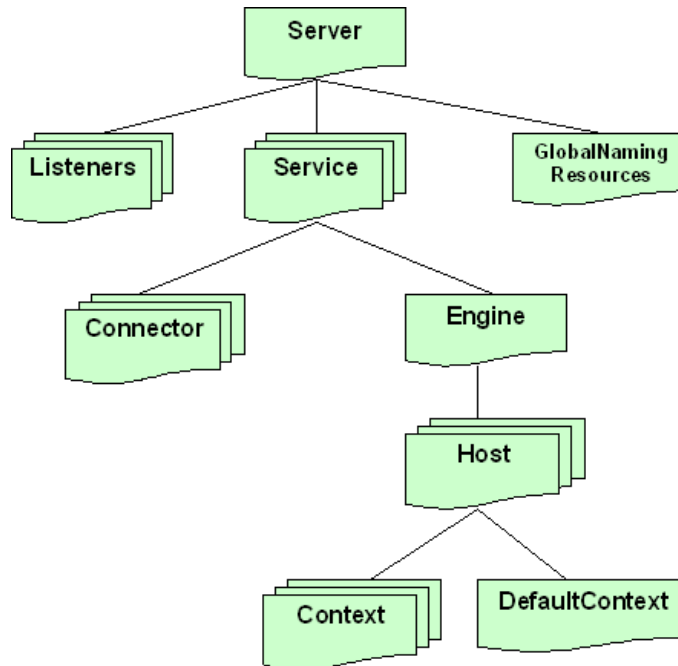
Estos ficheros se encuentran dentro de la carpeta `logs`. Algunos ya vienen por defecto. Dependiendo de la versión de Tomcat, podemos encontrarnos uno o varios de los siguientes:

- `catalina.aaaa-mm-dd.log`: *logger* global definido al nivel del engine.
- `localhost.aaaa-mm-dd.log`: *logger* global a todas las aplicaciones del host `localhost`.
- `manager.aaaa-mm-dd.log`: *logger* que utiliza el manager de Tomcat.
- `admin.aaaa-mm-dd.log`: *logger* que utiliza la aplicación de administración.

En estos ficheros podremos encontrar mensajes sobre el funcionamiento general y errores generales del servidor Tomcat.

Estructura lógica: módulos

Tomcat está compuesto por una serie de módulos cuyo comportamiento es altamente configurable. Incluso se pueden cambiar las clases que utiliza Tomcat por clases propias modificando el fichero `server.xml`. La estructura general de dichos módulos se muestra en la siguiente figura.



Cada módulo viene representado por su correspondiente etiqueta en el fichero `server.xml`.

- **Server:** es el propio Tomcat. Solo existe una instancia de este componente, que a su vez contiene a todos los demás elementos y subelementos.
- **Listener:** monitoriza la creación y eliminación de contenedores web
- **GlobalNamingResources:** sirve para definir mapeados de JNDI globales a todas las aplicaciones. Por ejemplo, para definir métodos de conexión a bases de datos.
- **Service:** un objeto de este tipo representa el sistema formado por un conjunto de conectores (`connector`) que reciben las peticiones de los clientes y las pasan a un `engine`, que las procesa. Por defecto viene definido el servicio llamado `Tomcat-Standalone`.
- **Connector:** acepta ciertos tipos de peticiones para pasarlas al `engine`. Por defecto, Tomcat incorpora un conector HTTP/1.1 (sin SSL) por el puerto 8080, y otro para comunicación con otros servidores (como Apache). Para cambiar el puerto por el que Tomcat acepta las peticiones HTTP basta con cambiar el atributo `port` de dicho conector.
- **Engine:** representa al contenedor web.
- **Host:** representa un host (o un host virtual). Mediante `appBase` se especifica el

- directorio de donde "colgarán" las aplicaciones web (por defecto `webapps`)
- **Context:** representa una aplicación web. Veremos de manera más detallada su configuración.
- **DefaultContext:** se aplica por defecto a aquellas aplicaciones que no tienen `context` propio.

Hay una serie de elementos que se pueden definir a varios niveles, de modo que por ejemplo pueden afectar a todo el servidor o solo a una aplicación web:

- **Valve:** es un componente que puede "filtrar" las peticiones.
- **Logger:** define el funcionamiento de los *logs* para depuración de errores (no los *logs* de acceso, que se definen mediante un `valve`).
- **Realm:** define un conjunto de usuarios con permisos de acceso a un determinado contexto.
- **Manager:** implementa el manejo de sesiones HTTP. Se puede configurar para que las sesiones se almacenen de manera permanente cuando se apaga el servidor.
- **Loader:** cargador de clases para una aplicación web. Es raro usar uno distinto al que viene por defecto.

3.2.2. Formas de cambiar la configuración

La configuración de Tomcat está almacenada en cuatro ficheros que se encuentran en el directorio `conf`. Tres de ellos están en formato XML y el cuarto es un fichero de políticas de seguridad en el formato estándar de Java:

- `server.xml`: el fichero principal de configuración.
- `web.xml`: es un fichero en el formato estándar para aplicaciones web con servlets, que contiene la configuración global a todas las aplicaciones.
- `tomcat-users.xml`: lista de usuarios y contraseñas para autenticación.
- `catalina.policy`: políticas de seguridad para la ejecución del servidor.

Además se puede cambiar gran parte de la configuración a través de la aplicación de administración.

3.2.3. Configurar el host

Mediante el elemento `Host` se define la configuración para un host o host virtual

```
<Host name="localhost" debug="0" appBase="webapps" u
    npackWARs="true" autoDeploy="true">...
```

Algunos de los principales atributos de este elemento son los siguientes:

Atributo	Significado	Valor por defecto
<code>name</code>	nombre del host o host virtual	ninguno
<code>debug</code>	nivel de mensajes de depuración	0

appBase	directorio donde se instalarán las aplicaciones de este host (si es relativa, la ruta se supone con respecto al directorio de Tomcat)	ninguno
unpackWARs	si es true, las aplicaciones empaquetadas en WAR se desempaquetan antes de ejecutarse	true
autoDeploy	si es true, se utiliza el despliegue automático de aplicaciones	true
liveDeploy	si es true, se utiliza despliegue automático sin necesidad de rearrancar Tomcat	true

El despliegue automático de aplicaciones es una interesante característica que permite instalarlas sin más que dejar el WAR o la estructura de directorios de la aplicación dentro del directorio appBase. Esta característica se puede desactivar poniendo autoDeploy y liveDeploy a false. Veremos más sobre ello en el punto siguiente.

3.2.4. Configuración en Eclipse WebTools

Al ejecutar aplicaciones mediante WebTools, Eclipse está empleando una copia de los ficheros de configuración de Tomcat, de modo que puedan cambiarse sin afectar a la instalación del servidor. Dicha configuración es accesible a través de la carpeta Servers que aparece en Eclipse como si fuera un proyecto más. Por cada *runtime* de servidor web configurado en Eclipse tendremos una subcarpeta dentro de Servers, en la que aparecerán los ficheros de configuración del servidor que ya hemos visto: server.xml, tomcat-users.xml,... Aunque no es necesario, ni recomendable salvo que sepamos con seguridad lo que estamos haciendo, podemos modificar manualmente estos archivos si necesitamos una configuración más flexible que la que ofrecen las opciones de WebTools.

3.3. Configuración de aplicaciones web en Tomcat

Una vez visto cómo cambiar aspectos comunes a todas las aplicaciones, veamos cómo configurar cada aplicación de modo individual. El mecanismo estándar en aplicaciones web JavaEE es el descriptor de despliegue, aunque Tomcat aporta algo de funcionalidad extra a través del *contexto*.

3.3.1. El descriptor de despliegue

Como hemos dicho anteriormente, el directorio `WEB-INF` de una aplicación web con servlets y/o páginas JSP, debe haber un fichero descriptor de despliegue (llamado `web.xml`) que contenga la información relativa a la aplicación.

El `web.xml` es estándar en JavaEE y por tanto todo lo visto en esta sección es igualmente aplicable a cualquier servidor compatible JavaEE, aunque no sea Tomcat.

Es un fichero XML, que comienza con una cabecera XML que indica la versión y la codificación de caracteres, y un DOCTYPE que indica el tipo de documento, y la especificación de servlets que se sigue. La etiqueta raíz del documento XML es `web-app`. Así, un ejemplo de fichero podría ser:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>
    Mi Aplicacion Web
  </display-name>
  <description>
    Esta es una aplicacion web sencilla a modo de
ejemplo
  </description>
</web-app>
```

En este caso se está utilizando la especificación 2.5 de servlets. Algunos servidores permiten omitir la cabecera `xml` y el DOCTYPE, pero sí es una buena costumbre el ponerlas.

Dentro de la etiqueta raíz `<web-app>` podemos colocar otros elementos que ayuden a establecer la configuración de nuestra aplicación web. Veremos a continuación algunos de ellos. En algunos elementos profundizaremos un poco más, por tratarse de elementos genéricos de una aplicación web (variables globales, etc). En otros (servlets, filtros, etc), simplemente se indicará qué elementos se tratan, pero su configuración se explicará en temas más específicos.

A partir de la versión 2.4 de la especificación (Tomcat 5.5), el orden de las etiquetas dentro del archivo es libre. No obstante, las discutiremos en el orden que había que seguir hasta esta versión.

1. Información general de la aplicación

Primero tenemos etiquetas con información general:

- `<display-name>`: nombre con que deben utilizar las aplicaciones gráficas para referenciar a la aplicación
- `<description>`: texto descriptivo de la aplicación

Variables globales

Podemos tener varias etiquetas:

- **<context-param>**: para declarar las variables globales a toda la aplicación web, y sus valores. Dentro tiene dos subetiquetas:
 - **<param-name>**: nombre de la variable o parámetro
 - **<param-value>**: valor de la variable o parámetro

Un ejemplo:

```
<context-param>
  <param-name>param1</param-name>
  <param-value>valor1</param-value>
</context-param>
```

Estos parámetros pueden leerse desde servlets con el método `getInitParameter` del objeto `ServletContext`.

2. Filtros

Para el tratamiento de filtros se tienen las etiquetas:

- **<filter>**: para asociar un nombre identificativo con la clase que implementa el filtro
- **<filter-mapping>**: para asociar un nombre identificativo de filtro con una URL o patrón de URL

Se pueden tener varias de estas etiquetas, cada una para un filtro.

3. Oyentes

Se tiene la etiqueta:

- **<listener>**: para definir una clase oyente que responde ante eventos en sesiones y contextos (al iniciar, al cerrar, al modificar).

4. Servlets

Para definir los servlets en nuestro fichero de configuración, se tienen las etiquetas:

- **<servlet>**: asocia un nombre identificativo con una clase Java que implementa un servlet
- **<servlet-mapping>**: asocia un nombre identificativo de servlet con una URL o patrón de URL.

Se pueden tener varias de estas etiquetas, cada una para un servlet.

5. Configuración de sesión

Se tiene la etiqueta:

- **<session-config>**: para indicar parámetros de configuración de las sesiones.

Por ejemplo, podemos indicar el tiempo (en minutos) que le damos a una sesión de

usuario antes de que el servidor la finalice:

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

6. Páginas de inicio

Se tiene la etiqueta:

- **<welcome-file-list>**: para indicar qué páginas debe buscar Tomcat como páginas de inicio en el caso de que en la URL se indique el directorio, pero no la página, como por ejemplo:

```
http://localhost:8080/unadireccion/dir/
```

Para ello, esta etiqueta tiene una o varias subetiquetas **<welcome-file>** para indicar cada una de las posibles páginas

Por ejemplo, podemos indicar que las páginas por defecto sean `index.html` o `index.jsp` con:

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

Las páginas se buscan en el orden en que se especifican en esta etiqueta.

7. Librerías de tags

Se tiene la etiqueta:

- **taglib**: para cargar una librería de tags para utilizar en páginas JSP. Podemos tener una o varias de estas etiquetas.

8. Seguridad

Para gestionar la seguridad en las aplicaciones Web se tienen las etiquetas:

- **security-constraint**: permite especificar qué URLs de la aplicación deben protegerse
- **login-config**: indica cómo debe autorizar el servidor a los usuarios que quieran acceder a las URLs protegidas (indicadas con `security-constraint`)
- **security-role**: da una lista de roles en los que se encuadrarán los usuarios que intenten acceder a recursos protegidos.

Existen otras etiquetas internas, relacionadas con la seguridad, que no se encuentran detalladas aquí, ya que las veremos cuando hablemos de la seguridad en el servidor.

3.3.2. El contexto de la aplicación en Tomcat

Aunque el comportamiento interno de cada aplicación se define desde su propio fichero `web.xml`, se pueden especificar también sus características a nivel global. Esto se denomina en Tomcat el *contexto* de la aplicación. Se puede definir un contexto por defecto con propiedades comunes a todas las aplicaciones y además un contexto para cada aplicación de modo individual.

3.3.2.1. Un ejemplo

Lo recomendado es definir el contexto en formato XML en un fichero denominado precisamente `context.xml`. Este fichero se colocaría en un directorio `META-INF` dentro del `.war`. Por ejemplo:

```
<Context reloadable="true">
  <WatchedResource>META-INF/miConfig.xml</WatchedResource>
</Context>
```

La configuración se hace a través de atributos del elemento `Context` o bien subelementos dentro de este. En el ejemplo anterior, el atributo `reloadable` indica que Tomcat debe recargar la aplicación cuando se modifique algo de su código Java. La etiqueta `WatchedResource` hace algo parecido pero sirve para cualquier tipo de fichero. Así, cuando modificáramos el archivo `miConfig.xml`, Tomcat recargaría la aplicación permitiendo que esta tenga en cuenta los cambios.

Nota:

Tomcat es extremadamente flexible en cuanto a dónde resida físicamente la definición del contexto. Además de lo que hemos visto, se puede colocar en el propio `server.xml`, en un archivo XML dentro de `$TOMCAT_HOME/conf/`, etc. Esto hace la configuración a veces un poco confusa. Se recomienda consultar la documentación de Tomcat para ver todas las posibilidades.

En la documentación distribuida con Tomcat se incluye una referencia de la configuración del contexto. Se recomienda consultarla para tener una información más detallada.

3.3.2.2. Filtrando peticiones: valves

Un **valve** es un componente que se inserta en el ciclo de procesamiento de la petición. Así, se pueden filtrar peticiones "sospechosas" de ser ataques, no aceptar peticiones salvo que sean de determinados *hosts*, etc. Esto se puede hacer a nivel global (dentro del *engine*), para un host en concreto (dentro de *host*) o para una única aplicación (dentro de *context*). Tomcat viene con varios *valves* predefinidos (que no son más que clases Java), aunque el usuario puede escribir los suyos propios.

Registro de accesos (*access log valve*)

Crea un registro o *log* de accesos en el formato "estándar" para servidores web. Este *log* puede ser luego analizado con alguna herramienta para chequear el número de accesos, el tiempo que permanece cada usuario en el sitio, etc. Para crear un registro de accesos, introducir en el `server.xml` un elemento similar al siguiente:

```
<Valve className="org.apache.catalina.valves.AccessLogValve"
        directory="logs" prefix="localhost_access_log."
        suffix=".txt"
        pattern="common" resolveHosts="false"/>
```

Según el nivel en el que se introduzca, se pueden registrar los accesos en todo el `host` o bien solo dentro de una aplicación (`context`). El nombre del fichero de *log* se compone con el prefijo asignado a través del atributo `prefix`, la fecha del sistema en el formato `aaaa-mm-dd` y el sufijo elegido con el atributo `suffix`. Cuando cambia la fecha automáticamente se crea un nuevo fichero de *log*.

La información que aparece en el *log* es configurable. Si en el atributo `pattern` se especifica el valor `common`, se utiliza el formato estándar, típico de otros servidores web como Apache. Al especificar el valor `combined`, a la información anterior se añade el valor de los campos `User-agent` y `Referer` de la petición HTTP. Si estos formatos no cubren nuestras necesidades, se puede hacer uno "a medida" empleando los códigos de formato que aparecen en la documentación de Tomcat.

Filtro de hosts y de peticiones (*remote host filter* y *remote address filter valve*)

Sirven para permitir o bloquear el acceso desde determinados `host` o desde determinadas direcciones IP. El nombre de los `hosts` o rango de direcciones se puede especificar mediante expresiones regulares. Por ejemplo, para permitir el acceso a la aplicación de administración únicamente desde el `host` local se introduciría en el `server.xml` algo como

```
<Context path="/admin" docBase="admin">
    ...
    <Valve
className="org.apache.catalina.valves.RemoteAddrValve"
        allow="127.0.0.1" />
    ...
</Context>
```

El atributo `allow` sirve para especificar `hosts` o IPs permitidas y `deny` para especificar las prohibidas. En caso de no especificar valor para `allow`, se utilizará el de `deny` para denegar peticiones y se aceptará el resto. De manera similar, si no se especifica valor para `deny`, se permitirán peticiones según el valor de `allow` y se rechazará el resto.

Volcado de la petición (*request dumper valve*)

Este `valve` se puede utilizar para depurar aplicaciones, ya que guarda toda la información de la petición HTTP del cliente. No utiliza más atributo que `className` para indicar el nombre de la clase que implementa este *valve*:

```
<Valve className="org.apache.catalina.valves.RequestDumperValve" />
```

Autenticación única (*single sign-on valve*)

Se puede utilizar un Valve para que el usuario se identifique en una única aplicación y automáticamente conserve la misma identidad en todas las aplicaciones del mismo host, sin necesidad de identificarse de nuevo (*single sign-on valve*). Basta con añadir a nivel de Host un Valve como:

```
<Valve className="org.apache.catalina.authenticator.SingleSignOn" />
```

3.4. Servicios en JavaEE

Un servidor de aplicaciones, además de permitirnos ejecutar nuestras aplicaciones web, ofrece una serie de servicios que pueden usar dichas aplicaciones, como conexión con bases de datos, localización de recursos externos, ejecución de componentes distribuidos, etc. El estándar JavaEE especifica los servicios que debe incorporar un servidor de aplicaciones para poder ser considerado "compatible JavaEE". Como veremos en Aplicaciones Enterprise, existen varios servidores de aplicaciones JavaEE certificados, como Glassfish, Weblogic o JBoss. Tomcat no es uno de ellos, ya que le faltan algunos servicios exigidos por el estándar.

3.4.1. Fuentes de datos y JNDI

Las aplicaciones web que acceden a una base de datos son muy frecuentes. Si la aplicación es para uso particular, o para un uso muy reducido y poco concurrente, podemos configurar un acceso por JDBC simple desde las diferentes páginas y clases que la componen.

Sin embargo, la situación cambia cuando se hace un acceso concurrente. Abrir una conexión JDBC con la base de datos es un proceso costoso en tiempo. Si cada operación requiere una nueva conexión, este tiempo se multiplicará peligrosamente. Es más eficiente mantener lo que se denomina un *pool* de conexiones, una serie de conexiones que se abren al arranque del servidor y se mantienen abiertas. Cuando un cliente realiza alguna operación con la BD se usa una conexión ya abierta, y cuando esta acaba, se devuelve al *pool*, pero en realidad no se cierra. De esta forma el proceso es mucho más eficiente.

Todos los servidores JavaEE ofrecen *pools* de conexiones a través de la clase `DataSource`, que además puede ofrecer transaccionalidad y otros aspectos. El acceso al `DataSource` o fuente de datos se hace a través de un "nombre simbólico" para evitar dependencias del nombre físico de la base de datos. Para esto, el servidor hace uso de un estándar JavaEE denominado JNDI. Este estándar permite localizar recursos físicos a partir de un nombre lógico. Aquí lo usaremos para fuentes de datos pero en principio puede usarse para acceder a cualquier recurso: componentes distribuidos, servicios de

mail, etc.

Para configurar una fuente de datos necesitaremos realizar los siguientes pasos, aunque el procedimiento concreto dependerá del servidor:

- Dejar el *driver* de la base de datos accesible al servidor de aplicaciones, ya que es el que va a abrir y cerrar físicamente las conexiones, no nuestra aplicación. Este driver normalmente lo va a proporcionar el propio distribuidor de la base de datos. En nuestro caso, como usaremos MySQL, el driver está accesible desde su web, con el nombre de *conector java*.
- Configurar las propiedades del DataSource, indicando el nombre lógico (JNDI), nombre físico de la BD, número de conexiones máximas simultáneas, etc. Generalmente el proceso se realiza en un fichero de configuración XML o a través de alguna consola de administración.
- Ya podemos acceder al DataSource en nuestro código Java y realizar operaciones con la BD.

Vamos a ver cómo se realiza este proceso en Tomcat.

3.4.2. Fuentes de datos en Tomcat

Primero, el driver java se coloca en el directorio `lib` de Tomcat. Tenemos que rearrancar el servidor para que se cargue la librería.

Ahora hay que configurar las propiedades de la fuente de datos. Esto se hace dentro de los ficheros de configuración de Tomcat, en el elemento `<Context>`. Ya hemos visto que básicamente hay dos sitios donde se pueden configurar las aplicaciones: de manera centralizada, en el `server.xml` o bien en un XML propio de la aplicación (`context.xml`). Veremos la segunda forma, aunque todo esto es aplicable también a una configuración centralizada.

Deberemos crear en una carpeta **META-INF** dentro de nuestra aplicación web, un fichero llamado **context.xml**, con el siguiente contenido:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Context>
  <Resource
    name="PruebaDS"
    type="javax.sql.DataSource"
    auth="Container"
    username="prueba"
    password="prueba"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/prueba"
    maxActive="20"
    maxIdle="5"
    maxWait="10000"/>
</Context>
```

Se ha establecido así un pool de conexiones a una base de datos MySQL. Los elementos

en negrita son dependientes de la configuración de la base de datos en concreto y variarán de caso a caso:

- **name:** El atributo `name` de la etiqueta `Resource` indica el nombre que le queremos dar al pool de conexiones. Es arbitrario, y totalmente a nuestra elección
- **username y password:** Estos atributos de `Resource` indican el usuario y password para acceso a la base de datos MySQL
- **url:** El atributo `url` de la etiqueta `Resource` especifica la URL de conexión a la base de datos. En general, la URL será casi igual que la del ejemplo, cambiando únicamente el nombre de la base de datos (*bdprueba*) por el que nos interese
- Hay una serie de parámetros adicionales de configuración para especificar características del pooling:
 - **maxActive:** máximo número de conexiones a la BD que se mantendrán activas
 - **maxIdle:** máximo número de conexiones libres que habrá en el pool (poner 0 para no tener límite). Este parámetro permitirá limitar el máximo de conexiones activas en cada momento. Por ejemplo, si `maxActive` está puesto a 100, pero sólo tenemos 20 conexiones activas, y permitimos 5 desocupadas, en total habrá 25 conexiones en el pool en ese momento (luego se crearán más si son necesarias)
 - **maxWait:** tiempo en milisegundos que se deberá esperar como máximo para recibir una conexión libre (10 segundos, en el ejemplo)

Aunque una descripción más detallada del funcionamiento de JNDI está fuera del alcance de este tema, para nuestros propósitos basta saber que el servidor de aplicaciones mantiene un servidor JNDI. El acceso inicial al servidor JNDI se hace a través del `InitialContext`. Una vez obtenido el contexto, podemos obtener recursos (`lookup`) por su nombre lógico.

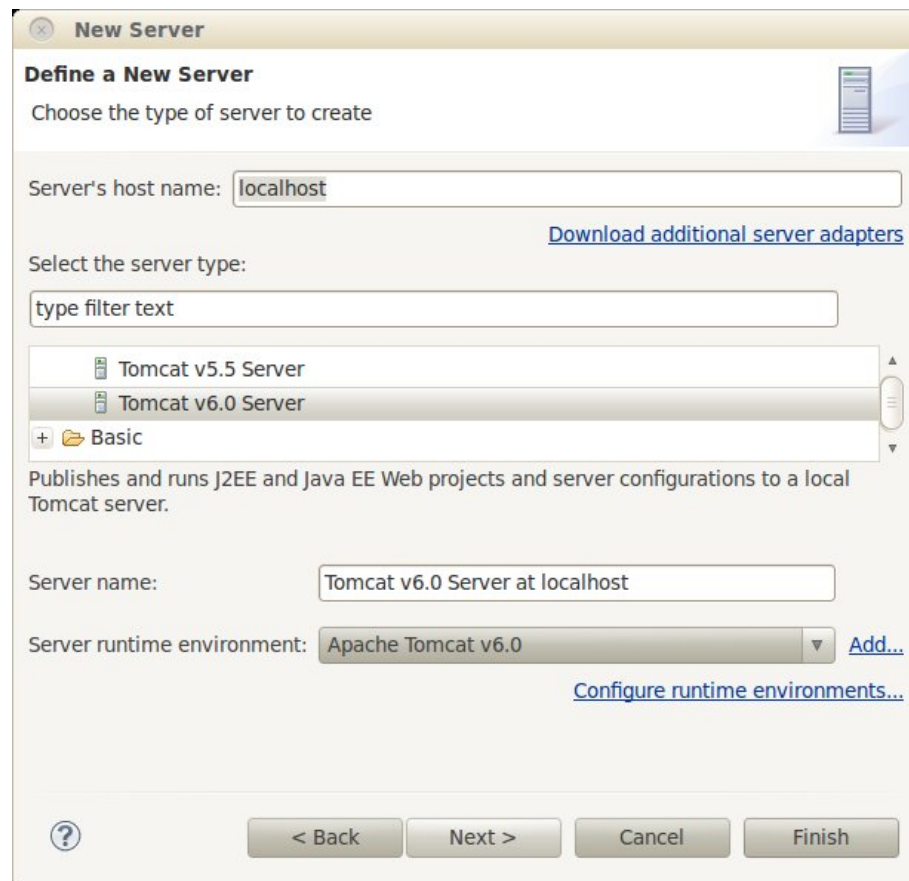
```
//Obtener el contexto JNDI
Context initCtx = new InitialContext();
//Obtener el recurso con su nombre lógico (JNDI)
DataSource ds = (DataSource)
initCtx.lookup("java:comp/env/PruebaDS");
//A través del DataSource podemos obtener una conexión con la BD
Connection conn = ds.getConnection();
//A partir de aquí trabajaríamos como es habitual en JDBC
...
```

4. Ejercicios de configuración de Tomcat y aplicaciones web

4.1. Uso de Eclipse WTP (1 punto)

Vamos a desplegar la aplicación de comentarios de la sesión anterior a través de Eclipse WebTools. Para ello:

1. **Crearemos una nueva instancia del servidor Tomcat:** Con la opción *File > New > Other > (carpeta Server) > Server* crearemos una instancia de Tomcat dentro de Eclipse. En el cuadro de diálogo que aparecerá simplemente tenemos que asegurarnos de elegir Tomcat v6.0, el resto de parámetros los podemos dejar por defecto.



Creación de nueva instancia de Tomcat con Eclipse WTP

En el Project Explorer debería aparecer el nuevo servidor entre nuestros proyectos, en una carpeta llamada Servers.

2. **Crearemos un proyecto de Eclipse de tipo Dynamic Web Project:.**
 - En la primera pantalla del asistente le damos el nombre: comentarios y nos aseguramos de que como Target runtime tenga el Apache Tomcat 6.0.

- En la segunda pantalla no hace falta modificar nada, nos basta con una sola carpeta de código Java
- En la tercera pantalla tampoco es necesario cambiar nada. No obstante aquí podríamos cambiar el "contexto" de la aplicación, que formará parte de su URL y por defecto es igual al nombre del proyecto. Es decir, que para acceder a este proyecto habría que ir a `http://localhost:8080/comentarios`.

3. Copiamos el contenido de la aplicación en el proyecto de Eclipse:

- El HTML y el fichero JSP son páginas web y por tanto las copiaremos a `WebContent`
- Crearemos un package `es.ua.jtech` y dentro de él colocaremos las dos clases que hay en la plantilla dentro de la carpeta `java`.
- Copiamos el fichero `WEB-INF/web.xml` sobrescribiendo el mismo archivo del proyecto de Eclipse (`WEB-INF` en Eclipse está dentro de `WebContent`)

4. **Ejecutamos el proyecto desplegándolo en Tomcat:** botón derecho sobre el proyecto y seleccionar *Run as > Run on server*. En la primera pantalla del asistente elegimos el servidor que hemos creado antes. En la segunda aparecerá la lista de proyectos que se van a desplegar en el servidor (solo aparecerá este). Pasados unos segundos aparecerá el navegador con la URL `http://localhost:8080/comentarios` abierta. Fijaos en que se muestra un mensaje HTTP 404 de "página no encontrada" porque la aplicación no tiene configurada una página principal. **Esto lo solucionaremos en el siguiente ejercicio.** Por el momento acceded manualmente a la página "comentarios.jsp" (la URL completa será `http://localhost:8080/comentarios`). Probad la aplicación insertando algún comentario y comprobando que se muestra correctamente.

4.2. Configuración a través del web.xml (0.5 puntos)

En el ejercicio anterior vimos que la aplicación carece de página "principal". Vamos a solucionar esto configurando la página `comentarios.jsp` como "principal". En el `web.xml` hay una sección llamada `welcome-file-list` que lista las páginas que el servidor debe buscar cuando se llame a la aplicación sin especificar la página a mostrar. Añadir `comentarios.jsp` a esta lista y comprobar que funciona adecuadamente, es decir que al llamar a `http://localhost:8080/comentarios` aparece dicha página. Será necesario rearrancar el servidor, ya que los cambios en el `web.xml` no se detectan mientras Tomcat está funcionando.

4.3. Configuración de fuentes de datos (1.5 puntos)

En las plantillas de la sesión están los archivos necesarios para crear un nuevo proyecto de Eclipse de la aplicación de comentarios pero ahora almacenando la información en una base de datos en lugar de en memoria

1. Crea otro proyecto web dinámico llamado `comentariosBD`, y copia en el lugar adecuado la página `comentarios.jsp` y los fuentes Java de `AddComentario`, `ComentariosDAO` y `FuenteDatos`

2. El fichero `comentarios.sql` contiene el script de creación de la base de datos. Desde una ventana de terminal, cambia al directorio donde esté el script `.sql` y crea la base de datos escribiendo

```
mysql -u root -p < comentarios.sql (te pedirá el password:
especialista)
```

3. Crea un `context.xml` en la carpeta `META-INF` del proyecto y configura en él la conexión con la base de datos. Ayúdate del ejemplo de los apuntes
4. La clase `FuenteDatos` es la que da acceso al `DataSource`. Rellena el código para que en el constructor de la clase se instancie el `DataSource` buscándolo con el API `JNDI` y se asigne a la variable estática `ds`.
5. Copia el [driver de MySQL](#) al directorio `lib` de Tomcat
6. Arranca el proyecto en Tomcat y comprueba que funciona correctamente. Con el MySQL Query Browser puedes comprobar si al insertar un comentario este aparece en la BD.

4.4. Configuración con el contexto (1 punto)

Vamos a configurar la aplicación a través de su contexto. Si has hecho el ejercicio anterior usa la aplicación "comentariosBD", si no usa la aplicación "comentarios". Lo más recomendable es crear la configuración en un fichero `context.xml` contenido en un directorio `META-INF` de nuestro proyecto.

- Definir un `Valve` para registrar únicamente los accesos a la aplicación de comentarios. Dichos accesos deben quedar registrados en el directorio **accesos** dentro del principal de Tomcat. El nombre del fichero de accesos debe comenzar por **comentarios** y tener extensión **".log"**. Probar a hacer peticiones para ver cómo quedan registrados los accesos.

5. Desarrollo, despliegue y pruebas de aplicaciones web con Maven

5.1. Creación y empaquetado de un proyecto web con Maven

Vamos a ver cómo crear y empaquetar en un WAR una aplicación web usando Maven.

5.1.1. Creación del proyecto

Aunque se puede crear el POM partiendo de cero, es aconsejable usar uno de los arquetipos que nos ofrece Maven. Si no vamos a emplear ningún *framework* de desarrollo web, podemos usar el arquetipo para aplicaciones web más sencillo que ofrece Maven: `maven-archetype-webapp`. Este arquetipo usa la estructura de directorios estándar de aplicaciones JavaEE y empaquetará el resultado de la compilación en un WAR en lugar de un JAR.

Podemos crear el proyecto usando el plugin de Eclipse para Maven o bien en línea de comandos. Vamos a ver aquí el proceso en línea de comandos. Supongamos que nuestra aplicación se va a llamar `WebAppEjemplo`

Primero ejecutamos `mvn archetype:generate` pasándole el `artifactId` `WebAppEjemplo` y el `groupId` `es.ua.jtech`. Elegiremos el arquetipo 85, "maven-archetype-webapp".

Elegir el arquetipo web más apropiado

Los repositorios de Maven tienen un gran número de arquetipos para aplicaciones web. En realidad el arquetipo denominado "webapp-jee5" crearía un `pom.xml` exactamente con la versión de JavaEE que estamos usando durante el curso. Pero aunque solo sea la primera vez, es mucho más instructivo examinar qué necesitamos añadir al arquetipo básico y ver por qué, que usar una plantilla hecha sin saber cómo funciona ni cómo adaptarla a la versión de JavaEE que podamos necesitar. Una vez entendidas las ideas básicas, para aumentar la productividad es mejor ir al arquetipo más similar a nuestras necesidades finales.

```
mvn archetype:generate -DgroupId=es.ua.jtech
-DartifactId=WebAppEjemplo
```

Se nos pedirá el número del arquetipo (85) y los parámetros que no hemos especificado en línea de comandos, como el `package` o la versión.

Si abrimos el `POM.xml` generado veremos que es bastante sencillo y muy similar a los proyectos Java de escritorio. Únicamente se ha cambiado el tipo de empaquetado (`<packaging>`) a WAR. La única dependencia del proyecto por el momento es JUnit (modificaremos esto más tarde).

Por otro lado, la estructura de directorios es ligeramente más compleja que en un proyecto

estándar. Dentro de la carpeta `src/main` tendremos una subcarpeta `webapp` donde se colocan los recursos web (páginas HTML o JSP, imágenes, etc) y el directorio `WEB-INF` con el descriptor de despliegue. El arquetipo seleccionado habrá creado un descriptor de despliegue mínimo y una página `index.jsp` tipo "hola, mundo".

Podemos probar el empaquetado e instalación de la aplicación en el repositorio local ejecutando

```
mvn install
```

Esto generará dentro del directorio `target` el WAR con la aplicación empaquetada. El nombre del `.war` sigue el formato `${artifactId}-${version}.war`. Además instalará el mismo `.war` en el repositorio local de maven. Podemos desplegar la aplicación en Tomcat sin más que dejar dicho `.war` en su directorio `webapps`. Posteriormente veremos cómo hacer el despliegue de forma automatizada con maven

5.1.2. Configuración de la fase de compilación

El arquetipo generado por Maven no tiene ningún fichero fuente Java de ejemplo. Evidentemente un proyecto web real contendrá clases java, que se colocarían en `src/main/java`.

La configuración más habitual en la fase de compilación es la versión de Java que debe usar el compilador a la hora de leer el fuente y generar el *bytecode*. Por defecto se supone la versión 1.4, así que si nuestro proyecto usa elementos de la 1.5 como genéricos, anotaciones, etc, tendremos que cambiarlo a esta versión. Para ello debemos configurar el `maven-compiler-plugin`. Hacia el final del `POM.xml` añadiremos el código necesario, quedando el fichero como se muestra:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>es.ua.jtech</groupId>
  <artifactId>WebAppEjemplo</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>WebAppEjemplo Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
```

```

<finalName>WebAppEjemplo</finalName>
<plugins>
  <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.5</source>
      <target>1.5</target>
    </configuration>
  </plugin>
</plugins>
</build>
</project>

```

Por otro lado, lo habitual es que nuestro proyecto dependa de librerías propias o de terceros además de JUnit. Como mínimo si es una aplicación web y tenemos algún servlet habrá dependencias del API de servlets que será necesario resolver para poder compilar. Los JAR con el API de servlets se suelen incluir en el propio servidor de aplicaciones, por lo que a maven habrá que decirle que es una dependencia únicamente en tiempo de compilación, pero que no debe empaquetarla en el .war resultante. Esto se especifica diciendo que la dependencia tiene ámbito (scope) *provided*.

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  [...]
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  [...]
</project>

```

Como se verá en el módulo de servlets y JSP, si nuestra aplicación incluye alguna librería de etiquetas (*taglib*) JSP propia, tendremos que especificar la dependencia del API de JSP. Al igual que ocurre con los servlets el JAR se suele tomar del servidor, por lo que es de ámbito *provided*.

```

<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.0</version>
  <scope>provided</scope>

```

```
</dependency>
```

5.1.3. Trabajar con proyectos web Maven desde Eclipse

Si queremos editar el código, compilarlo y probar el proyecto desde dentro de Eclipse podemos usar el plugin *m2eclipse* que ya hemos empleado en el curso. Simplemente tenemos que importar el proyecto de Maven a Eclipse, pero antes tenemos que asegurarnos de que el plugin opcional para trabajar con proyectos web está instalado. Si lo está, al seleccionar la opción de menú `Help > About Eclipse` y pulsar en `Installation details` debería aparecer entre los plugins instalados "Maven integration for WTP (optional)".

Para instalar este extra, en Eclipse ir a `Help > Install new software...`. Tenemos que añadir un nuevo *site* de donde descargar el extra. Pulsar sobre el botón `Add...`. Como nombre del sitio podemos poner por ejemplo "m2eclipse extras". La URL debe ser `http://m2eclipse.sonatype.org/sites/m2e-extras`. Tras un tiempo nos aparecerán los plugins disponibles. Seleccionamos "Maven integration for Eclipse Extras" y dentro de él debemos instalar al menos el "Maven integration for WTP (optional)".

Falta importar el proyecto que se ha creado con Maven a Eclipse. Se seleccionaría la opción de menú `File > Import...` y dentro de la categoría `Maven` se elegiría `Existing Maven projects`. A partir de este momento podemos trabajar con el proyecto como es habitual en Eclipse, con la única diferencia de que también podemos gestionar su ciclo de vida a través de Maven.

5.2. Despliegue de aplicaciones web con maven

Aunque ya conocemos diversos modos de desplegar una aplicación web en Tomcat, es interesante integrar el proceso de despliegue o de arranque/parada del contenedor con Maven. Para ello existen diversos plugins, de los cuales el más conocido es el plugin de Cargo para Maven. [Cargo](#) es un *framework* java para controlar servidores de aplicaciones (parar/arrancar, controlar el estado actual, desplegar/replegar aplicaciones,...). Se pueden controlar tanto servidores locales como remotos, y se acepta un gran número de servidores JavaEE: no solo Tomcat, sino también JBoss, Glassfish, Weblogic, Jetty y otros.

Plugins y más plugins...

Existe un [plugin Maven para Apache Tomcat](#), aunque aquí veremos Cargo porque está más extendido y además lo podremos usar también con el servidor Glassfish en aplicaciones enterprise. Otro plugin muy conocido es el [plugin de Jetty](#), ya que Jetty es un servidor ligero y de tamaño reducido. Por ello es muy usado en demos y en proyectos web distribuidos como código de ejemplo de alguna tecnología. Así ejecutar el ejemplo es muy sencillo porque Maven se encarga de resolver todas las dependencias incluyendo instalar y ejecutar el propio servidor web.

Cargo ofrece una gran versatilidad: se puede configurar cómo desplegar el artefacto creado por Maven (su localización física, la ruta de su contexto,...), todos los aspectos del servidor (si es local o remoto, de dónde tomar sus ficheros de configuración,...). Incluso se pueden hacer cosas como instalar el servidor sobre la marcha sin más que dar una URL de donde bajárselo en formato .zip. Es de esperar por tanto que su configuración detallada sea compleja. Nos limitaremos aquí a dar una configuración apropiada para nuestros propósitos.

Como nosotros usamos un servidor instalado en local, nos basta con decirle a Cargo qué servidor es y qué versión, en qué directorio está instalado y dónde queremos hacer el despliegue físico de la aplicación (se usará un directorio temporal). El plugin se incluiría en el POM.xml y se configuraría del siguiente modo:

```
[...]
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <!-- configuración del plugin de cargo -->
  <configuration>
    <!-- configuración del contenedor -->
    <container>
      <containerId>tomcat6x</containerId>
      <home>/opt/apache-tomcat-6.0.29</home>
    </container>
    <!-- configuración del despliegue -->
    <configuration>
      <home>${project.build.directory}/tomcat6x</home>
    </configuration>
  </configuration>
</plugin>
[...]
```

En el caso de Tomcat, en el directorio donde se va a hacer el despliegue se hace una copia temporal de la estructura de directorios del servidor ("conf", "logs", "webapps", etc.). El despliegue se hace copiando el war generado por maven en esta carpeta "webapps" (no en la original de la instalación local de Tomcat). Esto nos permitiría usar una configuración distinta para el servidor y específica para nuestro proyecto (algo parecido a lo que hace el plugin WTP de Eclipse).

5.2.1. Despliegue de aplicaciones

Para arrancar el servidor, hay que ejecutar el objetivo `cargo:start`. El despliegue del .war generado por Maven se hará automáticamente, pero antes tendríamos que asegurarnos de que se genere con `mvn install`. Por tanto podemos hacer:

```
mvn install cargo:start
```

Por supuesto también podríamos enlazar el objetivo start con alguna de las fases del ciclo de vida estándar, para que se ejecute automáticamente. Posteriormente veremos cómo

hacer esto en el contexto de pruebas de integración. El shell desde el que se arranca el servidor se queda parado por defecto con el servidor ejecutándose (salvo que especifiquemos `<wait>false</wait>` en la configuración del plugin). La parada del servidor se puede hacer con Ctrl-C desde el shell con el que se ha arrancado o bien ejecutando el objetivo `cargo:stop`

Si lo único que queremos hacer es el despliegue, ejecutamos `cargo:deploy`. Para replegar, `cargo:undeploy`

5.2.2. Inclusión de librerías comunes

Es muy típico tener un mismo JAR que está compartido por varias aplicaciones. Normalmente todos los servidores tienen un directorio donde dejar estos JAR de manera que son accesibles a todas las aplicaciones desplegadas, o un mecanismo equivalente (recordemos que en Tomcat este directorio es "lib"). Así no es necesario incluir el JAR por separado en cada aplicación.

Cargo nos permite incluir el JAR en el servidor y hacerlo accesible a todas las aplicaciones sin necesidad de preocuparnos de en qué directorio físico debe residir con el servidor actual. Para ello hay que incluir en el POM la dependencia del JAR del modo estándar en Maven y luego añadir un `<dependencies>` con la lista de dependencias dentro de la configuración del contenedor de Cargo. Por ejemplo, para incluir en el servidor el JAR con el driver de MySQL haríamos algo como:

```
<!-- sección estándar de dependencias -->
<dependencies>
[...]
```

```
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.13</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

```
<build>
  <plugins>
    <plugin>
      [...]
    </plugin>
    <plugin>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-maven2-plugin</artifactId>
      <configuration>
        <container>
          <containerId>tomcat6x</containerId>
          <home>/opt/apache-tomcat-6.0.29</home>
          <!-- dependencias para el contenedor -->
          <dependencies>
            <dependency>
              <groupId>mysql</groupId>
```

```

que solo
        <artifactId>mysql-connector-java</artifactId>
        <!-- nótese que aquí no ponemos la versión ya
        estamos referenciando la dependencia
        que está en la sección "estándar" de Maven -->
        </dependency>
    </dependencies>
</container>
<configuration>
    ${project.build.directory}/tomcat6x</home>
</configuration>
</configuration>
</plugin>
[... ]
</plugins>
[... ]
</build>

```

5.3. Pruebas en aplicaciones web

En la capa web puede ser necesario hacer diversos tipos de pruebas:

- **Pruebas unitarias:**, que comprueben el funcionamiento de los servlets propiamente dichos. Se usarían *mocks* para modelar el resto de elementos con los que va a interactuar el servlet, sean clases de la capa de negocio o bien elementos del contexto de los servlets (la petición, la respuesta, *cookies*, la sesión,...). Un ejemplo de herramienta que nos permite hacer pruebas de este tipo es [ServletUnit](#), herramienta que forma parte de [HttpUnit](#).
- **Pruebas de integración** de los servlets con el resto de elementos que forman parte de la aplicación. En este caso es apropiado probar los servlets dentro de un contenedor web real. Esto podemos hacerlo por ejemplo con la herramienta [Cactus](#).
- **Pruebas funcionales:** un tercer tipo de pruebas serían aquellas que se hacen acercándose más al punto de vista del usuario final. Es decir, seguir la secuencia de un caso de uso y comprobar que el sistema devuelve los resultados esperados. Para hacer este tipo de pruebas disponemos de herramientas como [HtmlUnit](#), [Selenium](#) o [JWebUnit](#) (esta última "envuelve" a las dos primeras para poder realizar las pruebas con un API común).

5.3.1. Pruebas unitarias en la capa web

Gestionar las pruebas unitarias con Maven sería muy similar a gestionar el tipo de pruebas unitarias que hemos hecho hasta ahora en el curso, con la única diferencia de que nos ayudaría disponer de una herramienta que incorpore los *mocks* de las clases del contenedor con las que debe interactuar el servlet. Como hemos dicho, ServletUnit es una de dichas herramientas. Como forma parte de HttpUnit habrá que incluir la dependencia de esta última en el POM.

```
<dependency>
```

```

<groupId>httpunit</groupId>
<artifactId>httpunit</artifactId>
<version>1.7</version>
</dependency>

```

ServletUnit nos permite hacer pruebas de "caja negra", comprobando el valor de la respuesta que devuelve el servlet. También podemos ejecutar métodos individuales del servlet y comprobar los valores que devuelve el método o los que coloca el servlet en los distintos ámbitos (petición, sesión, ...).

Por ejemplo, supongamos el código del siguiente servlet:

```

public class SaludoServlet extends HttpServlet {
    public SaludoServlet() {
        super();
    }

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

    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        String nom = request.getParameter("nombre");
        request.getSession().setAttribute("usuario", nom);
        out.println("<html> <head> <title>Saludo</title></head>");
        out.println("<body> <h1> Hola " + nom + "</h1>");
        out.println("</body> </html>");
    }
}

```

Podemos comprobar por ejemplo que al llamar al doPost se guarda en la sesión el valor esperado:

```

public class SaludoTest extends TestCase {
    //esto es lo que nos permite lanzar el servlet en el entorno
    simulado
    ServletRunner sr;

    protected void setUp() throws Exception {
        super.setUp();
        sr = new ServletRunner();
        //registramos el servlet para poder llamarlo
        sr.registerServlet("SaludoServlet",
        SaludoServlet.class.getName());
    }

    public void testSaludo() throws IOException, SAXException,
    ServletException {
        //Llamamos al servlet por el nombre registrado, no por su
        URL del web.xml
    }
}

```

```

        WebRequest pet = new
GetMethodWebRequest("http://localhost:8080/SaludoServlet");
pet.setParameter("nombre", "Juan");
//Queremos obtener el "invocation context", necesario para
//poder llamar a los métodos del servlet
ServletUnitClient sc = sr.newClient();
InvocationContext ic = sc.newInvocation(pet);
//Del "invocation context" obtenemos el servlet
SaludoServlet ss = (SaludoServlet) ic.getServlet();
//Llamamos manualmente al método "doPost"
ss.doPost(ic.getRequest(), ic.getResponse());
//comprobamos que en la sesión se guarda el valor correcto
assertEquals("nombre en sesión", "Juan",
ic.getRequest().getSession().getAttribute("usuario"));
}

```

También podemos probar el resultado devuelto por el servlet sin entrar a ejecutar los métodos manualmente. Esto es más sencillo (aunque para eso hay otras herramientas más sofisticadas)

```

public void testSaludoCajaNegra() {
    ServletUnitClient sc = sr.newClient();
    WebRequest pet = new
GetMethodWebRequest("http://localhost:8080/SaludoServlet");
    pet.setParameter("nombre", "Juan");
    WebResponse resp = sc.getResponse(pet);
    assertEquals("Titulo de la página", "Saludo",
resp.getTitle());
}

```

5.3.2. Pruebas de integración y funcionales

No obstante, el asunto de las pruebas funcionales y de integración es distinto, ya que se deben ejecutar con el contenedor web arrancado, y por tanto en una fase distinta a las pruebas unitarias. En el ciclo de vida estándar de Maven hay tres fases correspondientes a pruebas de integración: `pre-integration-test`, `integration-test` y `post-integration-test`. Estas fases se ejecutan después de `package`, cuando se empaqueta el artefacto generado en un JAR, WAR o EAR. Vamos a ver cómo usar estas fases para hacer las pruebas.

5.3.2.1. Adaptar la estructura del proyecto

Por desgracia, Maven no incluye por defecto la posibilidad de ejecutar un conjunto de prueba en la fase `test` (pruebas unitarias) y usar otro distinto en la fase `integration-test`. Tendremos que conseguirlo configurando manualmente el POM. Caben dos posibilidades:

- Configurar el plugin `surefire` de modo que ejecute algunos ficheros de prueba en la fase de pruebas unitarias y otros en la fase de integración. La distinción entre unos y otros se puede hacer por el directorio en que están guardados o por el nombre de las

clases de prueba. Cargo incluye un arquetipo que nos creará un proyecto de esta clase: cargo-archetype-webapp-single-module (179). En el POM podremos ver que los test de integración se meten en un directorio llamado `it` dentro de `src/test/java` y que se configura el plugin surefire para que ignore este directorio durante la fase de pruebas unitarias y ejecute los tests durante la fase de pruebas de integración.

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-surefire-plugin</artifactId>
<configuration>
  <!--
    Excluir los test de integración de la fase de test
    (fase de pruebas unitarias)
  -->
  <excludes>
    <exclude>**/it/**</exclude>
  </excludes>
</configuration>
<executions>
  <execution>
    <phase>integration-test</phase>
    <goals>
      <goal>test</goal>
    </goals>
    <configuration>
      <!--
        Incluir los test de integración en la fase
        integration-test
      -->
      <excludes>
        <exclude>none</exclude>
      </excludes>
      <includes>
        <include>**/it/**</include>
      </includes>
    </configuration>
  </execution>
</executions>
</plugin>
```

- Construir un proyecto multimódulo. En uno de los módulos irá la aplicación web junto con las pruebas unitarias, y en otro irán las pruebas de integración. Así tenemos por separado este código, lo que hace la estructura más limpia y más flexible. Podemos, por ejemplo, ejecutar las pruebas de integración o funcionales solo en determinadas ocasiones y no en cada compilación del proyecto. Cargo nos ofrece un arquetipo que seguirá esta estructura:
cargo-archetype-webapp-functional-tests-module (178)

5.3.2.2. Gestionar el servidor web en la fase de pruebas

En cualquier caso, hay que arrancar el servidor web antes de ejecutar las pruebas de integración (fase "pre-integration-test") y pararlo después de ejecutar dichas pruebas (fase "post-integration-test"). Simplemente tendremos que enlazar el objetivo cargo:start con la

primera fase y cargo:stop con la otra. Podemos conseguirlo introduciendo en el POM un código como este:

```
[...]
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    <container>
      <!-- Aquí faltaría la configuración del contenedor
           (qué servidor es, dónde está,...) -->
      [...]
    </container>
    <wait>false</wait>
  </configuration>
  <executions>
    <execution>
      <id>start</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>start</goal>
      </goals>
    </execution>
    <execution>
      <id>stop</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>stop</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Nótese que en la configuración del contenedor ponemos `<wait>false</wait>` para que Maven no se quede "esperando" tras arrancar el contenedor.

5.3.2.3. Ejecutar las pruebas

Como ya hemos dicho, existen diversas herramientas para ejecutar pruebas de integración y funcionales. Vamos a ver aquí un ejemplo de JWebUnit, que nos ofrece un API sencillo para probar la aplicación web del mismo modo que lo haría un usuario final. Así podemos acceder a una página, rellenar los campos de un formulario y enviarlo o pinchar en un enlace. Podemos también comprobar si en la página aparece algún elemento que tendría que estar (un determinado enlace, un texto, una tabla) o si tiene las características deseadas (el título, el tamaño,...).

Para usar JWebUnit debemos incluir la siguiente dependencia en el POM

```
<dependency>
  <groupId>net.sourceforge.jwebunit</groupId>
  <artifactId>jwebunit-htmlunit-plugin</artifactId>
  <version>2.2</version>
```

```
<scope>test</scope>
</dependency>
```

JWebUnit es una capa de abstracción sobre HtmlUnit (una herramienta bastante veterana para pruebas web), así que veremos que Maven descarga también a nuestro repositorio esta última.

Suponiendo que tuviéramos el servlet `SaludoServlet` del que hablábamos en el apartado de pruebas unitarias y además el siguiente JSP que lo llama a través de un formulario:

```
<html>
  <head> <title>Página principal</title> </head>
  <body>
    <h2>Esto es la página principal</h2>
    <form action="SaludoServlet" method="get">
      Escribe tu nombre: <input type="text" name="nombre"><br>
      <input type="submit" value="Enviar">
    </form>
  </body>
</html>
```

Podríamos comprobar que tanto el JSP como el servlet devuelven el resultado esperado:

```
public class WebAppTest extends WebTestCase {
    public void setUp() throws Exception {
        super.setUp();
        //especificar la URL base de la aplicación
        setBaseUrl("http://localhost:8080/EjemploTestsIntWebApp");
    }

    public void testIndex() {
        //página por la que empezamos a navegar ("/" será la
principal)
        beginAt("/");
        //comprobar el <title> de la página
        assertEquals("Página principal");
    }

    public void testSaludo() {
        beginAt("/");
        //rellenar un formulario
        setTextField("nombre", "Pepe");
        //enviarlo
        submit();
        assertEquals("Saludo");
        //comprobar que la página contiene un determinado texto
        assertTextPresent("Hola Pepe");
    }
}
```

Se recomienda consultar la [referencia básica](#) de JWebUnit para ver con más detalle las posibilidades que ofrece la herramienta.

6. Ejercicios de desarrollo, despliegue y pruebas de aplicaciones web con Maven

Instalación del extra de m2eclipse para WTP

Antes de realizar los ejercicios de esta sesión deberías instalar el extra para WTP del plugin m2eclipse, que estamos usando para poder importar proyectos Maven a este entorno. Consulta los apuntes para ver el proceso. Si no instalas este extra no podrás ejecutar el proyecto web desde Eclipse, solo desde Maven. Eso sí, aunque no tengas el extra instalado podrás compilarlo desde Eclipse sin problemas y al menos estar seguro de que no tiene errores de compilación

6.1. Creación de un proyecto web en Eclipse basado en Maven (1 pto)

Vamos a crear una pequeña aplicación web que va a convertir una cantidad de dinero entre pesetas y euros. La aplicación tendrá un HTML con un formulario para escribir los datos y un servlet para calcular y mostrar el resultado.

Debéis realizar los siguientes pasos:

- 1. Crear el proyecto:** File > New > Other... y dentro de "Maven" escoger Maven Project.
 - En la pantalla de selección del arquetipo, escoger webapp-jee5. Este arquetipo ya tiene configurada la versión adecuada del API de servlets (2.5) y la del código generado por el compilador Java (1.5).
 - En la última pantalla del asistente, poner como groupId `es.ua.jtech` y como artifactId `conversor`
- 2. Incluir el código fuente:**
 - Tomar el "index.html" incluido en las plantillas de la sesión y copiarlo a la carpeta "Web Resources" del proyecto. El "index.jsp" que contiene "Web Resources" debéis eliminarlo (si no lo hacéis aparecerá como página principal en lugar del index.html).
 - Crear un servlet llamado Conversor: File > New > Servlet. En la pantalla del asistente debes poner como package "es.ua.jtech" y como nombre de la clase "ConversorServlet". No darle al botón "Finish" sino a "Next" para que nos cree automáticamente el mapeo del servlet en el web.xml. En esta segunda pantalla del asistente puedes aceptar los valores por defecto (asociar con la URL `"/ConversorServlet"`)
 - Sustituye el código fuente del servlet generado por el código incluido en el fichero "ConversorServlet.txt" de las plantillas de la sesión.
- 3. Ejecutar el proyecto:** ejecuta el proyecto desde Eclipse como es habitual y comprueba que funciona correctamente. CUIDADO: esto no podrás hacerlo si no has instalado el extra del plugin m2eclipse.

6.2. Despliegue de la aplicación en Maven usando Cargo (1 pto)

1. Incluye en el "pom.xml" la configuración del plugin de Cargo como se explica en el apartado "Despliegue de aplicaciones web con maven" de los apuntes.
2. Desde un terminal muévete al directorio del proyecto y ejecuta

```
mvn install cargo:start
```

para ejecutar la aplicación en Tomcat. La primera vez tardará un rato ya que tiene que bajarse Cargo y todas sus dependencias. Abre una ventana del navegador y comprueba que la aplicación funciona. Puedes parar el servidor pulsando Ctrl-C en la terminal desde la que lo has arrancado o ejecutando `mvn cargo:stop` desde otra terminal y desde el directorio del proyecto

3. Comenta todo lo relativo a la "configuration" en el plugin anterior (es decir, solo debería quedar el "groupId" y el "artifactId"). No lo borres, solo ponlo entre comentarios (`<!--` y `-->`). De este modo usarás la configuración por defecto de Cargo, que es el servidor Jetty. Arranca de nuevo la aplicación y comprueba que funciona en este servidor.

6.3. Pruebas funcionales con JWebUnit (1 pto)

1. Si has hecho el ejercicio 2, haz una copia del fichero pom.xml como "pom_ejer2.xml". Así quedará una copia de lo que hiciste en el ejercicio anterior, ya que vas a modificar de nuevo el pom.xml.
2. Incluye en el pom.xml la nueva configuración del plugin de Cargo. Esta configuración gestionará el arranque y parada automáticos del servidor en la fase de "integration-test" (cuidado, tienes que sustituir la configuración de Cargo que has puesto en el ejercicio 2, esta nueva incluye también la anterior)

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    <wait>false</wait>
    <container>
      <containerId>tomcat6x</containerId>
      <home>/opt/apache-tomcat-6.0.29</home>
    </container>
    <configuration>
<home>${project.build.directory}/tomcat6x</home>
    </configuration>
  </configuration>
  <executions>
    <execution>
      <id>start</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>start</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```

        </execution>
        <execution>
            <id>stop</id>
            <phase>post-integration-test</phase>
            <goals>
                <goal>stop</goal>
            </goals>
        </execution>
    </executions>
</plugin>

```

3. Incluye la configuración del plugin "surefire" para que se usen los test del directorio "it" en la fase de integración:

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <!--
      Excluir los test de integración de la fase de test
      (fase de pruebas unitarias)
    -->
    <excludes>
      <exclude>*/it/*</exclude>
    </excludes>
  </configuration>
  <executions>
    <execution>
      <phase>integration-test</phase>
      <goals>
        <goal>test</goal>
      </goals>
      <configuration>
        <!--
          Incluir los test de integración en la fase
          integration-test
        -->
        <excludes>
          <exclude>none</exclude>
        </excludes>
        <includes>
          <include>*/it/*</include>
        </includes>
      </configuration>
    </execution>
  </executions>
</plugin>

```

4. Incluye la dependencia de JWebUnit en la sección de <dependencies>

```

<dependency>
  <groupId>net.sourceforge.jwebunit</groupId>
  <artifactId>jwebunit-htmlunit-plugin</artifactId>
  <version>2.2</version>
  <scope>test</scope>
</dependency>

```

5. Escribe la clase de prueba:

- Crea una nueva clase llamada `ConversorTest` que herede de `WebTestCase` y resida en el package **es.ua.jtech.conversor.it** pero asegúrate de que se mete en la carpeta `src/test/java`, y no `src/main/java`, ya que es un test.
- Fijándote en el ejemplo de test `JWebUnit` de los apuntes, escribe el método `setUp` estableciendo como URL base "`http://localhost:8080/conversor`"
- Escribe un método de prueba llamado `testConversor` que introduzca un valor en euros en el campo "cantidad", envíe el formulario y compruebe que en la página resultante aparece el resultado correcto.

7. Seguridad en aplicaciones web

Cuando hablamos de seguridad en aplicaciones web podemos distinguir dos aspectos: por un lado la seguridad del servidor, es decir, cómo configurar las restricciones de seguridad que va a tener la propia aplicación (qué operaciones va a poder realizar y qué operaciones no, independientemente del usuario que esté accediendo). Por otro, como autenticar a los usuarios y cómo aplicar las restricciones a las operaciones que puedan realizar dentro de la aplicación. Hablaremos aquí de estos dos aspectos, centrándonos sobre todo en el segundo.

7.1. Seguridad del servidor

7.1.1. Políticas de seguridad

Si los usuarios del servidor pueden colocar en él sus propias aplicaciones web es necesario asegurar que dichas aplicaciones no van a comprometer la seguridad de las demás, del servidor o del propio sistema. Aunque confiemos en la buena fe del que desarrolla las aplicaciones, alguien podría aprovechar un "bug" en una de ellas para efectuar operaciones no permitidas.

Tomcat utiliza los mecanismos estándar de seguridad de Java para asegurar que las clases ejecutadas en el servidor cumplen una serie de restricciones. Dichas restricciones se definen en el fichero de políticas de seguridad `conf/catalina.policy`. No obstante, por defecto estas restricciones no están activadas salvo que se arranque Tomcat con la opción `-security`.

Políticas de seguridad por defecto

Si se observa el contenido del fichero `conf/catalina.policy` se verá que por defecto, a las clases que componen Tomcat y a las pertenecientes al JDK se les asignan todos los permisos posibles. A continuación se muestra un extracto de dicho fichero:

```
...
// Permisos para las clases del JDK
// (están dentro de JAVA_HOME)
grant codeBase "file:${java.home}/lib/-" {
    permission java.security.AllPermission;
};
...
// Permisos para las clases de Tomcat
// (están dentro de CATALINA_HOME)
grant codeBase "file:${catalina.home}/bin/bootstrap.jar" {
    permission java.security.AllPermission;
};
...
// Permisos para aplicaciones web
grant {
```



```
// Required for JNDI lookup of named JDBC DataSource's and
// javamail named MimePart DataSource used to send mail
permission java.util.PropertyPermission "java.home",
"read";
...
```

Como puede observarse si se examina detenidamente el fichero, a las clases que componen una aplicación web cualquiera se les dan permisos adecuados para poder acceder a JNDI, leer ciertas propiedades del sistema, etc.

Cambiar las políticas de seguridad

Cambiando el fichero de políticas podemos asignar permisos especiales a ciertas aplicaciones web que lo requieran, manteniendo las restricciones en las demás. Por ejemplo, para dar ciertos permisos a la aplicación j2ee habrá que introducir un código similar al siguiente en el fichero de políticas

```
grant codeBase "file:${catalina.home}/webapps/j2ee/-" {
    permission ...
}
```

7.1.2. Autenticación en Tomcat: realms

Cualquier aplicación medianamente compleja tendrá que autenticar a los usuarios que acceden a ella, y en función de quiénes son, permitirles o no la ejecución de ciertas operaciones

Los mecanismos de autenticación en Tomcat se basan en el concepto de *realm*. Un *realm* es un conjunto de usuarios, cada uno con un *password* y uno o más *roles*. Los roles determinan qué permisos tiene el usuario en una aplicación web (esto es configurable en cada aplicación a través del descriptor de despliegue, *web.xml*).

Tomcat proporciona distintas implementaciones para los realms, que básicamente se diferencian en dónde están almacenados los datos de *logins*, *passwords* y *roles*. En la configuración de Tomcat por defecto, dichos datos están en el fichero *conf/tomcat-users.xml*, pero pueden almacenarse en una base de datos con JDBC, tomarse de un directorio LDAP u obtenerse mediante JAAS, el API estándar de Java para autenticación.

Los *realms* se definen en el fichero de configuración *server.xml* introduciendo el elemento *Realm* con un atributo *className* que indique la implementación que deseamos utilizar. El resto de atributos dependen de la implementación en concreto. Podemos definir *realms* en distintos puntos del fichero, de manera que afecten a todo el servidor, a un *host* o solo a una aplicación. En cada caso, siempre se utilizará el *realm* aplicable más bajo en la "jerarquía de configuración", de modo que si se configura para una aplicación en concreto, el general deja de utilizarse.

UserDatabaseRealm

Es la implementación que utiliza por defecto la distribución de Tomcat, y como se ha comentado, toma los datos de un fichero. Este fichero se lee cuando arranca el servidor, de modo que cualquier cambio en el mismo requiere el rearranque de Tomcat para tener efecto. La definición del *realm* en el fichero de configuración se realiza de la siguiente manera:

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
      debug="0" resourceName="UserDatabase"/>
```

Donde el atributo `resourceName` determina el recurso del que se obtiene la información. Este está definido dentro de `GlobalNamingResources` para que apunte al fichero `conf/tomcat-users.xml`.

Dicho fichero tiene un formato similar al siguiente:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<tomcat-users>
  <role rolename="usuario"/>
  <role rolename="admin"/>
  <user username="pepe" password="pepepw" roles="usuario"/>
  <user username="manuel" password="manolo" roles="admin"/>
  <user username="toni" password="toni" roles="usuario,
admin"/>
</tomcat-users>
```

Así, por ejemplo, para un recurso (URL) al que sólo puedan acceder roles de tipo *admin*, podrían acceder los usuarios *manuel* y *toni*. Notar también que los passwords están visibles en un fichero de texto fácilmente accesible por casi cualquiera, con lo que no es una buena forma de gestionar los passwords para una aplicación profesional.

JDBCRealm

Esta implementación almacena los datos de los usuarios en una base de datos con JDBC, lo cual es mucho más flexible y escalable que el mecanismo anterior y además no requiere el rearranque de Tomcat cada vez que se cambia algún dato. Para configurar Tomcat con esta implementación de *realm* se puede descomentar uno que ya viene definido en el `server.xml`, a nivel global dentro del elemento `Engine`. Los atributos de este *realm* indican cómo efectuar la conexión JDBC y cuáles son los campos de la base de datos que contienen *logins*, *passwords* y roles.

Atributo	Significado
<code>className</code>	clase Java que implementa este <i>realm</i> . Debe ser <code>org.apache.catalina.realm.JDBCRealm</code>
<code>connectionName</code>	nombre de usuario para la conexión JDBC
<code>connectionPassword</code>	password para la conexión JDBC
<code>connectionURL</code>	URL de la base de datos
<code>debug</code>	nivel de depuración. Por defecto 0 (ninguno).

	Valores más altos indican más detalle.
digest	Algoritmo de "digest" (puede ser SHA, MD2 o MD5). Por defecto es <code>cleartext</code>
driverName	clase Java que implementa el <i>driver</i> de la B.D.
roleNameCol	nombre del campo que almacena los roles
userNameCol	nombre del campo que almacena los <i>logins</i>
userCredCol	nombre del campo que almacena los <i>passwords</i>
userRoleTable	nombre de la tabla que almacena la relación entre <i>login</i> y roles
userTable	nombre de la tabla que almacena la relación entre <i>login</i> y <i>password</i>

Un ejemplo de etiqueta con este tipo de Realm (fichero `conf/server.xml`) sería:

```
<Realm className="org.apache.catalina.realm.JDBCRealm"
  driverName="com.mysql.jdbc.Driver"
    connectionURL="jdbc:mysql://localhost/authority"
    connectionName="test" connectionPassword="test"
    userTable="users" userNameCol="user_name"
userCredCol="user_pass"
    userRoleTable="user_roles" roleNameCol="role_name" />
```

7.2. Seguridad en aplicaciones web

7.2.1. Tipologías de seguridad y autenticación

Podemos tener básicamente dos motivos para proteger una aplicación web:

- Evitar que usuarios no autorizados accedan a determinados recursos.
- Prevenir que se acceda a los datos que se intercambian en una transferencia a lo largo de la red.

Para cubrir estos agujeros, un sistema de seguridad se apoya en tres aspectos importantes:

- **Autenticación:** medios para identificar a los elementos que intervienen en el acceso a recursos.
- **Confidencialidad:** asegurar que sólo los elementos que intervienen entienden el proceso de comunicación establecido.
- **Integridad:** verificar que el contenido de la comunicación no se modifica durante la transmisión.

Control de la seguridad

Desde el punto de vista de quién controla la seguridad en una aplicación web, existen dos formas de implantación:

- **Seguridad declarativa:** Aquella estructura de seguridad sobre una aplicación que es externa a dicha aplicación. Con ella, no tendremos que preocuparnos de gestionar la seguridad en ningún servlet, página JSP, etc, de nuestra aplicación, sino que el propio servidor Web se encarga de todo. Así, ante cada petición, comprueba si el usuario se ha autenticado ya, y si no le pide login y password para ver si puede acceder al recurso solicitado. Todo esto se realiza de forma transparente al usuario. Mediante el descriptor de la aplicación principalmente (archivo *web.xml* en Tomcat), comprueba la configuración de seguridad que queremos dar.
- **Seguridad programada:** Mediante la seguridad programada, son los servlets y páginas JSP quienes, al menos parcialmente, controlan la seguridad de la aplicación.

En este tema veremos la seguridad declarativa, que es la que puede configurar el administrador del servidor web, dejando para módulos posteriores la seguridad programada

Autenticación

Tenemos distintos tipos de autenticación que podemos emplear en una aplicación web:

- **Autenticación *basic*:** Con HTTP se proporciona un mecanismo de autenticación básico, basado en cabeceras de autenticación para solicitar datos del usuario (el servidor) y para enviar los datos del usuario (el cliente). Esta autenticación no proporciona confidencialidad ni integridad, sólo se emplea una codificación Base64.
- **Autenticación *digest*:** Existe una variante de lo anterior, la autenticación **digest**, donde, en lugar de transmitir el password por la red, se emplea un password codificado utilizando el método de encriptado MD5. Sin embargo, algunos servidores no soportan este tipo de autenticación.
- **Autenticación basada en formularios:** Con este tipo de autenticación, el usuario introduce su login y password mediante un formulario HTML (y no con un cuadro de diálogo, como las anteriores). El fichero descriptor contiene para ello entradas que indican la página con el formulario de autenticación y una página de error. Tiene el mismo inconveniente que la autenticación *basic*: el password se codifica con un mecanismo muy pobre.
- **Certificados digitales y SSL:** Con HTTP también se permite el uso de SSL y los certificados digitales, apoyados en los sistemas de criptografía de clave pública. Así, la capa SSL, trabajando entre TCP/IP y HTTP, asegura, mediante criptografía de clave pública, la integridad, confidencialidad y autenticación.

7.2.2. Autenticación basada en formularios

Veremos ahora con más profundidad la autenticación basada en formularios comentada anteriormente. Esta es la forma más comúnmente usada para imponer seguridad en una aplicación, puesto que se emplean **formularios HTML**.

El programador emplea el descriptor de despliegue para identificar los recursos a proteger, e indicar la página con el formulario a mostrar, y la página con el error a mostrar en caso de autenticación incorrecta. Así, un usuario que intente acceder a la parte restringida es redirigido automáticamente a la página del formulario, si no ha sido autenticado previamente. Si se autentifica correctamente accede al recurso, y si no se le muestra la página de error. Todo este proceso lo controla el servidor automáticamente.

Este tipo de autenticación no se garantiza que funcione cuando se emplea reescritura de URLs en el seguimiento de sesiones. También podemos incorporar SSL a este proceso, de forma que no se vea modificado el funcionamiento aparente del mismo.

Para utilizar la autenticación basada en formularios, se siguen los pasos que veremos a continuación. Sólo el primero es dependiente del servidor que se utilice.

1. Establecer los logins, passwords y roles

En este paso definiríamos un *realm* del modo que se ha explicado en apartados anteriores.

2. Indicar al servlet que se empleará autenticación basada en formularios, e indicar las páginas de formulario y error.

Se coloca para ello una etiqueta `<login-config>` en el descriptor de despliegue. Dentro, se emplean las subetiquetas:

- `<auth-method>` que en general puede valer:
 - FORM: para autenticación basada en formularios (como es el caso)
 - BASIC: para autenticación BASIC
 - DIGEST: para autenticación DIGEST
 - CLIENT-CERT: para SSL
- `<form-login-config>` que indica las dos páginas HTML (la del formulario y la de error) con las etiquetas:
 - `<form-login-page>` (para la de autenticación)
 - `<form-error-page>` (para la página de error).

Por ejemplo, podemos tener las siguientes líneas en el descriptor de despliegue:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
...
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>
        /login.jsp
      </form-login-page>
      <form-error-page>
```

```

                                /error.html
                        </form-error-page>
                </form-login-config>
        </login-config>
        ...
</web-app>

```

3. Crear la página de login

El formulario de esta página debe contener campos para introducir el login y el password, que deben llamarse *j_username* y *j_password*. La acción del formulario debe ser *j_security_check*, y el METHOD = POST (para no mostrar los datos de identificación en la barra del explorador). Por ejemplo, podríamos tener la página:

```

<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<body>
    <form action="j_security_check" METHOD="POST">
    <table>
    <tr>
        <td>
            Login:<input type="text"
name="j_username"/>
        </td>
    </tr>
    <tr>
        <td>
            Password:<input type="text"
name="j_password"/>
        </td>
    </tr>
    <tr>
        <td>
            <input type="submit" value="Enviar"/>
        </td>
    </tr>
    </table>
    </form>
</body>
</html>

```

4. Crear la página de error

La página puede tener el mensaje de error que se quiera. Ante fallos de autenticación, se redirigirá a esta página con un código 401. Un ejemplo de página sería:

```

<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<body>
    <h1>ERROR AL AUTENTIFICAR USUARIO</h1>
</body>
</html>

```

5. Indicar qué direcciones deben protegerse con autenticación

Para ello utilizamos etiquetas `<security-constraint>` en el descriptor de despliegue. Dichos elementos debe ir inmediatamente antes de `<login-config>`, y utilizan las subetiquetas:

- `<display-name>` para dar un nombre identificativo a emplear (opcional)
- `<web-resource-collection>` para especificar los patrones de URL que se protegen (requerido). Se permiten varias entradas de este tipo para especificar recursos de varios lugares. Cada uno contiene:
 - Una etiqueta `<web-resource-name>` que da un nombre identificativo arbitrario al recurso o recursos
 - Una etiqueta `<url-pattern>` que indica las URLs que deben protegerse
 - Una etiqueta `<http-method>` que indica el método o métodos HTTP a los que se aplicará la restricción (opcional)
 - Una etiqueta `<description>` con documentación sobre el conjunto de recursos a proteger (opcional)

NOTA: este modo de restricción se aplica sólo cuando se accede al recurso directamente, no a través de arquitecturas MVC (Modelo-Vista-Controlador), con un *RequestDispatcher*. Es decir, si por ejemplo un servlet accede a una página JSP protegida, este mecanismo no tiene efecto, pero sí cuando se intenta acceder a la página JSP directamente.

- `<auth-constraint>` indica los roles de usuario que pueden acceder a los recursos indicados (opcional) Contiene:
 - Uno o varios subelementos `<role-name>` indicando cada rol que tiene permiso de acceso. Si queremos dar permiso a todos los roles, utilizamos una etiqueta `<role-name>*</role-name>`.
 - Una etiqueta `<description>` indicando la descripción de los mismos.

En teoría esta etiqueta es opcional, pero omitiéndola indicamos que ningún rol tiene permiso de acceso. Aunque esto puede parecer absurdo, recordar que este sistema sólo se aplica al acceso directo a las URLs (no a través de un modelo MVC), con lo que puede tener su utilidad.

Añadimos alguna dirección protegida al fichero que vamos construyendo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>
                Prueba
            </web-resource-name>
            <url-pattern>
                /prueba/*
            </url-pattern>
        </web-resource-collection>
```

```

        <auth-constraint>
            <role-name>admin</role-name>
            <role-name>subadmin</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        ...
</web-app>

```

En este caso protegemos todas las URLs de la forma `http://host/ruta_aplicacion/prueba/*`, de forma que sólo los usuarios que tengan roles de *admin* o de *subadmin* podrán acceder a ellas.

7.2.3. Autenticación basic

El método de autenticación basada en formularios tiene algunos inconvenientes: si el navegador no soporta cookies, el proceso tiene que hacerse mediante reescritura de URLs, con lo que no se garantiza el funcionamiento.

Por ello, una alternativa es utilizar el modelo de autenticación *basic* de HTTP, donde se emplea un cuadro de diálogo para que el usuario introduzca su login y password, y se emplea la cabecera *Authorization* de petición para recordar qué usuarios han sido autorizados y cuáles no. Una diferencia con respecto al método anterior es que es difícil entrar como un usuario distinto una vez que hemos entrado como un determinado usuario (habría que cerrar el navegador y volverlo a abrir).

Al igual que en el caso anterior, podemos utilizar SSL sin ver modificado el resto del esquema del proceso.

El método de autenticación *basic* consta de los siguientes pasos:

1. Establecer los logins, passwords y roles

Este paso es exactamente igual que el visto para la autenticación basada en formularios.

2. Indicar al servlet que se empleará autenticación BASIC, y designar los dominios

Se utiliza la misma etiqueta `<login-config>` vista antes, pero ahora una etiqueta `<auth-method>` con valor `BASIC`. Se emplea una subetiqueta `<realm-name>` para indicar qué dominio se empleará en la autorización. Por ejemplo:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    ...
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>dominio</realm-name>
    </login-config>

```



```
...
</web-app>
```

3. Indicar qué direcciones deben protegerse con autenticación

Este paso también es idéntico al visto en la autenticación basada en formularios.

7.2.4. Anotaciones relacionadas con la seguridad

Con la especificación 3.0 de servlets se introduce la posibilidad de configurar los permisos de acceso mediante anotaciones en lugar de en el web.xml, lo que hace la configuración menos farragosa y más clara. Casi todas las anotaciones que se usan para ello en realidad no son del estándar de servlets sino que se han tomado de estándares Java ya existentes, como JSR250.

Cuidado con la versión de Tomcat

Recordemos que para desplegar servlets 3.0 como mínimo necesitaremos Tomcat 7, ya que Tomcat 6 solo es compatible con la versión 2.5. En esta versión solo se admite el uso de las anotaciones `@DeclareRoles` y `@RunAs`

- **@DeclareRoles:** especifica qué roles de seguridad están asociados a una clase concreta.

```
@WebServlet("/miServlet")
@DeclareRoles({"unrol", "otrorol"})
public class MiServlet extends HttpServlet ...
```

- **@RolesAllowed:** Indica qué roles tienen permiso para ejecutar los métodos de un servlet. Si se especifica a nivel de clase, afecta a todos sus métodos; si se hace a nivel de método, afecta sólo a ese método, y prevalece sobre lo establecido para toda la clase.

```
@WebServlet("/miServlet")
@RolesAllowed("unrol")
public class MiServlet extends HttpServlet...
    @RolesAllowed("otrorol")
    protected void doGet(...) {
        ...
    }
}
```

- **@PermitAll:** Permite a cualquier rol invocar los métodos del servlet. Si se especifica a nivel de clase, afecta a todos sus métodos; si se hace a nivel de método, afecta sólo a ese método, y prevalece sobre lo establecido para toda la clase.
- **@DenyAll:** Prohíbe a cualquier rol invocar los métodos del servlet. Si se especifica a nivel de clase, afecta a todos sus métodos; si se hace a nivel de método, afecta sólo a ese método, y prevalece sobre lo establecido para toda la clase.
- **@RunAs:** Se aplica a nivel de servlet, e indica el rol con el que se accede a sus

elementos.

- **@TransportProtected:** la conexión no se permite si no se está usando SSL. Se puede usar a nivel del servlet o de cada uno de sus métodos. Con `@TransportProtected(false)` especificamos que no es necesario SSL (nos puede servir en un método para anular la configuración global del servlet si ésta especifica SSL obligatorio).

8. Ejercicios de seguridad en Tomcat

¡Cuidado!

Antes de realizar los ejercicios debes estar seguro de que el JAR con el driver de MySQL está en el directorio `lib` de Tomcat. En caso contrario Tomcat no podrá usar la base de datos para autenticar a los usuarios.

8.1. Creación de un Realm de base de datos (1 pto)

Vamos a crear la configuración y la estructura necesarias para un realm de Tomcat con BD, que usaremos en el segundo ejercicio para asegurar la aplicación de comentarios.

1. En las plantillas de la sesión hay un script SQL que crea las tablas necesarias para almacenar logins, passwords y roles. Ejecútalo y comprueba que se han creado correctamente y que contienen usuarios creados.
2. Las plantillas también incluyen un proyecto de Eclipse llamado `testJDBCRealm` que ya está configurado para usar la BD para autenticación. **Tendrás que crear el META-INF/context.xml con la información del JDBCRealm.** Mira el ejemplo de las transparencias. Una vez creado el realm, prueba a identificarte como "espe", "espe" (debes tener permiso de acceso) y luego como "pepe", "pepe" (no lo tendrás). Ten en cuenta que como la aplicación usa autenticación BASIC debes cerrar el navegador si quieres entrar con otro usuario (en Eclipse esto no es posible, así que tendrás que hacerlo desde un navegador externo).

8.2. Seguridad basada en formularios (1 pto)

Vamos a asegurar la aplicación de comentarios con seguridad basada en formularios, de modo que solamente los usuarios con rol "registrado" puedan insertar comentarios. Para ello, tendrás que:

1. Crear la configuración del realm en el `context.xml` de la aplicación. Puedes tomar como ejemplo la de la aplicación `testJDBCRealm` del ejercicio anterior
2. Crear una página `login.html` con un formulario de autenticación como se indica en los apuntes.
3. Configurar la seguridad protegiendo la URL `"/addComentario"` para que solo puedan acceder a ella los usuarios con rol "registrado". Usa las etiquetas vistas en los apuntes para configurar seguridad basada en formularios.
4. Comprobar que al intentar añadir un comentario se salta a la página de login y tras introducir usuario y password válido se permite el acceso

8.3. Seguridad del servidor (1 punto)

En las plantillas de la sesión tienes una aplicación llamada `visitas.war` muy similar a la de

comentarios, pero que guarda los comentarios enviados por los usuarios en un fichero. En este ejercicio no es necesario que uses Eclipse

1. Despliega el war copiándolo manualmente al directorio webapps.
2. Arranca manualmente Tomcat entrando en su directorio bin y ejecutando `./startup.sh -security` para activar las políticas de seguridad. Comprueba que la aplicación a un error al intentar guardar el comentario, ya que por defecto las aplicaciones web no pueden crear ficheros con las restricciones de seguridad activas.
3. Modifica el fichero de Tomcat `conf/catalina.policy` añadiendo al final un permiso para guardar archivos.
 - Toma como modelo el ejemplo de las transparencias para darle el permiso en este caso a la aplicación "visitas"
 - El "permission" para guardar ficheros es `java.io.FilePermission`
`"directorio_donde_hay_permiso", "read,write"`
4. Para Tomcat (`./shutdown.sh`) y vuelve a arrancarlo de nuevo con la seguridad activada (`./startup.sh -security`) y comprueba que ahora sí se guarda el archivo `"prueba.txt"`

