Componentes Web

Miguel Angel Lozano <malozano@ua.es>

Table of Contents

1. Introducción a las aplicaciones web 5
1.1. ¿Qué es una aplicación web? 5
Aplicaciones en el lado del servidor 5
Aplicaciones en el lado del cliente 5
1.2. Aplicaciones web Java EE 5
Estructura de la aplicación 5
Rutas relativas al contexto 6
Empaquetamiento de aplicaciones web en ficheros WAR 7
1.3. Creación y despliegue de aplicaciones web con IntelliJ y WildFly 8
Creación de un proyecto web con IntelliJ 8
Despliegue del proyecto en WildFly desde línea de comando 9
Despliegue del proyecto en WildFly con IntelliJ 10
1.4. Componentes de la aplicación Java EE 13
El descriptor de despliegue 13
Servlets 16
JSP 27
1.5. Ejercicios 30
Aplicación web en WildFly (0.5 puntos) 30
Hola mundo con IntelliJ (0.5 puntos) 31
Servlet que muestra la fecha y hora actuales (0.5 puntos) 32
Servlet que muestra parámetros de inicio (0.5 puntos) 33
Configurar *logging* en servlets (0.5 puntos) 33
2. Procesamiento de peticiones 34
2.1. Petición y respuesta HTTP 34
Peticiones del cliente 34
Respuestas del servidor 36
Cabeceras 36
Códigos de estado 38
Peticiones: HttpServletRequest 39
Respuestas: HttpServletResponse 40
2.2. Procesamiento de peticiones GET y POST 41
Procesamiento secuencial de peticiones 42
Manejo de formularios 43
Ejemplo 43
2.3. Cabeceras y códigos 45
Cabeceras de petición 45
Cabeceras de respuesta 45
Variables CGI 46
Códigos de estado HTTP 47
2.4. Procesamiento asíncrono 48
Procesamiento de operaciones de larga duración 48
Entrada y salida no bloqueante 49
2.5. Ejemplos 50
Ejemplo de cabeceras de petición 50
Ejemplo de cabeceras de respuesta 51

- Ejemplo de autentificación ... 55
- 2.6. Ejercicios ... 58
  - Recogida de parámetros del usuario (0,3 puntos) ... 58
  - Trabajando con redirecciones (0,4 puntos) ... 58
  - Un buscador sencillo (0,4 puntos) ... 58
  - Distinguir el navegador (0,3 puntos) ... 58
  - Redirecciones con retardo (0,3 puntos) ... 59
  - Loggear variables CGI (0,3 puntos) ... 59
- 3. Manejo de Cookies y Sesiones ... 60
  - 3.1. Cookies ... 60
    - Cookies en HTTP ... 60
    - Enviar una cookie ... 61
    - Obtener una cookie ... 62
    - Ejemplo ... 63
  - 3.2. Seguimiento de sesiones ... 65
    - Obtener una sesión ... 65
    - Guardar y obtener datos de la sesión ... 65
    - Invalidar la sesión ... 66
    - Compatibilidad con los navegadores ... 67
    - Oyentes ... 68
    - Ejemplos ... 69
  - 3.3. Ejercicios ... 72
    - Personalizar un sitio web (0.3 puntos) ... 72
    - Carro de la compra (0.5 puntos) ... 72
    - Mejoras para el carro de la compra (0.2 puntos) ... 73
- 4. Contexto global de la aplicación web ... 74
  - 4.1. Contexto de los servlets ... 74
    - Atributos de contexto ... 75
    - Parámetros de inicialización ... 75
    - Acceso a recursos estáticos ... 76
    - Redirecciones ... 76
    - Otros métodos ... 77
    - Listeners de contexto ... 78
    - Declaración dinámica de servlets ... 79
  - 4.2. Inyección de dependencias ... 80
    - Configuración de Weld ... 80
    - Managed beans ... 81
    - Inyección de objetos ... 81
    - Ámbito de los beans ... 81
    - Clasificadores ... 82
    - Productores ... 83
    - Definición de alternativas ... 84
  - 4.3. Ejercicios ... 86
    - Ejemplo de contexto (0 puntos) ... 86
    - Chat con servlets (0.7 punto) ... 86
    - Inyección de dependencias (0.3 puntos) ... 87
- 5. WebSocket ... 89
  - 5.1. WebSocket en Java EE ... 89
    - Endpoints programados ... 89
    - Endpoints mediante anotaciones ... 90
    - Mantenimiento del estado ... 91
  - 5.2. Intercambio de mensajes ... 91
    - Envío de mensajes ... 91

Recepción de mensajes ... 93
5.3. Conversión entre Java y mensajes WebSocket ... 93
Encoders ... 94
Decoders ... 94
Uso de *encoders* y *decoders* ... 95
5.4. Parámetros del *path* y de la *query* ... 95
5.5. Cliente JavaScript ... 97
5.6. Ejercicios ... 98
Chat básico con WebSocket (0.5 puntos) ... 98
Chat con nombres de usuario y salas (0.5 puntos) ... 98
6. Seguridad en aplicaciones web ... 100
6.1. Mecanismos de autentificación ... 100
6.2. Usuarios y roles ... 102
6.3. Autentificación en aplicaciones web Java EE ... 102
Autentificación basada en formularios ... 103
Autentificación *basic* ... 106
6.4. Anotaciones relacionadas con la seguridad ... 107
6.5. Acceso a la información de seguridad ... 108
6.6. Ejercicios ... 110
Seguridad básica (0.3 puntos) ... 110
Seguridad basada en formularios (0.4 puntos) ... 110
Seguridad mediante anotaciones (0.3 puntos) ... 110
7. Filtros y Wrappers ... 111
7.1. Filtros ... 111
¿Qué es un filtro? ... 111
Funcionalidades de los filtros ... 112
Aplicaciones de los filtros ... 112
Configuración de un filtro ... 112
Implementación básica de un filtro ... 114
Acceso al contexto ... 116
Ciclo de vida de un filtro ... 116
7.2. Wrappers ... 117
¿Qué es un wrapper? ... 117
Implementación de un wrapper ... 118
Utilización de un wrapper ... 120
7.3. Ejemplos ... 120
Acceso restringido ... 120
Ranking de páginas más visitadas ... 122
Extracción automática de información ... 123
7.4. Ejercicios ... 125
Filtro de acceso restringido (0 puntos) ... 125
Restringir el acceso al chat (0.5 puntos) ... 125
Wrapper de ejemplo (0 puntos) ... 125
Registro de accesos (0.5 puntos) ... 125
8. Facelets, JSTL y lenguajes de expresiones ... 127
8.1. JavaServer Faces y Facelets ... 127
8.2. Introducción a los Facelets ... 127
Mapeo del *servlet* de JSF ... 128
Estructura básica de un *Facelet* ... 129
Declaración de librerías de etiquetas ... 129
8.3. Lenguaje de expresiones ... 130
Introducción al lenguaje de expresiones ... 130
Atributos y expresiones ... 131

Operadores 132
Nombres de variables 133
Lenguaje de expresiones y CDI 134
8.4. Librerías de etiquetas 134
Librería de etiquetas JSTL 135
Librerías de JSF/*Facelets* 145
8.5. Ejercicios 149
Página de chat con *Facelets* (1 punto) 149

1. Introducción a las aplicaciones web

1.1. ¿Qué es una aplicación web?

Una aplicación web es una aplicación a la que accedemos mediante protocolo HTTP utilizando un navegador web. 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.

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.

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. Aplicaciones web Java EE

Vamos a centrarnos en las aplicaciones web Java EE, 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).

Estructura de la aplicación

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

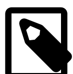
- En el directorio raíz de la aplicación se colocan los **documentos web** HTML o JSP, junto a los recursos que necesiten, como por ejemplo imágenes, hojas de estilo, ficheros de código

JavaScript u otros ficheros referenciados desde los documentos web (podemos dividirlas 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:

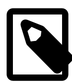
  # Fichero **descriptor de despliegue** de la aplicación: es el fichero descriptor de la aplicación web. Es un fichero XML (llamado `web.xml`) que contiene información genérica sobre la aplicación. Lo veremos con más detalle más adelante.

  # 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 `paquete1.subpaquete1.MiClase` dentro de `classes`, se quedará almacenada en el directorio `classes/paquete1/subpaquete1/MiClase`.



Es recomendable utilizar como nombre de paquete nuestra URL al revés, por ejemplo si tenemos la dirección `expertojava.org` podríamos utilizar para nuestros proyectos paquetes como `org.expertojava.web.servlets`.

  # 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)



Los ficheros `.class` se separan 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:

```
<img src="/aplic/imagenes/miImagen.jpg">
```

Empaquetamiento de aplicaciones web en ficheros WAR

Una forma de distribuir aplicaciones Web Java EE 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 Java EE, por lo que podremos utilizarlos en los diferentes servidores de aplicaciones Java EE 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 para un servidor de aplicaciones concreto en el siguiente apartado.

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 Java EE 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. Creación y despliegue de aplicaciones web con IntelliJ y WildFly

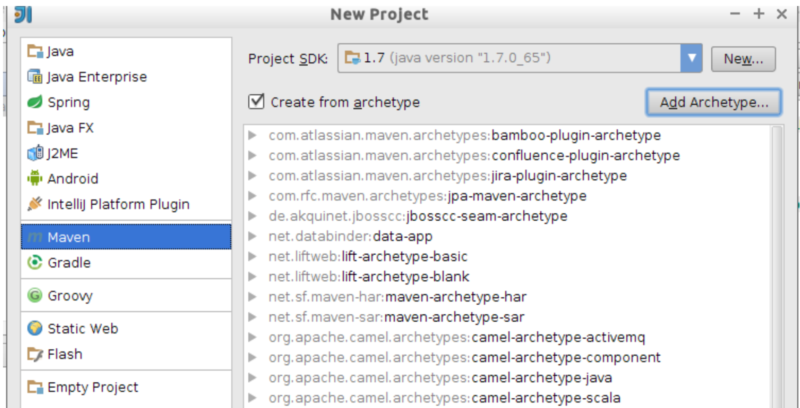
Vamos a pasar ahora a ver la forma de crear y desplegar aplicaciones web Java EE con un IDE y un servidor de aplicaciones concreto. Vamos a utilizar el IDE IntelliJ IDEA y el servidor de aplicaciones WildFly (JBoss 8.1).

Creación de un proyecto web con IntelliJ

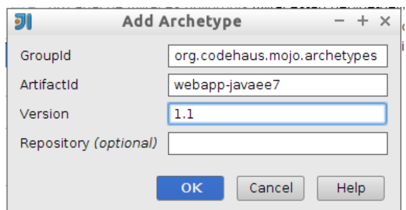
Empezaremos creando un nuevo proyecto web Java EE desde el IDE IntelliJ. Encontramos dos formas de hacer esto:

- Crearlo como proyecto web IntelliJ, con *New Project* > *Java Enterprise* > *Web Application*. El proyecto web tendrá ficheros de configuración propios de este IDE.
- Crearlo como proyecto Maven dentro de IntelliJ, con *New Project* > *Maven*, y seleccionando un arquetipo que genere un proyecto web Java EE 7. De esta forma tendremos un proyecto que se podrá construir con Maven fuera del entorno IntelliJ y podrá ser importado por otros IDEs, y que además contará con los ficheros de configuración propios de IntelliJ para poder trabajar con él dentro de este entorno. Cuando se produzcan cambios en la configuración de Maven del proyecto, podremos importar dichos cambios a la configuración del proyecto IntelliJ.

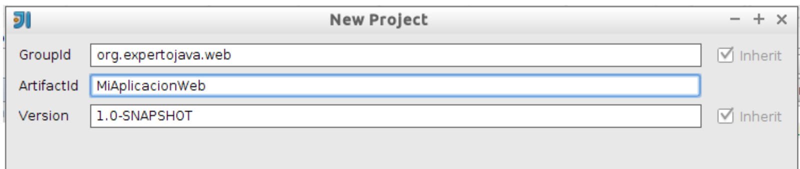
Vamos a utilizar la segunda opción, dada la flexibilidad que nos proporciona Maven para poder construir el proyecto de forma independiente al IDE utilizado. En la pantalla de creación del proyecto deberemos marcar la casilla *Create from archetype*, y pulsar sobre el botón *Add archetype …*, ya que el arquetipo que vamos a necesitar (aplicación web Java EE 7) no está en la lista por defecto:



Del arquetipo introduciremos los siguientes datos:

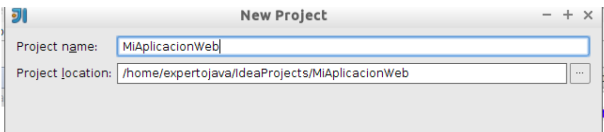


Una vez seleccionado el arquetipo deberemos introducir la configuración del módulo Maven (`artifactId`, `groupId`, y `version`):

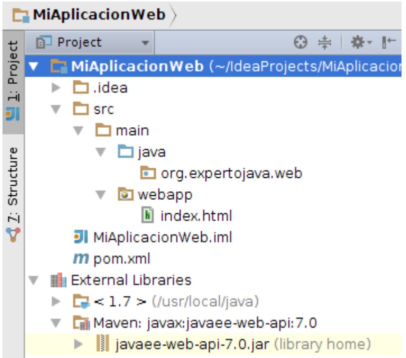



```
<groupId>org.codehaus.mojo.archetypes</groupId>
<artifactId>webapp-javaee7</artifactId>
<version>1.1</version>
```

Tras esto, introduciremos el nombre que va a tener el proyecto dentro de IntelliJ. Esto ya no es configuración de Maven, sino del propio IDE:



Con esto habremos finalizado, y veremos nuestro nuevo proyecto con la siguiente organización:



Es posible que durante la creación del proyecto IntelliJ nos pregunte repetidas veces si queremos recargar la información de Maven en el proyecto. Responderemos siempre que si y podemos indicar que esta información se importe de forma automática cuando haya cambios (*Enable Auto-Import*).

Despliegue del proyecto en WildFly desde línea de comando

Podemos desplegar nuestro proyecto en el servidor de aplicaciones WildFly directamente desde línea de comando, al igual que ocurre con la mayoría de servidores de aplicaciones Java EE.

En primer lugar deberemos poner el servidor WildFly en marcha. Por el momento será suficiente con poner el servidor *standalone*, que contiene una única instancia del servidor de aplicaciones. Para poner en marcha el servidor ejecutaremos el siguiente comando desde el directorio `$WILDFLY_HOME/bin`, siendo `$WILDFLY_HOME` el directorio de instalación de este servidor de aplicaciones:

```
$ ./standalone.sh
```

Con el servidor WilfFly en marcha podremos desplegar una aplicación simplemente "dejando caer" el fichero *war* en el directorio `$WILDFLY_HOME/standalone/desployments`. Pasados unos segundos desde la copia del fichero la aplicación se habrá desplegado y podremos acceder a ella mediante un navegador. Por ejemplo, si copiamos un fichero `miaplicacion.war`, por defecto podremos acceder a ella mediante:

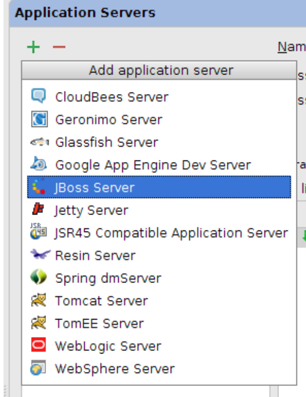
http://localhost:8080/miaplicacion/

Despliegue del proyecto en WildFly con IntelliJ

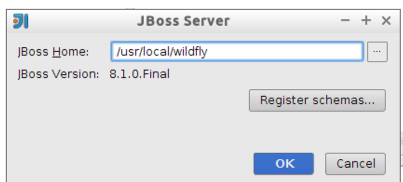
Una vez tenemos un proyecto web creado podemos desplegarlo en un servidor de aplicaciones. Para ello deberemos en primer lugar configurar en IntelliJ el servidor donde queramos desplegar la aplicación, y en segundo lugar crear un perfil de ejecución que le indique a WildFly cómo debe hacer el despliegue de nuestra aplicación web.

Configuración de WildFly en IntelliJ

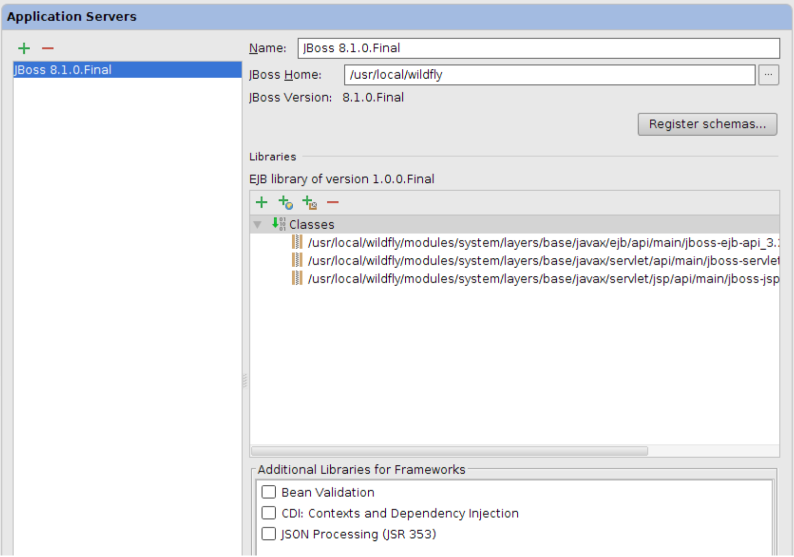
En primer lugar deberemos configurar el servidor de aplicaciones a utilizar. Para ello entraremos en *File* > *Settings ...* > *Application Servers*. En dicha sección pulsaremos sobre el botón para añadir un nuevo servidor de tipo JBoss:



Nos pedirá el directorio donde está instalado WildFly:



Tras introducirlo veremos el servidor JBoss configurado y ya podremos desplegar el proyecto en él:



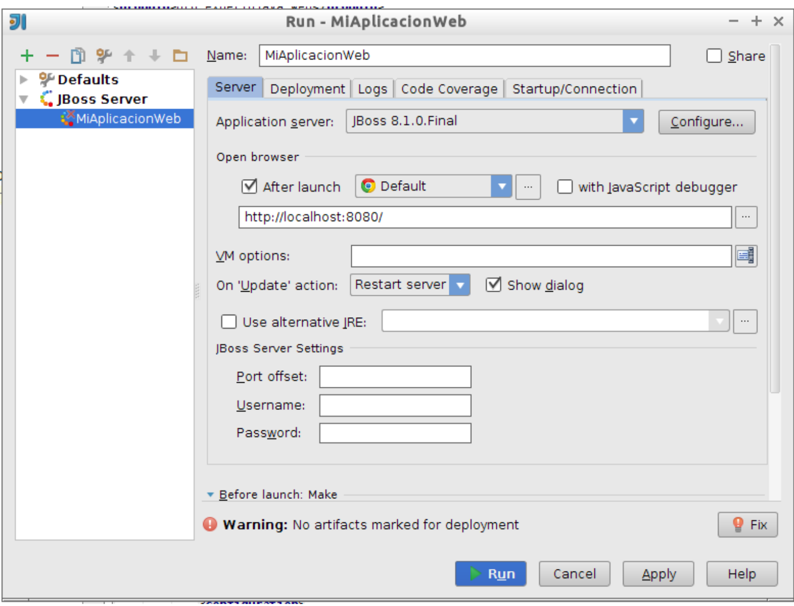
También se podría configurar el servidor en el momento de la creación del perfil de ejecución.

Creación de un perfil de ejecución

Para poder desplegar nuestra aplicación web deberemos crear un perfil de ejecución para ella dentro de IntelliJ.

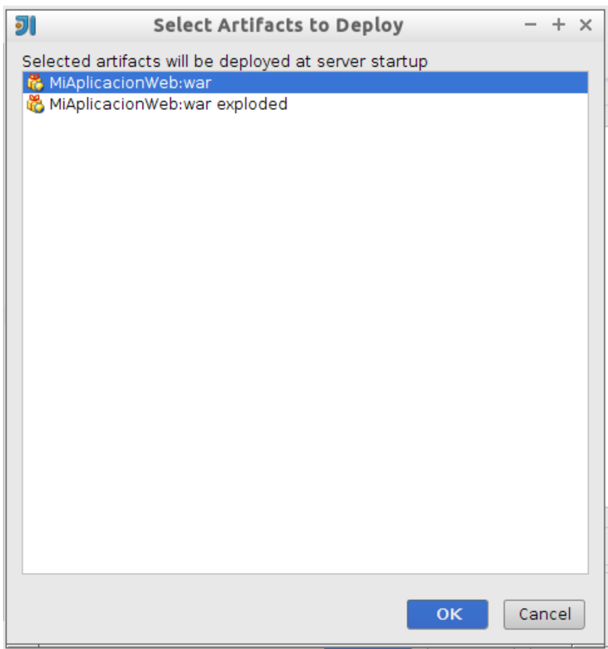
Necesitaremos un perfil de ejecución por cada aplicación web que queramos poder desplegar.

En primer lugar seleccionaremos *Run > Edit configurations …*. Dentro de esta pantalla pulsaremos el botón + y añadiremos un nuevo perfil de tipo *JBoss Server > Local*. Veremos un formulario como el siguiente:



En el campo *Name* podemos darle un nombre al nuevo perfil de ejecución. Si ya hemos configurado el servidor JBoss, veremos que ya aparece seleccionado como servidor en el que desplegar. En caso contrario, podríamos configurarlo desde esta pantalla.

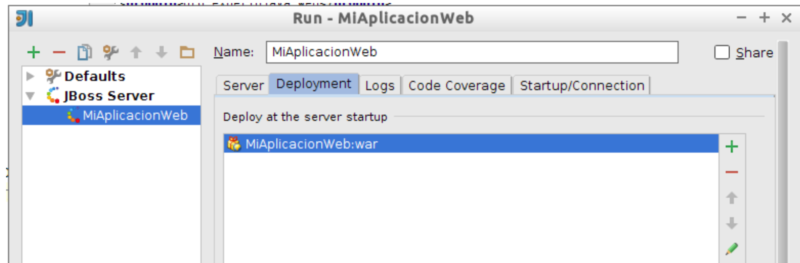
Veremos también que en la parte inferior aparece un *Warning* indicando que no hay seleccionados artefactos para ser desplegados. Deberemos indicar qué artefacto va a ser desplegado en el servidor cuando ejecutemos este perfil. Podemos indicarlo pulsando sobre el botón *Fix* o pasando de forma manual a la pestaña *Deployment*. Desde esta pestaña podemos añadir un nuevo artefacto a desplegar pulsando sobre el botón + :



Tenemos dos artefactos configurados por defecto:

- *war*: Despliega la aplicación web empaquetada en un fichero WAR
- *war exploded*: Despliega la aplicación web publicando el directorio sin empaquetar.

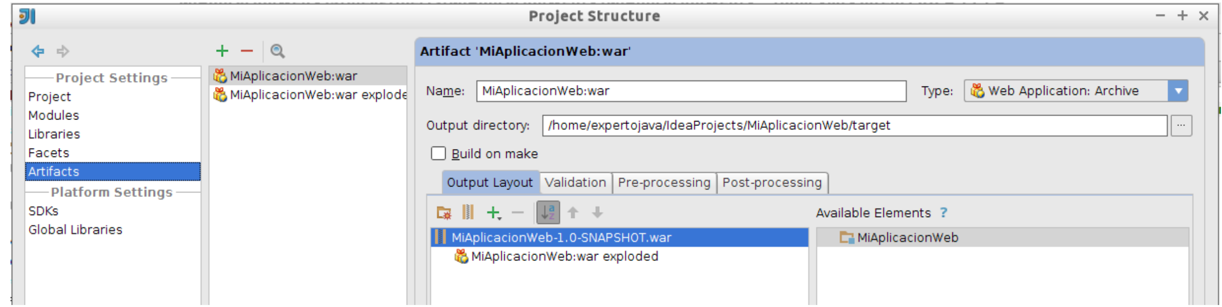
Seleccionaremos uno de ellos y nos aparecerá como artefacto a desplegar:



Ahora ya podremos guardar el perfil de ejecución y pulsar el botón *Run* para realizar el despliegue. A partir de este momento podremos lanzar este perfil en cualquier momento para volver a realizar el despliegue de la aplicación.

Configuración de artefactos

Podemos configurar los artefactos generados por el proyecto en la pantalla *File > Project Structure ... > Artifacts*. Aquí veremos los dos artefactos que se crear por defecto para las aplicaciones web, y podremos editarlos o crear nuevos:



Por ejemplo, podríamos modificar el nombre del fichero WAR para así cambiar el nombre del contexto donde se desplegará la aplicación.

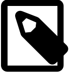
Si queremos cambiar el nombre del contexto en el que se despliega la aplicación es mejor hacerlo introduciendo en el fichero `pom-xml` el elemento <finalName> dentro de la etiqueta <build>.

```
<project>
    ...
    <build>
        <finalName>MiAplicacionWeb</finalName>
        ...
    </build>
</project>
```

Depuración y cambios en caliente

El entorno IntelliJ nos permite realizar cambios en caliente dentro de nuestra aplicación web, es decir, conforme modificamos el código de la aplicación se aplicarán los cambios a la aplicación en ejecución sin necesidad de volver a desplegar. Para poder realizar cambios en caliente deberemos desplegar de la siguiente forma:

1. Seleccionaremos como artefacto a generar la aplicación web descomprimida (*war exploded*).

   Deberemos renombrar el artefacto de tipo *war exploded* de forma que acabe con `.war`, ya que de no ser así obtendremos un error en IntelliJ.

2. Desplegaremos la aplicación en modo *debug*.

Con esto, cada vez que realicemos un cambio se aplicará a la aplicación ya desplegada en el servidor. Además, la ejecución en modo *debug* nos permitirá añadir puntos de parada y realizar la ejecución paso a paso inspeccionando el valor de cada variable.

1.4. Componentes de la aplicación Java EE

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. Este descriptor de despliegue es el mecanismo estándar para configurar aplicaciones web JavaEE.

A partir de la API de servlets 3.0 (Java EE 7) el descriptor de despliegue es opcional, pero es recomendable contar con él ya que será necesario para configurar determinados elementos.

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 WildFly.

Estructura del descriptor de despliegue

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_3_0.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">

 <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 3.0 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 los elementos de configuración general de la aplicación. Conforme veamos cada característica de la API de servlets, repasaremos los elementos del descriptor del despliegue que hagan referencia a ella.

Información general de la aplicación

Tenemos dos etiquetas que nos permiten especificar información sobre la aplicación que se mostrará cuando se presente en una interfaz gráfica, y nos servirán para identificarla:

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

Páginas de inicio

Podemos también indicar la página (o posibles páginas) que se devolverán por defecto si no se especifica ninguna concreta en la URL:

- `<welcome-file-list>` para indicar qué páginas debe buscar el servidor 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>
```

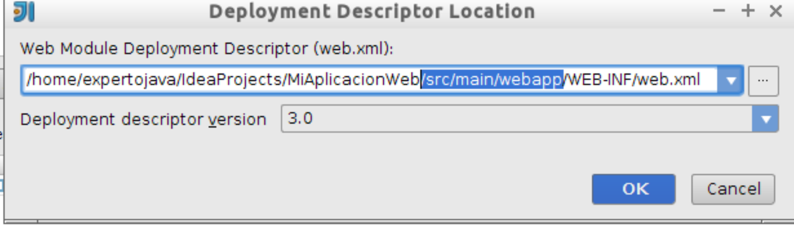
Las páginas se buscan en el orden en que se especifican en esta etiqueta.

Creación del descriptor de despliegue con IntelliJ

Al crear un proyecto Java EE 7 (o superior) con IntelliJ nos da la opción de crear o no el descriptor de despliegue. Esto es debido a que, como se ha comentado anteriormente, en esta versión de la plataforma es opcional contar con este fichero. Cuando el proyecto es creado con el arquetipo `webapp-javaee7` de Maven el descriptor de despliegue no se crea. A pesar de ser un fichero opcional, si no se ha creado de forma automática al crear el proyecto es recomendable crearlo posteriormente.

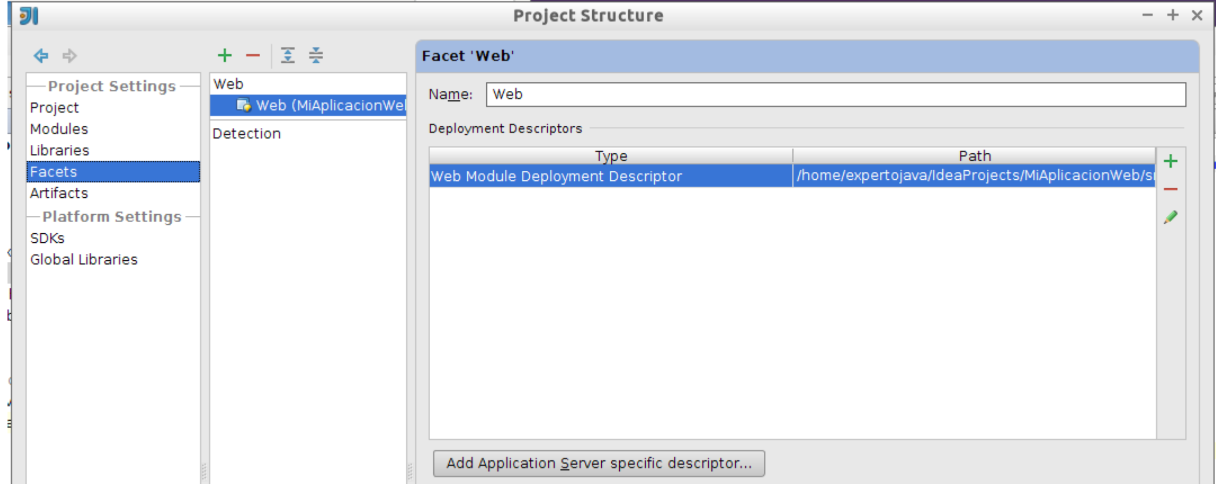
En algunas versiones de IntelliJ no funciona correctamente la creación de servlets si no se cuenta con un descriptor de despliegue. Se trata de un *bug*, ya que no es necesario contar con un descriptor de despligue para crear servlets en Java EE 7.

Para crear el descriptor de despliegue con IntelliJ en un proyecto que todavía no cuenta con él, podemos entrar en *File* > Project Structure …_ > *Facets*, y seleccionar el *facet Web*. Dentro de esta pantalla pulsaremos el botón `+` para crear un nuevo descriptor de despliegue en la aplicación. Deberemos indicar el directorio donde crearlo, que al ser un proyecto de Maven deberá ser el siguiente:



En el proyecto Maven IntelliJ no detecta correctamente el directorio donde crear el fichero `web.xml`. Es importante que esté en `src/main/webapp/WEB-INF`.

Una vez creado, veremos que la aplicación ya cuenta con un *Web Module Deployment Description*:



Servlets

Un **servlet** es un programa Java que se ejecuta en un servidor Web y construye o sirve páginas web. De esta forma se pueden construir páginas dinámicas, basadas en diferentes fuentes variables: datos proporcionados por el usuario, fuentes de información variable (páginas de noticias, por ejemplo), o programas que extraigan información de bases de datos.

Comparado con un CGI, un servlet es más sencillo de utilizar, más eficiente (se arranca un hilo por cada petición y no un proceso entero), más potente y portable. Con los servlets podremos, entre otras cosas, procesar, sincronizar y coordinar múltiples peticiones de clientes, reenviar peticiones a otros servlets o a otros servidores, etc.

Recursos de servlets y JSP

Normalmente al hablar de servlets se habla de JSP y viceversa, puesto que ambos conceptos están muy interrelacionados. Para trabajar con ellos se necesitan tener presentes algunos recursos:

- Un **servidor web** que dé soporte a servlets / JSP (contenedor de servlets y páginas JSP). Ejemplos de estos servidores son Apache Tomcat, Resin, JRun, Java Web Server, BEA WebLogic, etc.
- Las **librerías** (clases) necesarias para trabajar con servlets / JSP. Normalmente vienen en ficheros JAR en un directorio `lib` del servidor. Al desarrollar nuestra aplicación, deberemos tener las librerías necesarias en el *classpath* para que compilen los ficheros (sólo necesitaremos compilar los servlets, no los JSP).
- La **documentación** sobre la API de servlets / JSP (no necesaria, pero sí recomendable)

Cuando trabajemos con un entorno como IntelliJ, al crear un proyecto de aplicación web se añadirán de forma automática referencias a las librerías de componentes web del servidor.

Arquitectura del paquete servlet

Dentro del paquete `javax.servlet` tenemos toda la infraestructura para poder trabajar con servlets. El elemento central es la interfaz `Servlet`, que define los métodos para cualquier servlet. La clase `GenericServlet` es una clase abstracta que implementa dicha interfaz para un servlet genérico, independiente del protocolo. Para definir un servlet que se utilice vía

web, se tiene la clase `HttpServlet` dentro del subpaquete `javax.servlet.http`. Esta clase hereda de `GenericServlet`, y también es una clase abstracta, de la que heredaremos para construir los servlets para nuestras aplicaciones web. Cuando un servlet acepta una petición de un cliente, se reciben dos objetos:

- Un objeto de tipo `ServletRequest` que contiene los datos de la petición del usuario (toda la información entrante). Con esto se accede a los parámetros pasados por el cliente, el protocolo empleado, etc. Se puede obtener también un objeto `ServletInputStream` para obtener datos del cliente que realiza la petición. La subclase `HttpServletRequest` procesa peticiones de tipo HTTP.
- Un objeto de tipo `ServletResponse` que contiene (o contendrá) la respuesta del servlet ante la petición (toda la información saliente). Se puede obtener un objeto `ServletOutputStream`, y un `Writer`, para poder escribir la respuesta. La clase `HttpServletResponse` se emplea para respuestas a peticiones HTTP.

Ciclo de vida de un servlet

Todos los servlets tienen el mismo ciclo de vida:

- Un servidor **carga e inicializa** el servlet.
- El servlet **procesa** cero o más peticiones de clientes. Por cada petición se lanza un hilo, ejecutándose estos hilos de forma concurrente en sobre un mismo objeto *servlet*.
- El servidor **destruye** el servlet, normalmente en momentos en los que no tiene peticiones o cuando se apaga el servidor.

1. Inicialización

En cuanto a la inicialización de un servlet, se tiene una por defecto en el método `init()`.

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

public void init(ServletConfig conf) throws ServletException
{
  super.init(conf);
  ...
}
```

El primer método se utiliza si el servlet no necesita parámetros de configuración externos. El segundo se emplea para tomar dichos parámetros del objeto `ServletConfig` que se le pasa. La llamada a `super.init(…)` al principio del método es MUY importante, porque el servlet utiliza esta configuración en otras zonas. Si queremos definir nuestra propia inicialización, deberemos sobreescribir alguno de estos métodos. Si ocurre algún error al inicializar y el servlet no es capaz de atender peticiones, debemos lanzar una excepción de tipo `UnavailableException`. Podemos utilizar la inicialización para establecer una conexión con una base de datos (si trabajamos con base de datos), abrir ficheros, o cualquier tarea que se necesite hacer una sola vez antes de que el servlet comience a funcionar.

2. Procesamiento de peticiones

Una vez inicializado, cada petición de usuario lanza un hilo que llama al método `service()` del servlet.

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

Este método obtiene el tipo de petición que se ha realizado (GET, POST, PUT, DELETE). Dependiendo del tipo de petición que se tenga, se llama luego a uno de los métodos:

- `doGet()`:

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

Para peticiones de tipo GET (aquellas realizadas al escribir una dirección en un navegador, pinchar un enlace o rellenar un formulario que no tenga METHOD=POST)

- `doPost()`:

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

Para peticiones POST (aquellas realizadas al rellenar un formulario que tenga METHOD=POST)

- `doXXX()`: normalmente sólo se emplean los dos métodos anteriores, pero se tienen otros métodos para peticiones de tipo DELETE (`doDelete()`), PUT (`doPut()`), OPTIONS (`doOptions()`) y TRACE (`doTrace()`).

3. Destrucción

El método `destroy()` de los servlets se emplea para eliminar un servlet y sus recursos asociados.

```
public void destroy() throws ServletException
```

Aquí debe deshacerse cualquier elemento que se construyó en la inicialización (cerrar conexiones con bases de datos, cerrar ficheros, etc). El servidor llama a `destroy()` cuando todas las llamadas de servicios del servlet han concluido, o cuando haya pasado un determinado número de segundos (lo que ocurra primero). Si esperamos que el servlet haga tareas que requieran mucho tiempo, tenemos que asegurarnos de que dichas tareas se completarán. Podemos hacer lo siguiente:

- Definir un contador de tareas activas, que se incremente cada vez que una tarea comienza (entendemos por *tarea* cada petición que se realice al servlet), y se decremente cada

vez que una termina. Podemos utilizar bloques de código `synchronized` para evitar problemas de concurrencia.

- Hacer que el método `destroy()` no termine hasta que lo hagan todas las tareas pendientes (comprobando el contador de tareas pendientes)
- Hacer que las tareas pendientes terminen su trabajo si se quiere cerrar el servlet (comprobando algún flag que indique si el servlet se va a cerrar o no).

Estructura básica de un servlet

La plantilla común para implementar un servlet es:

```
import javax.servlet.*;
import javax.servlet.http.*;

public class ClaseServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
               throws ServletException, IOException {
    // ... codigo para una peticion GET
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
               throws ServletException, IOException {
    // ... codigo para una peticion POST
  }
}
```

El servlet hereda de la clase `HttpServlet`. Normalmente se deben sobreescribir los métodos `doGet()`, `doPost()` o ambos, colocando el código que queremos que se ejecute cuando se reciba una petición GET o POST, respectivamente. Conviene definir los dos para distinguir ambas peticiones. En caso de que queramos hacer lo mismo para GET o POST, definimos el código en uno de ellos, y hacemos que el otro lo llame. Aparte de estos métodos, podemos utilizar otros de los que hemos visto: `init()` (para inicializaciones), `doXXX()` (para tratar otros tipos de peticiones (PUT, DELETE, etc)), `destroy()` (para finalizar el servlet), etc, así como nuestros propios métodos internos de la clase.

Configuración de servlets en aplicaciones web

Para instalar un servlet en una aplicación web, se coloca la clase compilada del servlet dentro del directorio `WEB-INF/classes` de la aplicación (respetando también la estructura de paquetes, creando tantos subdirectorios como sea necesario).

De forma alternativa, también podríamos empaquetar nuestros servlets en un JAR y poner esta librería de clases dentro del directorio `WEB-INF/lib`. De cualquiera de las dos formas la clase del servlet estará localizable para la aplicación. Veremos ahora las formas que tenemos de invocar a ese servlet.

Mapeo de servlets en el fichero descriptor

Los servlets se invocarán cuando desde un cliente hagamos una petición a una URL determinada. Para especificar la URL a la que está asociada cada servlet, deberemos configurar dicho servlet en el fichero descriptor de despliegue (`web.xml`).

En primer lugar deberemos introducir una marca `<servlet>` para declarar cada servlet de la siguiente forma:

```
<servlet>
  <servlet-name>nombre</servlet-name>
  <servlet-class>unpaquete.ClaseServlet</servlet-class>
</servlet>
```

Donde `<servlet-name>` es un nombre identificativo y arbitrario del servlet, y `<servlet-class>` es la clase del servlet.

Al declarar un servlet, debe indicarse el nombre completo de la clase en la que está implementado, incluyendo paquetes y subpaquetes. Esto es así porque en el `web.xml` no tenemos ningún "import" que nos permita desambiguar entre posibles diferentes clases con el mismo nombre.

Una vez declarado nuestro servlet, deberemos mapearlo a una URL. Esto se consigue mediante las etiquetas `<servlet-mapping>`:

```
<servlet-mapping>
  <servlet-name>nombre</servlet-name>
  <url-pattern>/ejemploservlet</url-pattern>
</servlet-mapping>
```

En la subetiqueta `<servlet-name>` se pone el nombre del servlet al que se quiere asignar la URL (será uno de los nombres dados en alguna etiqueta `<servlet>` previa), y en `<url-pattern>` colocamos la URL que le asignamos al servlet, que debe comenzar con '/'.

Destacamos que primero se colocan todas las etiquetas `<servlet>`, y luego las `<servlet-mapping>` que se requieran. En las actuales versiones de los servidores web el orden es indiferente, pero si queremos garantizar la compatibilidad con versiones anteriores, deberemos respetarlo.

Así, con lo anterior, podremos llamar al servlet identificado con `nombre` accediendo a la URL a la que se encuentra mapeado:

http://localhost:8080/<dir>/ejemploservlet

siendo `<dir>` el directorio en el que se encuentra el contexto de nuestra aplicación Web.

También podemos asignar en `<url-pattern>` expresiones como:

```
<servlet-mapping>
  <servlet-name>nombre</servlet-name>
  <url-pattern>/ejemploservlet/*</url-pattern>
</servlet-mapping>
```

o como:

```
<servlet-mapping>
```

```
  <servlet-name>nombre</servlet-name>
  <url-pattern>/ejemploservlet/*.do</url-pattern>
</servlet-mapping>
```

Con el primero, cualquier URL del directorio de nuestra aplicación Web que comience con `/ejemploservlet/` se redirigirá y llamará al servlet identificado con `nombre`. Por ejemplo, las direcciones:

http://localhost:8080/<dir>/ejemploservlet/unapagina.html

http://localhost:8080/<dir>/ejemploservlet/login.do

acabarían llamando al servlet `nombre`.

Con el segundo, cualquier llamada a un recurso acabado en `.do` del directorio `/ejemploservlet/` de nuestra aplicación se redirigiría al servlet `nombre`. Podemos hacer que distintas URLs llamen a un mismo servlet, sin más que añadir varios grupos `<servlet-mapping>`, uno por cada patrón de URL diferente, y todos con el mismo `<servlet-name>`.

Este mismo procedimiento se puede aplicar también si en lugar de un servlet queremos tratar una página JSP. Para declarar una página **JSP**, sustituiremos la etiqueta `<servlet-class>` por la etiqueta `<jsp-file>`:

```
<servlet>
  <servlet-name>nombre2</servlet-name>
  <jsp-file>/mipagina.jsp</jsp-file>
</servlet>
```

Esta página se podrá mapear a diferentes URLs de la misma forma en la que lo hacemos para un servlet.

Mapeo de servlets mediante anotaciones

En la API de Servlets 3.0 se incluyen una serie de importantes novedades dirigidas principalmente a facilitar el desarrollo de los componentes web. Esto se consigue mediante el uso de anotaciones para configurar los servlets, en lugar de tener que declararlos en el fichero `web.xml`. Esto supone un cambio importante en la forma de trabajar con servlets. Ahora bastará con anotar la clase en la que implementamos el servlet de la siguiente forma:

```
@WebServlet(name="miServlet", urlPatterns="/UrlServlet")
public class ClaseServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) {
        ...
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response) {
        ...
    }
}
```

En la anotación `@WebServlet` se debe especificar obligatoriamente el atributo `urlPatterns`, con el que se especifica la URL a la que queremos mapear el servlet. Si no necesitamos más parámetros que la URL, podemos especificarla como valor por defecto de la anotación:

```
@WebServlet("/UrlServlet")
```

Con esto ya no se necesitará declarar el servlet en el fichero `web.xml`.

Asignar parámetros de inicio a un servlet o página JSP

El hecho de asignar un nombre a un servlet o página JSP mediante la etiqueta `<servlet>` y sus subetiquetas nos permite identificarlo con ese nombre, y también poderle asignar parámetros de inicio. Para asignar parámetros se colocan etiquetas `<init-param>`dentro de la etiqueta `<servlet>` del servlet o página JSP al que le queremos asignar parámetros. Dichas etiquetas tienen como subetiquetas un `<param-name>`(con el nombre del parámetro) y un `<param-value>` (con el valor del parámetro). Por ejemplo:

```
<servlet>
  <servlet-name>nombre</servlet-name>
  <servlet-class>ClaseServlet</servlet-class>
  <init-param>
    <param-name>param1</param-name>
    <param-value>valor1</param-value>
  </init-param>
  <init-param>
    <param-name>param2</param-name>
    <param-value>valor2</param-value>
  </init-param>
</servlet>
```

Si estamos utilizando Servlet 3.0, podemos también utilizar la anotación `@WebServlet` para configurar los parámetros de inicio:

```
@WebServlet(urlPatterns="/UrlServlet",
         initParams ={
             @InitParam(name="param1", value="valor1"),
             @InitParam(name="param2", value="valor2") })
```

Estos parámetros también pueden ser declarados por separado con la anotación `@WebInitParam`:

```
@WebInitParam(name="param1", value="valor1")
```

Para obtener luego los parámetros desde el servlet se utiliza `getServletConfig().getInitParameter(nombre)` donde `nombre` es el valor `<param-name>` del parámetro que se busca, y devuelve el valor (elemento `<param-value>` asociado), que es de tipo `String` siempre. Para obtener estos valores desde páginas JSP se emplean otros métodos. Los parámetros de inicio sólo se aplican cuando accedemos al servlet o página JSP a través del nombre asignado en `<servlet-name>`, o a través de la URL asociada en un `<servlet-mapping>`.

Cargar servlets al inicio

A veces nos puede interesar que un servlet se cargue al arrancar el servidor, y no con la primera petición de un cliente. Para hacer eso, incluimos una etiqueta `<load-on-startup>` dentro de la etiqueta `<servlet>`. Dicha etiqueta puede estar vacía:

```
<servlet>
  <servlet-name>nombre</servlet-name>
  <servlet-class>ClaseServlet</servlet-class>
  <load-on-startup/>
</servlet>
```

o contener un número:

```
<servlet>
  <servlet-name>nombre</servlet-name>
  <servlet-class>ClaseServlet</servlet-class>
  <load-on-startup>2</load-on-startup>
</servlet>
```

que indica el orden en que el servidor irá cargando los servlets (de menor a mayor valor).

Como en los casos anteriores, esto también puede ser indicado mediante la anotación `@WebServlet`:

```
@WebServlet(name="miServlet", urlPatterns="/UrlServlet",
 loadOnStartup="2")
```

Logging en aplicaciones Java EE con servlets

Utilizaremos Log4J como librería de *logging*, pero encapsulada dentro de la librería *commons-logging* de Jakarta. Para poder imprimir mensajes de log en una aplicación que contenga servlets se deben seguir estos pasos:

- Añadir los ficheros JAR de las librerías (`commons-logging-X.X.jar` y `log4j-X.X.X.jar`) en la carpeta `WEB-INF/lib` de nuestra aplicación (o añadirlas como dependencias de Maven).
- Colocar dos ficheros `.properties` en el CLASSPATH de la aplicación (carpeta `WEB-INF/classes`):

  # Un fichero `commons-logging.properties` indicando que vamos a utilizar Log4J como librería subyacente.

  # Un fichero `log4j.properties` con la configuración de logging para Log4J.

Estos ficheros los colocaremos en una carpeta fuente llamada `resources`, para que al compilarse la aplicación se vuelquen a `/WEB-INF/classes`.

- Finalmente, sólo queda en cada servlet o clase Java colocar los mensajes de log donde queramos. Veremos cómo hacerlo en servlets y páginas JSP en el siguiente módulo. Aquí vemos un ejemplo de cómo ponerlos en cada servlet:

```
...

import org.apache.commons.logging.*;

public class ServletLog4J1 extends HttpServlet {

  static Log logger = LogFactory.getLog(ServletLog4J1.class);

  // Metodo para procesar una peticion GET

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                        throws ServletException, IOException {
    logger.info("Atendiendo peticion Servlet Log4J");
    PrintWriter out = response.getWriter();
    out.println ("Servlet sencillo de prueba para logging");
    logger.debug("Fin de procesamiento de petición");
  }
}
```

Ejemplos básicos de servlets

Servlet que genera texto plano

El siguiente ejemplo de servlet muestra una página con un mensaje de saludo: "Este es un servlet de prueba". Lo cargamos mediante petición GET.

```
package ejemplos;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

@WebServlet(name="ejemplo1_1", urlPatterns="/ejemploservlet")
public class ClaseServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                 throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println ("Este es un servlet de prueba");
  }
}
```

Se obtiene un `Writer` para poder enviar datos al usuario. Simplemente se le envía la cadena que se mostrará en la página generada.

Servlet que genera una página HTML

Este otro ejemplo escribe código HTML para mostrar una página web.

```
package ejemplos;
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

@WebServlet(name="ejemplo1_2", urlPatterns="/ejemploservletHTML")
public class ClaseServletHTML extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                 throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println ("<!DOCTYPE HTML PUBLIC \""+
                 "-//W3C//DTD HTML 4.0 " +
                 "Transitional//EN\">");
    out.println ("<HTML>");
    out.println ("<BODY>");
    out.println ("<h1>Titulo</h1>");
    out.println ("<br>Servlet que genera HTML");
    out.println ("</BODY>");
    out.println ("</HTML>");
  }
}
```

Para generar una página HTML con un servlet debemos seguir dos pasos:

- Indicar que el contenido que se va a enviar es HTML (mediante el método `setContentType()` de `HttpServletResponse`):

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

Esta línea es una cabecera de respuesta, que veremos más adelante cómo utilizar. Hay que ponerla antes de obtener el `Writer`.

- Escribir en el flujo de salida el texto necesario para generar la página HTML. La línea que genera el DOCTYPE no es necesaria, aunque sí muy recomendada para que se sepa qué versión de HTML se está empleando.

Servlet que utiliza parámetros de inicialización

Este otro ejemplo utiliza dos parámetros de inicialización externos:

```
package ejemplos;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

@WebServlet(name="ejemplo1_3", urlPatterns="/ejemploservletInit")
public class ClaseServletInit extends HttpServlet {

  // Mensaje que se va a mostrar en la pagina
  String mensaje = "";
  // Numero de veces que se va a repetir el mensaje
```

```
    int contador = 1;

    // Metodo de inicializacion

    public void init(ServletConfig conf)
                                 throws ServletException {
      super.init(conf); // MUY IMPORTANTE

      mensaje = conf.getInitParameter("mensaje");
      if (mensaje == null)
        mensaje = "Hola";

      try
      {
        contador = Integer.parseInt(
           conf.getInitParameter("contador"));
      } catch (NumberFormatException e) {
        contador = 1;
      }
    }

    // Metodo para procesar una peticion GET

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                 throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println ("<!DOCTYPE HTML PUBLIC \""+
                   "-//W3C//DTD HTML 4.0 " +
                   "Transitional//EN\">");
      out.println ("<HTML>");
      out.println ("<BODY>");

      for (int i = 0; i < contador; i++)
      {
        out.println (mensaje);
        out.println ("<BR>");
      }

      out.println ("</BODY>");
      out.println ("</HTML>");
    }
  }
```

- Se utiliza el método `init()` con un parámetro `ServletConfig` para poder tomar los parámetros externos. Es importante la llamada a `super` al principio del método.
- Mediante el método `getInitParameter()` de `ServletConfig` obtenemos dos parámetros: `mensaje` y `contador`, que asignamos a las variables del mismo nombre. El primero indica el mensaje que se va a mostrar en la página, y el segundo el número de veces que se va a mostrar.
- En `doGet()` hacemos uso de esos parámetros obtenidos, para mostrar el mensaje las veces indicadas.
- Necesitaremos definir en el fichero `web.xml` los parámetros de inicio que el servlet va a leer.

JSP

JSP (**JavaServer Pages**) es una tecnología que permite incluir código Java en páginas web. El denominado *contenedor JSP* (que sería un componente del servidor web) es el encargado de tomar la página, sustituir el código Java que contiene por el resultado de su ejecución, y enviarla al cliente. Así, se pueden diseñar fácilmente páginas con partes fijas y partes variables. El siguiente es un ejemplo muy sencillo de página JSP:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Mi primera página JSP
</head>
<body>
<h1> Hoy es:
<%= new java.util.Date() %>
</h1>
</body>
</html>
```

Para ejecutar la página basta con colocarla en una aplicación web (por ejemplo, en WildFly, dentro de `webapps/ROOT`). No es necesario que sea en un directorio específico como ocurre con los servlets, sino que puede ir en cualquier directorio en el que se colocaría normalmente un HTML.

Aunque JSP y servlets parecen a primera vista tecnologías distintas, en realidad el servidor web traduce internamente el JSP a un servlet, lo compila y finalmente lo ejecuta cada vez que el cliente solicita la página JSP. Por ello, en principio, JSPs y servlets ofrecen la misma funcionalidad, aunque sus características los hacen apropiados para distinto tipo de tareas. Los JSP son mejores para generar páginas con gran parte de contenido estático. Un servlet que realice la misma función debe incluir gran cantidad de sentencias del tipo `out.println()` para producir el HTML. Por el contrario, los servlets son mejores en tareas que generen poca salida, datos binarios o páginas con gran parte de contenido variable. En proyectos más complejos, lo recomendable es combinar ambas tecnologías: los servlets para el procesamiento de información y los JSP para presentar los datos al cliente.

Traducción de los JSP a servlets

Como se ha comentado, la primera vez que se solicita una página JSP, el servidor genera el servlet equivalente, lo compila y lo ejecuta. Para las siguientes solicitudes, solo es necesario ejecutar el código compilado. El servlet generado de manera automática tiene un método `_jspService` que es el equivalente al `service` de los servlets "generados manualmente". En este método es donde se genera el código HTML, mediante instrucciones `println` y donde se ejecuta el código Java insertado en la página. Por ejemplo, la página `primera.jsp` podría generar un servlet con estructura similar al siguiente:

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse  response)
                        throws java.io.IOException, ServletException {
  JspWriter out = null;
  response.setContentType("text/html;ISO-8859-1");
  out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0
              Transitional//EN\">");
```

```
  out.println("<html>");
  out.println("<head>");
  out.println("<title>Mi primera pagina JSP");
  out.println("</head>");
  out.println("<body>");
  out.print("Hoy es ");
  out.println(new java.util.Date());
  out.println("</body>");
  out.println("</html>");
}
```

El directorio donde se coloca el servlet generado, así como su nombre, dependen del servidor de aplicaciones.

Inserción de código en páginas JSP

Hay tres formas de insertar código Java en una página JSP:

- **Expresiones** de la forma `<%= expresión %>`: en este caso, la expresión se evalúa, su resultado se convierte a `String` y se inserta en la salida.
- **Scriptlets** de la forma `<% código %>` : el código se ejecuta dentro del método `_jspService` del servlet generado.
- **Declaraciones** de la forma `<%! código %>`: se insertan en el cuerpo del servlet generado, fuera de sus métodos.

Expresiones

Como se ha visto, se evalúan, su resultado se convierte a un `String` y se escriben en la salida (el objeto predefinido `out`). La forma de traducir una expresión a código de servlet es imprimiéndola en `out` (mediante una sentencia `out.write(expresion)`) o similar.

Scriptlets

Permiten ejecutar código arbitrario, cuyo resultado no es necesario enviar a la salida. Si desde un *scriptlet* se desea escribir algo en ésta, bastará con utilizar el objeto predefinido `out`. Un uso común de los *scriptlets* es hacer que ciertas partes de código HTML aparezcan o no en función de una condición. Por ejemplo:

```
<%
  java.util.Calendar ahora = java.util.Calendar.getInstance();
  int hora = ahora.get(java.util.Calendar.HOUR_OF_DAY);
%>
<b> Hola mundo, <i>
<% if ((hora>20)||(hora<6)) { %>
     buenas noches
<% }
   else if ((hora>=6)&&(hora<=12)) { %>
          buenos días
<%      }
 else { %>
          buenas tardes
<%      } %>
</i> </b>
```

Declaraciones

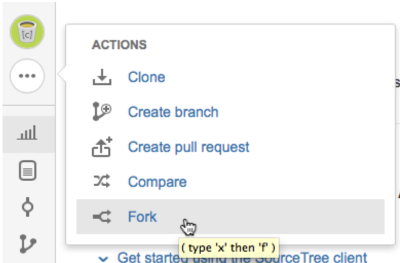
Permiten definir variables o métodos que se insertarán dentro del cuerpo del servlet generado. Esto da la posibilidad de sobreescribir los métodos `jspInit` y `jspDestroy` que son el equivalente en JSP del `init` y `destroy` de los servlets. Las variables declaradas conservarán su valor entre sucesivas llamadas a la página, ya que son variables miembro del servlet y no locales al método `jspService`. Esto nos permite, por ejemplo, crear un contador de accesos a la página:

```
<%! private int accesos = 0; %>
<h1> Visitas: <%= ++accesos %> </h1>
```

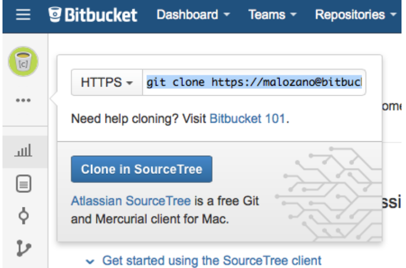
1.5. Ejercicios

Antes de empezar a crear los proyectos, debes descargarte el repositorio git `java_ua/ejercicios-cweb` en el que vas a crear los proyectos del módulo. El repositorio remoto contiene las plantillas que vamos a usar a lo largo del módulo.

En primer lugar deberemos acceder a dicho repositorio en *bitbucket* y realizar un *Fork* en nuestra cuenta personal.



De esta forma tendremos una copia propia con permisos de escritura. Una vez contamos con nuestra copia podremos realizar un *Clone* en nuestra máquina. En primer lugar consultamos en *bitbucket* el comando necesario para hacer el *Clone*:



Podemos copiar el comando y pegarlo en un terminal para clonar el repositorio en nuestra máquina:

```
$ git clone https://bitbucket.org/<alumno>/ejercicios-cweb
```

De esta forma se crea en nuestro ordenador el directorio `ejercicios-cweb` y se descarga en él una plantilla inicial de los proyectos del módulo y un fichero `.gitignore`. A partir de este momento se puede trabajar con dichos proyectos y realizar *Commit* y *Push* cuando sea oportuno:

```
$ cd ejercicios-cweb
$ git add .
$ git commit -a -m "Mensaje de commit"
$ git push origin master
```

Aplicación web en WildFly (0.5 puntos)

Vamos a probar el servidor de aplicaciones WildFly y a subir recursos estáticos a él para comprobar que funciona.

a) Comenzaremos poniendo en marcha el servidor entrando en el directorio `$WILDFLY_HOME/bin` y ejecutando el comando:

```
$ ./standalone.sh
```

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

b) Vamos a crear nuestra propia aplicación web y desplegarla en WildFly. En el directorio `cweb-comentarios` de 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 un fichero *war* y desplegarla.

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

  ```
  jar cMvf cweb-comentarios.war *
  ```

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

- **Despliegue manual en WildFly**. Para desplegar el .war manualmente basta con dejarlo en la carpeta `$WILDFLY_HOME/standalone/deployments`. Observad que transcurridos unos segundos el `.war` se desplegará de forma automática. Para probar la aplicación abre un navegador y accede a la URL:

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

Hola mundo con IntelliJ (0.5 puntos)

Vamos a crear y desplegar una aplicación *Hola mundo* mediante IntelliJ.

a) Configuramos el servidor WildFly en IntelliJ.

- **Crear una instancia del servidor WildFly:** Con la opción *File* > *Settings …* > *Application Servers* crearemos una nueva instancia de WildFly dentro de IntelliJ. En el cuadro deplegable para añadir un servidor simplemente tenemos que asegurarnos de elegir *JBoss 8.1.0 Final* > *Local*, e indicar la ruta donde está instalado (`/usr/local/wildfly`).

b) **Crearemos un proyecto Maven** de tipo *webapp-javaee7* desde IntelliJ.

- Creamos el proyecto con el arquetipo `webapp-javaee7` (deberemos añadir el arquetipo a la lista si es la primera vez que lo utilizamos).
- Como *GroupId* utilizaremos `org.expertojava.cweb`.
- Como *ArtifactId* introduciremos: `cweb-hola`. Utilizaremos este mismo nombre posteriormente como nombre del proyecto IntelliJ.
- Dejamos la versión como `1.0-SNAPSHOT`.

c) **Desplegamos el proyecto** que acabamos de crear en WildFly.

- Introducimos el elemento `finalName` en el fichero `pom.xml` para que la aplicación se despliegue en un contexto `cweb-hola`.
- Creamos un perfil de ejecución que utilice el servidor configurado en el primer paso (*Run > Edit Configurations …*).
- Añadimos al perfil de ejecución el artefacto de tipo `war` que tenemos creado por defecto (pestaña *Deployments*).
- Lanzamos el despliegue. Veremos la página de prueba que ha creado como plantilla.

d) **Añadimos** desde IntelliJ un fichero `web.xml` al proyecto.

- Entramos en *File > Project Structure … > Facets*.
- Seleccionamos el *facet Web* y añadimos un descriptor de despliegue.

e) **Crear un nuevo servlet** dentro del proyecto, en un paquete `org.expertojava.cweb.hola`, con nombre `HolaMundoServlet`. Mapearemos el servlet a la dirección `/HolaMundoServlet` utilizando anotaciones.

f) **Introducir** en el método `doGet` del servlet el código para que muestre como salida el texto `"Hola Mundo"`:

```
PrintWriter out = response.getWriter();
out.println ("Hola Mundo");
```

g) **Ejecutar la aplicación web** en WildFly y comprobar que el servlet funciona correctamente, accediendo a la dirección a la que está mapeado.

h) Vamos a hacer que el *servlet* que hemos añadido se muestre de forma automática como **página principal** de nuestra aplicación web. 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. Indicar únicamente `/HolaMundoServlet` en esta lista y comprobar que funciona adecuadamente, es decir que al llamar a `http://localhost:8080/cweb-hola` aparece dicho *servlet*.

Servlet que muestra la fecha y hora actuales (0.5 puntos)

Completar el servlet `org.expertojava.cweb.ejercicios.FechaServlet` de la aplicación `cweb-servlets` para que, tanto por GET como por POST, muestre una página HTML con la fecha y hora actuales en una cabecera `<h3>`, y en el `<title>` de la página. Para ello podéis utilizar la clase `java.util.Date`, y sacar por la salida del servlet la hora en formato cadena:

```
public void doGet(...) throws ...
{
  String fecha = "" + new java.util.Date();
  response.setContentType(...);
  out = response.getWriter();
  ... // sacar la fecha tanto en el TITLE como en una cabecera H3
}
```

Una vez hecho, configurad el descriptor de la aplicación para que el servlet se mapee a la dirección `/FechaHora`.

Servlet que muestra parámetros de inicio (0.5 puntos)

Crear un servlet `org.expertojava.cweb.ejercicios.ParamIniServlet` en la aplicación `cweb-servlets` que muestre en una tabla el nombre y el valor de todos los parámetros de inicio que se tengan configurados para ese servlet en el fichero descriptor (`web.xml`). La tabla tendrá dos columnas: una con el nombre del parámetro y otra con el valor.

Una vez hecho, probadlo añadiéndole en el fichero `web.xml` 3 parámetros de inicio con nombres `param1`, `param2` y `param3` y valores `val1`, `val2` y `val3`. Para ello deberéis dar un nombre al servlet (el nombre es arbitrario).

```
public void doGet(...) throws...
{
    Enumeration<String> nombres = this.getInitParameterNames();
    while (nombres.hasMoreElements()) {
        String nombre = nombres.nextElement();
        String valor = this.getInitParameter(nombre);
        ... // Mostrar nombre y valor en una tabla
    }
}
```

Configurar *logging* en servlets (0.5 puntos)

En la aplicación `cweb-servlets` tenemos dos servlets, `ServletLog4J1` y `ServletLog4J2` en el paquete `org.expertojava.cweb.ejercicios`. Queremos configurar las librerías de logging para poder ver los mensajes que emiten. Se pide:

- Comprobar que las librerías de logging de *commons-logging* y *log4j* están correctamente configuradas como dependencias en el `pom.xml`.
- Comprobar que los ficheros de configuración `commons-logging.properties` y `log4j.properties` están en la carpeta de fuentes llamada `resources`. Estos ficheros están configurados para que volcar los mensajes de ambos servlets (de tipo INFO o superior) a un fichero `/home/expertojava/errores.log`, con el formato:

dd/MM/aaaa hh:mm:ss - prioridad - texto del mensaje - salto de línea

- El servlet `ServletLog4J2` no saca mensajes de *log*. Añadid las líneas de código necesarias para que saque un mensaje de tipo INFO cuando empiece a procesar la petición (al inicio del `doGet`) y otro cuando la termine, anotando el tiempo transcurrido entre ambos mensajes (puede serte de utilidad el método `System.currentTimeMillis()` de Java).

Probad a llamar a los servlets `ServletLog4J1` o `ServletLog4J2` alguna vez, y que generen logs en el fichero `errores.log` que viene por defecto en el fichero de configuración.

2. Procesamiento de peticiones

Un servlet maneja peticiones de los clientes a través de su método `service` Con él se pueden manejar peticiones HTTP (entre otras), reenviando las peticiones a los métodos apropiados que las manejan. Por ejemplo, una petición GET puede redirigirse a un método `doGet`. Veremos ahora los elementos principales que intervienen en una interacción vía HTTP.

2.1. Petición y respuesta HTTP

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>
```

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.

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">
```

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 *Location*
`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

Peticiones: HttpServletRequest

Como hemos visto anteriormente, los objetos `ServletRequest` se emplean para obtener información sobre la petición de los clientes. Más en concreto, el subtipo `HttpServletRequest` se utiliza en las peticiones HTTP. Proporciona acceso a los datos de las cabeceras HTTP, cookies, parámetros pasados por el usuario, etc, sin tener que parsear nosotros a mano los datos de formulario de la petición. La clase dispone de muchos métodos, pero destacamos los siguientes:

* Para **obtener los valores de los parámetros** pasados por el cliente, se tienen los métodos:

```
Enumeration getParameterNames()
String      getParameter (String nombre)
String[]    getParameterValues (String nombre)
```

Con `getParameterNames()` se obtiene una lista con los nombres de los parámetros enviados por el cliente. Con `getParameter()` se obtiene el valor del parámetro de nombre `nombre`. Si un parámetro tiene varios valores (por ejemplo, si tenemos un array de cuadros de texto con el mismo nombre en un formulario), se pueden obtener todos separados con `getParameterValues()`. Los nombres de los parámetros normalmente sí distinguen mayúsculas de minúsculas, deberemos tener cuidado al indicarlos.

- Para **obtener la cadena de una petición GET**, se tiene el método:

```
String getQueryString()
```

que devuelve todos los parámetros de la petición en una cadena, que deberemos parsear nosotros como nos convenga.

- Para **obtener datos de peticiones POST, PUT o DELETE**, se tienen los métodos:

```
BufferedReader      getReader()
ServletInputStream getInputStream()
```

Con `getReader()` se obtiene un `BufferedReader` para peticiones donde esperemos recibir texto. Si esperamos recibir datos binarios, se debe emplear `getInputStream()`. Si lo que esperamos recibir son parámetros por POST igual que se haría por GET, es mejor utilizar los métodos `getParameterXXXX(...)` vistos antes.

- Para **obtener información sobre la línea de petición**, se tienen los métodos:

```
String getMethod()
String getRequestURI()
String getProtocol()
```

Con `getMethod()` obtenemos el comando HTTP solicitado (GET, POST, PUT, etc), con `getRequestURI()` obtenemos la parte de la URL de petición que está detrás del `host` y el puerto, pero antes de los datos del formulario. Con `getProtocol()` obtenemos el protocolo empleado (`HTTP/1.1`, `HTTP/1.0`, etc).

Respuestas: HttpServletResponse

Los objetos `ServletResponse` se emplean para enviar el resultado de procesar una petición a un cliente. El subtipo `HttpServletResponse` se utiliza en las peticiones HTTP. Proporciona acceso al canal de salida por donde enviar la respuesta al cliente.

La clase dispone de muchos métodos, pero destacamos:

```
Writer              getWriter()
ServletOutputStream getOutputStream()
```

Con `getWriter()` se obtiene un `Writer` para enviar texto al cliente. Si queremos enviar datos binarios, se debe emplear `getOutputStream()`. Si queremos especificar información de cabecera, debemos establecerla ANTES de obtener el `Writer` o el `ServletOutputStream`. Hemos visto en algún ejemplo el método `setContentType()` para indicar el tipo de contenido. Veremos las cabeceras con más detenimiento más adelante.

2.2. Procesamiento de peticiones GET y POST

Como se ha visto anteriormente, el método `doGet()` se emplea para procesar peticiones GET. Para realizar nuestro propio procesamiento de petición, simplemente sobreescribimos este método en el servlet:

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
               throws ServletException, IOException {
  // ... codigo para una peticion GET
}
```

Podemos utilizar los métodos del objeto `HttpServletRequest` vistos antes. Así podremos, entre otras cosas:

- Acceder a elementos de la petición, como valores de parámetros:

  ```
  String nombreUsuario = request.getParameter("nombre");
  ```

- Acceder a los parámetros en la cadena de la petición y procesarlos como queramos:

  ```
  String query = request.getQueryString();
  ...
  ```

- Obtener un canal de entrada (`Reader` o `InputStream`) con que leer los datos de la petición:

  ```
  BufferedReader r = request.getReader();
  ...
  ```

  Esta, sin embargo, no es una buena idea para tomar parámetros de peticiones u otras cosas. Se suele emplear sobre todo para transferencias de ficheros, pero hay que tener en cuenta que si obtenemos un canal de entrada, luego no podremos obtener parámetros u otros valores con métodos `getParameter()` y similares.

- etc.

También podemos utilizar los métodos del objeto `HttpServletResponse` para, entre otras cosas:

- Establecer valores de la cabecera (antes que cualquier otra acción sobre la respuesta):

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

- Obtener el canal de salida por el que enviar la respuesta:

  ```
  PrintWriter out = response.getWriter();
  out.println ("Enviando al cliente");
  ```

- Redirigir a otra página:

```
response.sendRedirect("http://localhost:8080/pag.html");
```

- etc.

De forma similar, el método `doPost()`, se emplea para procesar peticiones POST. Igual que antes, debemos sobreescribir este método para definir nuestro propio procesamiento de la petición:

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
               throws ServletException, IOException {
  // ... codigo para una peticion POST
}
```

Las posibilidades de los parámetros `HttpServletRequest` y `HttpServletResponse` son las mismas que para GET. Normalmente muchos servlets definen el mismo código para uno y otro método (hacen que `doPost()` llame a `doGet()` y definen allí el código, o al revés), pero conviene tenerlos separados para poder tratar independientemente uno y otro tipo de peticiones si se quiere.

Procesamiento secuencial de peticiones

Los servlets pueden gestionar múltiples peticiones de clientes concurrentemente. **Por cada petición se crea un hilo** que se ejecuta sobre el código del servlet al que se ha accedido. Si existen varias peticiones concurrentes tendremos varios hilos ejecutándose sobre un mismo objeto servlet. Esto podría ocasionar problemas de concurrencia por ejemplo si estuviésemos utilizando variables de instancia de la clase del servlet. Por este motivo deberemos evitar esta práctica cuando implementemos servlets.

Pero puede suceder que en determinado momento necesitemos acceder a cierto recurso compartido, y no nos interese que varios clientes accedan a dicho recurso simultáneamente. Para solucionar este problema, podemos definir bloques de código `synchronized`, o bien hacer que el servlet sólo atienda una petición cada vez.

Para esto último lo único que hay que hacer es que el servlet, además de heredar de `HttpServlet`, implemente la interfaz `SingleThreadModel`. Esto no supone definir más métodos, simplemente añadimos el `implements` necesario al definir la clase Servlet:

```
public class MiServlet extends HttpServlet
                       implements SingleThreadModel {
  ...
}
```

Esta es una práctica que deberemos evitar siempre que sea posible, ya que genera un cuello de botella en nuestra aplicación y no siempre evita los problemas de concurrencia (por ejemplo si accedemos a recursos compartidos por varios servlets). Por este motivo este mecanismo ha quedado *desaprobado* y en su lugar se recomienda evitar el uso de variables de instancia en servlets, y sincronizar los bloques concretos de código que pudiesen provocar problemas de concurrencia.

Lo más importante es ser consciente de que varios hilos ejecutarán de forma concurrente el código de nuestros servlets y tomar las precauciones oportunas.

Manejo de formularios

Los datos que se envían como parámetros en una petición (tras el interrogante si es una petición GET, o por otro lado si es POST) se llaman **datos de formulario.** Una vez enviados estos datos como petición, ¿cómo se extraen en el servidor?

Si trabajáramos con CGI, los datos se tomarían de forma distinta si fuese una petición GET o una POST. Para una GET, por ejemplo, tendríamos que tomar la cadena tras la interrogación, y parsearla convenientemente, separando los bloques entre `&`, y luego separando el nombre del parámetro de su valor a partir del `=`. También hay que descodificar los valores: los alfanuméricos no cambian, pero los espacios se han convertido previamente en `+`, y otros caracteres se convierten en `%XX%`.

Con servlets todo este análisis se realiza de forma automática. La clase `HttpServletRequest` dispone de métodos que devuelven la información que nos interesa ya procesada, e independientemente de si es una petición GET o POST. Hemos visto antes los métodos:

```
Enumeration getParameterNames()
String      getParameter (String nombre)
String[]    getParameterValues (String nombre)
```

Ejemplo

Veamos un ejemplo, supongamos que tenemos este formulario:

```
<html>
<body>
<form action="/appforms/servlet/ejemplos.ServletForm">
  Valor 1: <input type="text" name="texto1">
  <br>
  Valor2:
  <select name="lista">
  <option name="lista" value="Opcion 1">Opcion 1</option>
  <option name="lista" value="Opcion 2">Opcion 2</option>
  <option name="lista" value="Opcion 3">Opcion 3</option>
  </select>
  <br>
  Valores 3:
  <br>
  <input type="text" name="texto2">
  <input type="text" name="texto2">
  <input type="text" name="texto2">

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

Al validarlo se llama al servlet `ServletForm`, que muestra una página HTML con los valores introducidos en los parámetros del formulario:

```
package ejemplos;
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletForm extends HttpServlet
{
  // Metodo para GET

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                 throws ServletException, IOException {
    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

    // Mostramos los datos del formulario

    out.println ("<HTML>");
    out.println ("<BODY>");
    out.println ("<H1>Datos del formulario</H1>");
    out.println ("<BR>");

    String valor1 =
      request.getParameter("texto1");
    String valor2 =
      request.getParameter("lista");
    String[] valor3 =
      request.getParameterValues("texto2");

    out.println ("Valor 1:" + valor1);
    out.println ("<BR>");
    out.println ("Valor 2:" + valor2);
    out.println ("<BR>");
    out.println ("Valor 3:");
    out.println ("<BR>");
    if (valor3 != null)
      for (int i = 0; i < valor3.length; i++)
      {
        out.println (valor3[i]);
        out.println ("<BR>");
      }

    out.println ("</BODY>");
    out.println ("</HTML>");
  }

  // Metodo para POST

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

Para probar el ejemplo que viene en las plantillas, cargamos la URL:

http://localhost:8080/appforms/index_form.html

2.3. Cabeceras y códigos

Veremos a continuación cómo tratar las cabeceras HTTP de una petición y de una respuesta, así como los códigos de estado que emite un servidor Web ante una petición, y las variables CGI a las que podemos acceder.

Cabeceras de petición

Cuando se envía una petición HTTP, se pueden enviar, entre otras cosas, unas cabeceras con información sobre el navegador. Para leer estas cabeceras de una petición desde un servlet, se utiliza el método `getHeader()` del objeto `HttpServletRequest`.

```
String getHeader(String nombre)
```

El parámetro indica el nombre de la cabecera cuyo valor se quiere obtener. Devuelve el valor de la cabecera, o `null` si la cabecera no ha sido enviada en la petición. Se tienen otros métodos, como:

```
Enumeration getHeaderNames()
Enumeration getHeaders(String nombre)
int getIntHeader(String nombre)
...
```

Con `getHeaderNames()` obtendremos todos los nombres de las cabeceras enviadas. Con `getHeaders()` obtendremos todos los valores de la cabecera de nombre dado. También hay métodos como `getIntHeader()` que devuelve el valor de una cabecera con un tipo de dato específico (entero, en este caso). Los nombres de las cabeceras normalmente no distinguen mayúsculas de minúsculas. Algunas cabeceras son de uso común, y tienen métodos específicos para obtener sus valores, como:

```
Cookie[] getCookies()
String getContentLength()
String getContentType()
...
```

Con `getCookies()` obtendremos todas las cookies de la petición (veremos las cookies con más detalle en otro tema). Con `getContentLength()` obtenemos el valor de la cabecera `Content-Length`, y con `getContentType()` el de la cabecera `Content-Type`.

Cabeceras de respuesta

En la respuesta de un servidor web a una petición también pueden aparecer cabeceras que informan sobre el documento servido o sobre el propio servidor. Podemos definir cabeceras de respuesta para enviar cookies, indicar la fecha de modificación, etc. Estas cabeceras deben establecerse ANTES de enviar cualquier documento, o antes de obtener el `PrintWriter` si es el caso. Para enviar cabeceras, el método más general es `setHeader()` del objeto `HttpServletResponse`.

```
void setHeader(String nombre, String valor)
```

Al que se le pasan el nombre de la cabecera y el valor. Hay otros métodos útiles:

```
void setIntHeader(String nombre, int valor)
void addHeader(String nombre, String valor)
void addIntHeader(String nombre, int valor)
...
```

`setIntHeader()`o `setDateHeader()`se utilizan para enviar cabeceras de tipo entero o fecha. Los métodos `add...()`se emplean para añadir múltiples valores a una cabecera con el mismo nombre.

Algunas cabeceras tienen métodos específicos de envío, como:

```
void setContentType(String tipo)
void setContentLength(int tamaño)
void sendRedirect(String url)
void addCookie(Cookie cookie)
```

Con `setContentType()` se establece la cabecera `Content-Type` con el tipo MIME del documento. Con `setContentLength()`se indican los bytes enviados. Con `sendRedirect()`se selecciona la cabecera `Location`, y con ella se redirige a la página que le digamos. Finalmente, con `addCookie()`se establecen cookies (esto último ya lo veremos con más detalle en otro tema). Es recomendable utilizar estos métodos en lugar del método `setHeader()` para la cabecera en cuestión.

Variables CGI

Las variables CGI son una forma de recoger información sobre una petición. Algunas se derivan de la línea de petición HTTP y de las cabeceras, otras del propio socket (como el nombre o la IP de quien solicita la petición), y otras de los parámetros de instalación del servidor (como el mapeo de URLs a los paths actuales). Mostramos a continuación una tabla con las variables CGI, y cómo acceder a ellas desde servlets:

VARIABLE CGI	SIGNIFICADO	ACCESO DESDE SERVLETS
AUTH_TYPE	Tipo de cabecera Authorization (basic o digest)	`request.getAuthType()`
CONTENT_LENGTH	Número de bytes enviados en peticiones POST	`request.getContentLength()`
CONTENT_TYPE	Tipo MIME de los datos adjuntos	`request.getContentType()`
DOCUMENT_ROOT	Path del directorio raíz del servidor web	`getServletContext().getRealPath("/")`
HTTP_XXX_YYY	Acceso a cabeceras arbitrarias HTTP	`request.getHeader("Xxx-Yyy")`
PATH_INFO	Información de path adjunto a la URL	`request.getPathInfo()`
PATH_TRANSLATED	Path mapeado al path real del servidor	`request.getPathTranslated()`

VARIABLE CGI	SIGNIFICADO	ACCESO DESDE SERVLETS
QUERY_STRING	Datos adjuntos para peticiones GET	`request.getQueryString()`
REMOTE_ADDR	IP del cliente que hizo la petición	`request.getRemoteAddr()`
REMOTE_HOST	Nombre del dominio del cliente que hizo la petición (o IP si no se puede determinar)	`request.getRemoteHost()`
REMOTE_USER	Parte del usuario en la cabecera Authorization (si se suministró)	`request.getRemoteUser()`
REQUEST_METHOD	Tipo de petición (GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE)	`request. getMethod()`
SCRIPT_NAME	Path del servlet	`request.getServletPath()`
SERVER_NAME	Nombre del servidor web	`request.getServerName()`
SERVER_PORT	Puerto por el que escucha el servidor	`request.getServerPort()`
SERVER_PROTOCOL	Nombre y versión usada en la línea de petición (HTTP/1.0, HTTP/1.1 ...)	`request.getServerProtocol()`
SERVER_SOFTWARE	Información del servidor web	`getServletContext().getServerInfo()`

`request` se asume que es un objeto de tipo `HttpServletRequest`. Para obtener cualquiera de las variables antes mencionadas, sólo hay que llamar al método apropiado desde `doGet()` o `doPost()`.

Códigos de estado HTTP

Cuando un servidor web responde a una petición, en la respuesta aparece, entre otras cosas, un código de estado que indica el resultado de la petición, y un mensaje corto descriptivo de dicho código. El envío de cabeceras de respuesta normalmente se planifica junto con el envío de códigos de estado, ya que muchos de los códigos de estado necesitan tener una cabecera definida. Podemos hacer varias cosas con los servlets manipulando las líneas de estado y las cabeceras de respuesta, como por ejemplo reenviar al usuario a otros lugares, indicar que se requiere un password para acceder a un determinado sitio web, etc. Para enviar códigos de estado se emplea el método `setStatus()` de `HttpServletResponse`:

```
void setStatus(int estado)
```

Donde se le pasa como parámetro el código del estado. En la clase `HttpServletResponse` tenemos una serie de constantes para referenciar a cada código de estado. Por ejemplo, la constante:

```
HttpServletResponse.SC_NOT_FOUND
```

se corresponde con el código 404, e indica que el documento solicitado no se ha encontrado. Existen otros métodos para gestión de mensajes de error:

```
void sendError(int codigo, String mensaje)
void sendRedirect(String url)
```

`sendError()` genera una página de error, con código de error igual a `codigo`, y con mensaje de error igual a `mensaje`. Se suele utilizar este método para códigos de error, y `setStatus()` para códigos normales. `sendRedirect()` genera un error de tipo 302, envía una cabecera `Location` y redirige a la página indicada en `url`. Es mejor que enviar directamente el código, o hacer un `response.setHeader("Location", "http…")`, porque es más cómodo, y porque el servlet genera así una página con el enlace a la nueva dirección, para navegadores que no soporten redirección automática Si queremos enviar un código en la respuesta, se tiene que especificar antes de obtener el objeto `PrintWriter`.

2.4. Procesamiento asíncrono

Hemos visto anteriormente que por cada petición que se realiza a un servlet se crea un hilo de ejecución. Dado que los recursos del servidor son limitados, normalmente tenemos un número máximo de hilos que pueden atender a los clientes (tenemos un *pool* de hilos). Cuando los hilos se agoten no se podrán atender más peticiones.

Si realizamos operaciones de larga duración estos hilos quedarán ocupados durante más tiempo y será más fácil agotar los recursos disponibles. Podemos mejorar la escalabilidad del sistema utilizando procesamiento asíncrono. Este tipo de procesamiento nos permite liberar el hilo de la petición mientras se realiza una operación larga en segundo plano, de forma que otro cliente pueda utilizarlo mientras tanto. Estas operaciones podrían ser por ejemplo:

- Consultas a base de datos
- Acceso a servicios web remotos
- Operaciones que dependan del suceso de algún evento o de la interacción del usuario

Para habilitar el procesamiento asíncrono en un servlet debemos añadir el atributo `asyncSupported` a la anotación `WebServlet`:

```
@WebServlet(urlPatterns={"/ServletAsincrono"}, asyncSupported=true)
public class AsincronoServlet extends HttpServlet {
    ...
}
```

A continuación veremos como liberar el hilo de la petición del procesamiento de operaciones largas o del bloqueo de la entrada/salida utilizando el procesamiento asíncrono.

Procesamiento de operaciones de larga duración

Una vez habilitado el soporte para procesamiento asíncrono, podremos poner la petición en modo de procesamiento asíncrono mediante el método `startAsync`:

```
public void doGet(HttpServletRequest request, HttpServletResponse
 response) {
  ...
  final AsyncContext ac = request.startAsync();
  ...
}
```

Al llamar a este método obtenemos un objeto `AsyncContext` que representa el contexto de ejecución asíncrono y que será necesario para procesar la petición una vez hemos pasado a este modo. Esto nos permitirá pasar el procesamiento de la operación a otro hilo, liberando así al hilo de la petición para que pueda atender a otro cliente.

El procesamiento asíncrono de la operación se realizará como se muestra a continuación:

```
ac.start(new Runnable() { ❶
  public void run() {
      HttpServletRequest response = ac.getRequest(); ❷
      // Procesa la petición
      ...
      HttpServletResponse response = ac.getResponse(); ❸
      // Escribe la respuesta
      ...
      ac.complete(); ❹
  }
}
```

❶ Con `start` se crea un nuevo hilo dentro del cual realizaremos el procesamiento, liberando así el hilo de la petición.

❷ Podemos obtener el objeto de la petición a partir del `AsyncContext`.

❸ También podemos obtener el objeto de la respuesta a partir del `AsyncContext`, con el cual podremos devolver el resultado al cliente una vez finalizado el procesamiento.

❹ Es importante llamar a `complete` para que termine la operación asíncrona, cierre la respuesta y se la envíe el cliente. Siempre deberemos llamar a este método tras escribir la respuesta.

Entrada y salida no bloqueante

En el apartado anterior hemos visto cómo realizar una operación de larga duración de forma asíncrona. Sin embargo, a veces gran parte del tiempo que se mantiene ocupado el hilo de la petición es debido a la entrada y salida, normalmente debido a mensajes de petición extensos.

Podemos utilizar el soporte asíncrono también para leer la petición fuera del hilo de la petición, y así evitar que quede bloqueado más tiempo del necesario. Para hacer esto definiremos un `ReadListener` sobre el `InputStream` de la petición, al que le irá llegando el mensaje conforme se reciba:

```
final AsyncContext ac = request.startAsync();
final ServletInputStream sis = request.getInputStream();

sis.setReadListener(new ReadListener() { ❶
  byte buffer[] = new byte[4*1024];
```

```
    StringBuilder sb = new StringBuilder();

    @Override
    public void onDataAvailable() { ❷
        try {
          do {
              int length = sis.read(buffer);
              sb.append(new String(buffer, 0, length));
          } while(sis.isReady());
        } catch (IOException ex) {  }
    }

    @Override
    public void onAllDataRead() { ❸
        try {
          // Procesar peticion
          ...
          PrintWriter pw = ac.getResponse().getWriter();
          // Escribir respuesta
          ...
        } catch (IOException ex) {  }
        ac.complete();
    }

    @Override
    public void onError(Throwable t) {  }
});
```

❶ Definimos un objeto `ReadListener` y se lo asignamos al `InputStream` de la petición.
❷ Cada vez que se reciba un fragmento del mensaje se llamará a `onDataAvailable`. Deberemos leer todos los datos disponibles hasta el momento, y cuando no hayan llegado más saldremos del método para evitar bloquear el hilo. Este método se volverá a llamar cuando haya un nuevo fragmento disponible.
❸ Cuando todo el mensaje haya llegado se llama a `onAllDataRead`. Aquí podemos ya ver todo el contenido que hemos recibido con la petición, y podemos procesarlo y devolver una respuesta.

2.5. Ejemplos

Ejemplo de cabeceras de petición

El siguiente servlet muestra los valores de todas las cabeceras HTTP enviadas en la petición. Recorre las cabeceras enviadas y muestra su nombre y valor:

```
package ejemplos;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletCabecerasPeticion extends HttpServlet {

    // Metodo para GET
```

```
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                    throws ServletException, IOException {
        PrintWriter out = response.getWriter();

        // Mostramos las cabeceras enviadas
        // en la peticion

        out.println ("<HTML>");
        out.println ("<BODY>");
        out.println ("<H1>Cabeceras</H1>");
        out.println ("<BR>");

        Enumeration cabeceras = request.getHeaderNames();

        while (cabeceras.hasMoreElements())
        {
            String nombre = (String)(cabeceras.nextElement());
            out.println ("Nombre: " + nombre +
               ", Valor: " + request.getHeader(nombre));
            out.println ("<BR><BR>");
        }

        out.println ("</BODY>");
        out.println ("</HTML>");
    }

    // Metodo para POST
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                    throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Se puede probar con este formulario, pinchando el botón:

```
<html>
<body>
<form action=
 "/appcab/servlet/ejemplos.ServletCabecerasPeticion">
  <input type="submit" value="Pulsa aqui">
</form>
</body>
</html>
```

Ejemplo de cabeceras de respuesta

El siguiente servlet espera un parámetro `accion` que puede tomar 4 valores:

- **primos**: El servlet tiene un hilo que está constantemente calculando números primos. Al elegir esta opción se envía una cabecera `Refresh` y recarga el servlet cada 10 segundos, mostrando el último número primo que ha encontrado.
- **redirect**: Utiliza un `sendRedirect()` para cargar la página que se indique como parámetro

- **error**: Utiliza un `sendError()` para mostrar una página de error, con un mensaje de error definido por el usuario, y un código de error a elegir de una lista.
- **codigo**: Envía un código de estado HTTP (con `setStatus()`), a elegir de entre una lista.

```
package ejemplos;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletCabecerasRespuesta extends HttpServlet
                                       implements Runnable {
    // Ultimo numero primo descubierto
    long primo = 1;

    // Hilo para calcular numeros primos
    Thread t = new Thread(this);

    // Metodo de inicializacion
    public void init()
    {
        t.start();
    }

    // Metodo para GET
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                    throws ServletException, IOException {
        String accion = request.getParameter("accion");

        if (accion.equals("primo"))
        {
            // Buscar el ultimo numero
            // primo y enviarlo

            response.setContentType("text/html");
            response.setHeader("Refresh", "10");
            PrintWriter out = response.getWriter();
            out.println ("<HTML><BODY>");
            out.println ("Primo: " + primo);
            out.println ("</BODY></HTML>");

        } else if (accion.equals("redirect")) {

            // Redirigir a otra pagina

            String url = request.getParameter("url");
            if (url == null)
                url = "http://www.ua.es";
            response.sendRedirect(url);

        } else if (accion.equals("error")) {

            // Enviar error con sendError()

            int codigo = response.SC_NOT_FOUND;
```

```
            try {
                codigo = Integer.parseInt
               (request.getParameter("codigoMensaje"));
            } catch (Exception ex) {
                codigo = response.SC_NOT_FOUND;
            }

            String mensaje = request.getParameter("mensaje");
            if (mensaje == null)
                mensaje = "Error generado";
            response.sendError(codigo, mensaje);

        } else if (accion.equals("codigo")) {

            // Enviar un codigo de error

            int codigo = response.SC_NOT_FOUND;
            try {
                codigo = Integer.parseInt
                  (request.getParameter("codigo"));
            } catch (Exception ex) {
                codigo = response.SC_NOT_FOUND;
            }

            response.setStatus(codigo);
        }
    }

    // Metodo para POST
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                    throws ServletException, IOException {
        doGet(request, response);
    }

    ... el resto del codigo es para el hilo,
    para calcular numeros primos
    Puede consultarse en el fichero fuente,
    aqui se quita por simplificar
}
```

Se puede probar con este formulario, eligiendo la acción a realizar, introduciendo los parámetros necesarios en el formulario y pinchando el botón de Enviar Datos:

```
<html>
<body>
<form action=
"/appcab/servlet/ejemplos.ServletCabecerasRespuesta">

<table border="0">

<tr>
<td>
<input type="radio" name="accion" value="primo" selected>
Obtener ultimo numero primo
</td>
```

```
<td></td>
<td></td>
</tr>

<tr>
<td>
<input type="radio" name="accion" value="redirect">
Redirigir a una pagina
</td>
<td>
URL:
<input type="text" name="url" value="http://www.ua.es">
</td>
<td></td>
</tr>

<tr>
<td>
<input type="radio" name="accion" value="error">
Mostrar pagina de error
</td>
<td>
Mensaje:
<input type="text" name="mensaje"
value="Error generado por el usuario">
</td>
<td>
Codigo:
<select name="codigoMensaje">
<option name="codigoMensaje" value="400">400</option>
<option name="codigoMensaje" value="401">401</option>
<option name="codigoMensaje" value="403">403</option>
<option name="codigoMensaje" value="404" selected>404
</option>
</select>
</td>
</tr>

<tr>
<td>
<input type="radio" name="accion" value="codigo">
Enviar codigo de error
</td>
<td>
Codigo:
<select name="codigo">
<option name="codigo" value="200">200</option>
<option name="codigo" value="204">204</option>
<option name="codigo" value="404" selected>404</option>
</select>
</td>
<td></td>
</tr>

</table>

<input type="submit" value="Enviar Datos">
```

```
</form>
</body>
</html>
```

Ejemplo de autentificación

El siguiente servlet emplea las cabeceras de autentificación: envía una cabecera de autentificación si no ha recibido ninguna, o si la que ha recibido no está dentro de un conjunto de `Properties` predefinido, con logins y passwords válidos. En el caso de introducir un login o password válidos, muestra un mensaje de bienvenida. Los logins y passwords están en un objeto `Properties`, definido en el método `init()`. Podríamos leer estos datos de un fichero, aunque por simplicidad aquí se definen como constantes de cadena. Los datos de autentificación se envían codificados, y se emplea un objeto `sun.misc.BASE64Decoder` para descodificarlos y sacar el login y password.

```
package ejemplos;

import java.io.*;
import java.util.*;
import sun.misc.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletPassword extends HttpServlet
{
    // Conjunto de logins y passwords permitidos
    Properties datos = new Properties();

    // Metodo de inicializacion

    public void init()
    {
        datos.setProperty("usuario1", "password1");
        datos.setProperty("usuario2", "password2");
    }

    // Metodo para GET

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                    throws ServletException, IOException {
        response.setContentType("text/html");

        // Comprobamos si hay cabecera
        // de autorizacion

        String autorizacion = request.getHeader("Authorization");

        if (autorizacion == null) {
            // Enviamos el codigo 401 y
            // la cabecera para autentificacion

            response.setStatus(response.SC_UNAUTHORIZED);
            response.setHeader("WWW-Authenticate",
                               "BASIC realm=\"privileged-few\"");
```

```
        } else {
            // Obtenemos los datos del usuario
            // y comparamos con los almacenados

            // Quitamos los 6 primeros caracteres
            // que indican tipo de autentificación
            // (BASIC)

            String datosUsuario =
                autorizacion.substring(6).trim();

            BASE64Decoder dec = new BASE64Decoder();

            String usuarioPassword = new String
                (dec.decodeBuffer(datosUsuario));

            int indice = usuarioPassword.indexOf(":");

            String usuario =
                usuarioPassword.substring(0, indice);

            String password =
                usuarioPassword.substring(indice + 1);

            String passwordReal =
                datos.getProperty(usuario);

            if (passwordReal != null &&
                passwordReal.equals(password)) {

                // Mensaje de bienvenida

                PrintWriter out = response.getWriter();
                out.println ("<HTML><BODY>");
                out.println ("OK");
                out.println ("</BODY></HTML>");
            } else {

                // Pedir autentificacion

                response.setStatus(response.SC_UNAUTHORIZED);
                response.setHeader("WWW-Authenticate",
                           "BASIC realm=\"privileged-few\"");
            }
        }
    }

    // Metodo para POST

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

Se puede probar cada ejemplo, respectivamente, con:

http://localhost:8080/appcab/inicioCabecerasPeticion.html

http://localhost:8080/appcab/inicioCabecerasRespuesta.html

http://localhost:8080/appcab/servlet/ejemplos.ServletPassword

Un ejemplo de login y password válidos para el tercer ejemplo es: login=`usuario1`, password=`password1`.

2.6. Ejercicios

Recogida de parámetros del usuario (0,3 puntos)

La aplicación `cweb-peticiones` contiene un formulario `form_datos.html` con una serie de campos (tipo texto, listas, checkboxes…). Se pide que dicho formulario, en su *action*, llame al servlet `org.expertojava.cweb.ejercicios.DatosServlet` que deberéis crear e implementar. El servlet recogerá todos los parámetros del formulario y los mostrará en una tabla de dos columnas (una con el nombre del parámetro y otra con el valor).

Trabajando con redirecciones (0,4 puntos)

Sobre la aplicación anterior, tenemos otro formulario `form_datos2.html` idéntico al del ejercicio anterior, que deberá llamar al servlet `org.expertojava.cweb.ejercicios.DatosServlet2`. Crear este servlet y hacer que redirija a la página `bienvenida.html` con un mensaje de bienvenida, si los datos introducidos en el formulario son correctos, y a la misma página `form_datos2.html` si hay algún dato incorrecto. Entenderemos por dato incorrecto que alguno de los campos de texto se quede vacío. Utilizad el método `sendRedirect` de la respuesta para las redirecciones.

Un buscador sencillo (0,4 puntos)

En el fichero `libros.txt` hay un listado de libros, indicando su ISBN, título y autor. Cread e implementad un servlet `org.expertojava.cweb.ejercicios.LibroServlet` que lea dicho fichero, guarde los libros en un `ArrayList`, y reciba un parámetro `cadena`. Como resultado, sacará todos los libros de la lista que contengan dicha cadena (en el título, en el autor, o en cualquier parte de la cadena). Podéis hacer también una página HTML `libros.html` con el formulario de búsqueda que llame al servlet, para poderlo probar.

Para leer los libros, en la inicialización del servlet podemos utilizar un código como el siguiente:

```
ArrayList<String> libros = new ArrayList<String>();
...
InputStream is = getClass().getResourceAsStream("/libros.txt");
BufferedReader br = new BufferedReader(new InputStreamReader(is));
String libro;
try {
  while( (libro=br.readLine())!=null ) {
    libros.add(libro);
  }
} catch (IOException e) { }
```

Distinguir el navegador (0,3 puntos)

Muchas veces, cuando escribamos una aplicación Web, nos va a interesar poder distinguir el tipo de navegador que está utilizando el cliente, porque en función del mismo se podrán hacer o no determinadas acciones. Por ejemplo, el código Javascript que se emplea en un navegador Internet Explorer es diferente a veces del que se emplea en uno Mozilla. Para probar a distinguir entre navegadores, vamos a crear el servlet `org.expertojava.cweb.ejercicios.NavServlet` para que identifique si el cliente accede desde un tipo de navegador u otro. Para ello leemos la cabecera `User-Agent` con el método `getHeader(…)` de la petición, y comprobamos su valor. Mostrad la cadena en una

página, y cargad dicha página desde dos o tres navegadores diferentes. Cada uno mostrará algún rasgo distintivo en dicha cadena, que lo identifique de los demás. Una vez tengáis una parte de texto que los diferencia (por ejemplo, en Firefox el `User-Agent` tiene la cadena "Firefox", y en el otro navegador (Mozilla, por ejemplo), no la tiene) haríamos con algo como:

```
public void doGet(HttpServletRequest req, ...) throws ...
{
  String nav = req.getHeader("User-Agent");
  // Cambiar "Firefox" por el texto que sea
  if (nav.indexOf("Firefox") != -1)
    ... // Firefox
  else
    ... // Otro navegador (Mozilla)
  ...
}
```

Una vez distinguido el navegador, ya se podría hacer algo que sólo sirviera para ese navegador. En este caso, por simplificar, vamos a cargar como imagen el logo del navegador que hayamos detectado. Tenéis en la plantilla las imágenes correspondientes diferentes navegadores (directorio `webapp/logos`). Colocad como imagen de la página la del navegador que hayáis detectado (con una etiqueta `<img>` de HTML).

Redirecciones con retardo (0,3 puntos)

El formulario `form_datos3.html` se envía al servlet `org.expertojava.cweb.ejercicios.CompruebaServlet`. Se pide crear e implementar dicho servlet para comprobar que los datos sean correctos (que no haya ningún campo de texto vacío). En el caso de que no haya errores el servlet simplemente mostrará un mensaje indicando que todo ha ido bien. Si hay algún error, el servlet debe redirigir al servlet `org.expertojava.cweb.ejercicios.ErrorCompruebaServlet`, que deberéis crear para que muestre un mensaje con el error producido (indicando qué campo está incompleto), y a los 5 segundos redirija al formulario anterior

Se puede utilizar la cabecera de respuesta `Refresh` con valor `5; url=form_datos3.html`

Loggear variables CGI (0,3 puntos)

Sobre el servlet `org.expertojava.cweb.ejercicios.CompruebaServlet` anterior vamos a añadir mensajes de log de tipo INFO, para que:

- Tras cada petición (por `doGet` o `doPost`), genere un mensaje de tipo INFO que indique la IP del cliente que solicitó la petición (variable CGI `REMOTE_ADDR`), el tipo de petición (variable CGI REQUEST_METHOD), y el tipo de navegador (cabecera `User-Agent`).
- El mensaje en conjunto deberá tener el formato siguiente (se incluyen en las plantillas los ficheros de configuración para obtener este formato):

  `día/mes/año hora:minuto:segundo - texto del mensaje - nueva línea`

3. Manejo de Cookies y Sesiones

Veremos en este tema aspectos sobre el seguimiento de las acciones de los usuarios sobre un sitio web. Para ello veremos cómo trabajar con cookies en los servlets, y cómo manejar información sobre las sesiones de los usuarios.

3.1. Cookies

Cookies en HTTP

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 muestra 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 (los servlets que se ejecutan en un mismo servidor comparten las cookies). El tamaño máximo de una cookie es de 4096 bytes.

En Javascript, por ejemplo, el objeto `document.cookie` contiene como valor una lista de la forma:

`nombre1=valor1;nombre2=valor2;…;nombreN=valorN`

donde se almacenan así los valores de las cookies que se tengan definidas.

Se pueden emplear `cookies`, entre otras cosas, para:

- **Identificar a un usuario durante una o varias sesiones.** Por ejemplo, a la hora de realizar compras a través de una tienda web, se almacena su identidad (login y password) como una cookie y se recuerda a lo largo de diferentes visitas qué es lo que lleva almacenado en su cesta de la compra cada usuario.
- **Personalizar un sitio web de acuerdo a las preferencias de cada usuario**: definir el contenido, apariencia, etc, que queremos que tenga una determinada página en función de las preferencias del usuario que la esté visitando.

A la hora de trabajar con cookies, debemos tener en cuenta que nuestro sitio web no debe depender de ellas, puesto que muchos navegadores y usuarios las deshabilitan para evitar problemas de privacidad y otras cuestiones.

Veremos ahora cómo trabajar con cookies desde servlets.

Enviar una cookie

Para crear una nueva cookie y enviarla, se siguen los pasos:

1. Crear la cookie

Las cookies se manejan con la clase `Cookie`. Se tiene el constructor:

```
public Cookie (String nombre, String valor)
```

que crea una cookie de nombre `nombre`, dándole el valor `valor`.

2. Establecer los atributos de la cookie

Una vez creada la cookie, podemos establecer los atributos que queramos, con los métodos de la clase `Cookie`. Por ejemplo, se tienen:

```
public void setComment(String comentario)
public void setMaxAge(int edad)
```

```
...
```

El primero asigna una cadena descriptiva sobre la cookie. El segundo indica cuántos segundos de vida tiene. Si es un valor negativo, se borrará la cookie cuando se cierre el navegador. Si el valor es 0, se borra la cookie instantáneamente, y si es positivo, se borrará la cookie cuando pasen los segundos indicados (si cerramos y volvemos a abrir el navegador dentro de ese tiempo, la cookie todavía persistirá). Se tienen otros métodos para establecer atributos de la cookie.

3. Enviar la cookie

Las cookies se añaden a la cabecera de la respuesta, y se envían así al cliente, mediante el método de `HttpServletResponse`:

```
public void addCookie (Cookie cookie)
```

Ejemplo

Vemos un ejemplo completo de envío de cookie:

```
public class MiServlet extends HttpServlet
{
 public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
                throws ServletException, IOException {
  Cookie miCookie = new Cookie ("nombre", "Pepe");
  miCookie.setMaxAge(120);
  response.addCookie(miCookie);
  PrintWriter out = response.getWriter();
  ...
 }
}
```

Hay que tener en cuenta que las cookies son parte de la cabecera HTTP, con lo cual hay que enviarlas ANTES de escribir la respuesta (o antes de obtener el objeto `Writer` si lo queremos utilizar).

Obtener una cookie

Para obtener una cookie que envía el cliente se trabaja sobre la petición del cliente (`HttpServletRequest`), siguiendo los pasos:

1. Obtener todas las cookies

Obtenemos todas las cookies con el método **getCookies()** de la clase `HttpServletRequest`:

```
public Cookie[] getCookies()
```

Con esto se tiene un array con todas las cookies actuales para el usuario. Si no hay cookies el método devuelve `null`.

2. Obtener el valor de una cookie

Con lo anterior, para obtener el valor de una cookie simplemente recorremos el array de cookies buscando la que concuerde con el nombre que queramos. Pueden ser útiles los métodos de `Cookie`:

```
public String getName()
public String getValue()
```

El primero obtiene el nombre de la cookie, y el segundo el valor.

Ejemplo

Un ejemplo de uso, para obtener el nombre del usuario, guardado en la cookie `"nombre"`:

```
public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
                throws ServletException, IOException {
 Cookie[] cookies = request.getCookies();
 String nombre;
 for (int i = 0; i < cookies.length; i++)
  if (cookies[i].getName().equals("nombre"))
   nombre = cookies[i].getValue();
}
```

Ejemplo

Aquí tenéis un ejemplo de uso de cookies. El servlet `ServletCookies` cuenta el número de visitas a una página con una cookie que dura 3 minutos.

```
package ejemplos;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletCookies extends HttpServlet
{
   // Metodo para GET

   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                  throws ServletException, IOException {
 response.setContentType("text/html");
 response.setHeader("Cache-Control", "no-cache");

 Cookie[] cookies = request.getCookies();
 Cookie contador = buscaCookie("contador", cookies);

 if (contador == null)
 {
    // Creamos la cookie con el contador

    Cookie cookie = new Cookie ("contador", "1");
    cookie.setMaxAge(180);
    response.addCookie(cookie);
```

```
      // Mostramos el mensaje de primera visita

      PrintWriter out = response.getWriter();
      out.println ("<HTML>");
      out.println ("<BODY>");
      out.println ("Primera visita");
      out.println ("<BR>");
      out.println ("</BODY>");
      out.println ("</HTML>");

   } else {

      // Obtenemos el valor actual del contador

      int cont = Integer.parseInt(contador.getValue());
      cont++;

      // Modificamos el valor de la cookie
      // incrementando el contador

      Cookie cookie = new Cookie ("contador", "" + cont);
      cookie.setMaxAge(180);
      response.addCookie(cookie);

      // Mostramos el numero de visitas

      PrintWriter out = response.getWriter();
      out.println ("<HTML>");
      out.println ("<BODY>");
      out.println ("Visita numero " + cont);
      out.println ("<BR>");
      out.println ("</BODY>");
      out.println ("</HTML>");
   }
     }

     // Busca la cookie 'nombre'
     // en el array de cookies indicado.
     // Devuelve null si no esta

     private Cookie buscaCookie(String nombre,
                                Cookie[] cookies)
     {
   if (cookies == null)
      return null;

   for (int i = 0; i < cookies.length; i++)
      if (cookies[i].getName().equals(nombre))
    return cookies[i];

   return null;
     }
  }
```

3.2. Seguimiento de sesiones

El seguimiento de sesiones es un mecanismo empleado por los servlets para gestionar un estado sobre las peticiones realizadas desde un mismo cliente (un mismo navegador) a lo largo de un período de tiempo determinado. Las sesiones se comparten por los servlets a los que accede un cliente (algo útil si queremos construir una aplicación basada en múltiples servlets).

Para utilizar el seguimiento de sesiones se tienen los pasos:

- Utilizar una sesión (objeto `HttpSession`) para un usuario
- Almacenar u obtener datos del objeto `HttpSession`
- Opcionalmente, invalidar la sesión

Obtener una sesión

El método `getSession()` del objeto `HttpServletRequest` obtiene una sesión de usuario.

```
public HttpSession getSession()
public HttpSession getSession(boolean crear)
```

El primer método obtiene la sesión actual, o crea una si no existe. Con el segundo método podemos establecer, mediante el flag booleano, si queremos crear una nueva si no existe (`true`) o no (`false`). Si la sesión es nueva, el método `isNew()` del `HttpSession` devuelve `true`, y la sesión no tiene ningún dato asociado. Deberemos ir añadiéndole datos tras crearla.

Para mantener la sesión de forma adecuada, debemos llamar a `getSession()` antes de que se escriba nada en la respuesta `HttpServletResponse` (y si utilizamos un `Writer`, debemos obtenerla antes de obtener el `Writer`, no antes de escribir datos).

Por ejemplo:

```
public class MiServlet extends HttpServlet
{
 public void doGet (HttpServletRequest request,
                    HttpServletResponse response)
                 throws ServletException, IOException {
  HttpSession sesion = request.getSession(true);
  ...
  PrintWriter out = response.getWriter();
  ...
 }
}
```

Guardar y obtener datos de la sesión

La interfaz `HttpSession` proporciona métodos que permiten almacenar y obtener:

- Propiedades de la sesión, como por ejemplo su identificador:

```
public String getId()
```

- Datos de la aplicación, que se almacenan y obtienen como pares nombre-valor, donde el nombre es un `String` que identifica al dato, y el valor es un `Object` con el valor asociado. Tendremos que tener cuidado de no sobreescribir datos de un servlet desde otro accidentalmente. Se tienen los métodos:

```
public Object getAttribute(String nombre)
public void   setAttribute(String nombre, Object valor)
public void   removeAttribute(String nombre)
```

  que obtienen / establecen / eliminan, respectivamente, valores de atributos. Estos métodos eran `getValue()`, `putValue()` y `removeValue()` hasta la versión 2.2 de servlets. Se tienen además métodos como `getAttributeNames()` para obtener los nombres de los atributos que se tienen almacenados para la sesión, y otros métodos de utilidad en la clase `HttpSession`.

Por ejemplo:

```
public class MiServlet extends HttpServlet
{
 public void doGet (HttpServletRequest request,
                    HttpServletResponse response)
                 throws ServletException, IOException {
  HttpSession sesion = request.getSession(true);
  String nombreUsuario =
     (String)(sesion.getAttribute("nombre"));
  sesion.setAttribute("password", "secreto");
 }
}
```

El ejemplo lee el atributo "nombre" de la sesión, y establece el atributo "password" al valor "secreto".

Invalidar la sesión

Una sesión de usuario puede invalidarse manualmente, o automáticamente (dependiendo de dónde esté ejecutando el servlet). Esto implica eliminar el objeto `HttpSession` y sus valores de la memoria. Se tienen los métodos de `HttpSession`:

```
public int getMaxInactiveInterval()
public void setMaxInactiveInterval(int intervalo)
public void invalidate()
```

Para invalidarla automáticamente, la sesión expira cuando transcurre el tiempo indicado por el método `getMaxInactiveInterval()`entre dos accesos del cliente (en segundos). Se puede establecer dicho valor con `setMaxInactiveInterval(…)`.

Para invalidar manualmente una sesión, se emplea el método `invalidate()`de la misma. Esto puede ser interesante por ejemplo en comercio electrónico: podemos mantener una

sesión donde se vayan acumulando los productos que un usuario quiera comprar, e invalidar la sesión (borrarla) cuando el usuario compre los productos.

Por ejemplo:

```
public class MiServlet extends HttpServlet
{
 public void doGet (HttpServletRequest request,
                    HttpServletResponse response)
                 throws ServletException, IOException {
  HttpSession sesion = request.getSession(true);
  ...
  sesion.invalidate();
  ...
 }
}
```

Compatibilidad con los navegadores

Una alternativa para el seguimiento de sesiones consiste en la **reescritura de URLs**. Con esta técnica, se añaden como parámetros de la URL los datos relativos a la sesión actual, de forma que se van conservando entre las páginas.

El seguimiento de sesiones por defecto emplea cookies para almacenar el identificador de una sesión. Para poder utilizar seguimiento de sesiones con usuarios que accedan desde navegadores que no utilicen cookies, los servlets aplican automáticamente la reescritura de URLs cuando no se utilizan cookies.

Para poder utilizar esto, debemos codificar todas las URLs que referenciemos. Para esto se emplean los métodos:

```
public String encodeURL(String url)
public String encodeRedirectURL(String url)
```

El primero se emplea para asociar identificadores con URLs, se utilizará cuando pongamos urls en el contenido de la página que generamos. El segundo se utiliza para asociar identificadores con redirecciones. Lo emplearemos cuando utilicemos métodos `sendRedirect()`, para codificar la URL que se le pasa. Ambos devuelven la URL sobreescrita si la sobreescritura ha sido necesaria, o la misma URL si no ha sido necesario sobreescribir.

Por ejemplo:

```
public class MiServlet extends HttpServlet
{
 public void doGet (HttpServletRequest request,
                    HttpServletResponse response)
                 throws ServletException, IOException {
  ...
  String url = response.encodeURL(
     "http://localhost:8080/mipagina.html");
  out.println ("<a href=\"" + url + "\">...</a>");
```

```
    ...
    String url2 = response.encodeRedirectURL(
       "http://otrapagina.com");
    response.sendRedirect(url2);
  }
}
```

Oyentes

Existen tres tipos de oyentes (`listeners`) que podemos utilizar en sesiones, para dar respuesta a eventos que produzcan las propias sesiones, o que desde fuera se provoquen sobre las mismas:

- `HttpSessionListener` se emplea para eventos de crear la sesión y terminarla. Tiene los métodos:

  ```
  public void sessionCreated(HttpSessionEvent e)
  public void sessionDestroyed(HttpSessionEvent e)
  ```

  El objeto que implemente esta interfaz ejecutará el código de `sessionCreated()` cuando se inicie la sesión, y el de `sessionDestroyed()` cuando se termine. Las clases que implementen este oyente deben llevar la anotación `@WebListener` en caso de estar utilizando la API de Servlet 3.0, o bien configurarse en el fichero descriptor de la aplicación, mediante etiquetas **<listener>**:

  ```
  <listener>
   <listener-class>clase</listener-class>
  </listener>
  ```

  donde en **<listener-class>** se pone el nombre (completo) de la clase que implementa el listener. Así, el listener se registra, y el servidor ya sabe que tiene que notificarle en los momentos oportunos.

- `HttpSessionBindingListener`: si un objeto asociado a una sesión implementa esta interfaz, la sesión se encarga de notificarle de cuándo son añadidos y eliminados de la sesión. Para ello la interfaz tiene los métodos:

  ```
  public void valueBound(HttpSessionBindingEvent e)
  public void valueUnbound(HttpSessionBindingEvent e)
  ```

  El objeto que implemente esta interfaz ejecutará el código de `valueBound()` cuando la sesión lo añada, y el método `valueUnbound()` cuando la sesión lo elimine.

- `HttpSessionActivationListener`: si un objeto asociado a una sesión implementa esta interfaz, la sesión se encarga de notificarles de cuándo el contenedor cambia la sesión entre máquinas virtuales distintas, para un sistema distribuido. Para ello tiene los métodos:

  ```
  public void sessionDidActivate(HttpSessionEvent e)
  public void sessionWillPassivate(HttpSessionEvent e)
  ```

El objeto que implemente esta interfaz ejecutará el código de `sessionDidActivate()` cuando la sesión se active, y `sessionWillPassivate()` cuando se vuelva pasiva.

Ejemplos

Aquí tenéis un ejemplo de uso de sesiones. El servlet `ServletSesiones` cuenta el número de visitas a una página en una sola sesión (en una sola ventana de navegador).

```
package ejemplos;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletSesiones extends HttpServlet {

    // Metodo para GET

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                throws ServletException, IOException {
        response.setContentType("text/html");
        response.setHeader("Cache-Control", "no-cache");

        HttpSession sesion = request.getSession();
        PrintWriter out = response.getWriter();

        if (sesion.isNew()) {
            // Mostramos un mensaje de primera visita

            out.println ("<HTML>");
            out.println ("<BODY>");
            out.println ("Primera visita a la pagina");
            out.println ("<BR>");
            out.println ("</BODY>");
            out.println ("</HTML>");

            sesion.setAttribute("contador", new Integer(1));

        } else {

            // Mostramos el numero de visitas
            // y actualizamos el contador

            int contador = ((Integer)
                (sesion.getAttribute("contador"))).intValue();
            contador++;

            out.println ("<HTML>");
            out.println ("<BODY>");
            out.println ("Visita numero " +
                         contador +
                         " a la pagina en esta sesion");
            out.println ("<BR>");
            out.println ("</BODY>");
```

```
            out.println ("</HTML>");

            sesion.setAttribute("contador",
                                new Integer(contador));
  }
    }
 }
```

El siguiente servlet utiliza como atributo de sesión una clase interna `ObjetoSesion`, que implementa la interfaz `HttpSessionBindingListener`. Dicho objeto tiene dentro un valor entero, y una cadena. Cada vez que el objeto se añade a la sesión se modifica un mensaje de texto, mostrando el valor entero actual del objeto:

```
package ejemplos;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class EjemploServletListener extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                   throws ServletException, IOException {
        response.setContentType("text/html");
        response.setHeader("Cache-Control",
                           "no-cache");

        HttpSession sesion = request.getSession();
        PrintWriter out = response.getWriter();

        if (sesion.isNew()) {
            // Mostramos mensaje de inicio

            out.println ("<HTML><BODY>" +
                         "Mensaje de inicio" +
                         "</BODY></HTML>");
            sesion.setAttribute("contador",
                                new ObjetoSesion(1));

        } else {

            // Mostramos el valor actual del
            // objeto "contador"

            int contador = ((ObjetoSesion)
                (sesion.getAttribute("contador"))).getValor();
            String mensaje = ((ObjetoSesion)
                (sesion.getAttribute("contador"))).getEnlazado();

            out.println ("<HTML><BODY>");
            out.println ("Valor: " + contador +
                         "<br>Enlazado: " + mensaje);
            out.println ("</BODY></HTML>");

            sesion.setAttribute("contador",
                                new ObjetoSesion(contador+1));
```

```
        }
    }
}

class ObjetoSesion implements HttpSessionBindingListener
{
    int valor;
    String enlazado = "NO";

    public ObjetoSesion(int valor) {
        this.valor = valor;
    }

    public void valueBound(HttpSessionBindingEvent e) {
        enlazado = "Objeto enlazado a la sesion " +
                   valor + " veces";
    }

    public void valueUnbound(HttpSessionBindingEvent e) {
    }

    public String getEnlazado() {
        return enlazado;
    }

    public int getValor() {
        return valor;
    }
}
```

Si cargamos el servlet por primera vez en la sesión, muestra el mensaje:

```
Mensaje de bienvenida
```

Luego, cada vez que recarguemos el servlet se entrará en el bloque `else`, y al llamar al método `setAttribute()` se disparará el método `valueBound()`, actualizando la cadena. Con esto, con cada recarga se mostrará el mensaje:

```
Valor: i
Enlazado: Objeto enlazado a la sesion i veces
```

siendo `i` el número de veces que se ha enlazado (que coincide con el número de veces que se ha cargado el servlet, en este caso). Lo esencial aquí es que esta variable `enlazado` se modifica sólo desde el método `valueBound`, y éste es llamado sólo cuando el objeto se añade a la sesión.

3.3. Ejercicios

Personalizar un sitio web (0.3 puntos)

Una de las utilidades que se le dan a las cookies es la de personalizar sitios Web. La aplicación `cweb-sesiones` contiene un formulario `form_pers.html` que le indica al usuario que introduzca su nombre, y elija un color de fondo. Dicho formulario llama después al servlet `org.expertojava.cweb.ejercicios.PersonalizaServlet`. El formulario también tiene un enlace *Ir a página principal*, que internamente llama al servlet `org.expertojava.cweb.ejercicios.PrincipalServlet`. Se pide:

- Que en el servlet `org.expertojava.cweb.ejercicios.PersonalizaServlet` se guarde en dos cookies el nombre del usuario y el valor del color seleccionado. Después, redirigirá a `form_pers.html` de nuevo.
- Que el servlet `org.expertojava.cweb.ejercicios.PrincipalServlet` (llamado desde el formulario anterior) tome las cookies que le envíe el usuario, y genere una página de bienvenida con el color de fondo que haya en la cookie con el color, y con un mensaje de saludo al nombre que haya en la cookie con el nombre. Para establecer el color de fondo, en el *body* podemos tener un atributo *bgcolor*, y ahí le colocaremos el valor que tenga la cookie.

```
<body bgcolor="red">
...
```

Carro de la compra (0.5 puntos)

La aplicación `cweb-sesiones` contiene también una página `form_carro.html` que tiene una lista de artículos para comprar. Para comprarlos, basta con pulsar el botón de "Añadir al carro" correspondiente. Dicho formulario llama al servlet `org.expertojava.cweb.ejercicios.CarroServlet`. Se pide que dicho servlet almacene como objetos de sesión los objetos que se vayan comprando, y genere una página dinámica con:

- Cada uno de los objetos que se llevan comprados hasta ahora en la sesión.
- Precio total de la compra
- Un enlace al formulario `form_carro.html` para seguir comprando.

Para almacenar los objetos podemos utilizar cualquier estructura de datos que queramos (`ArrayList`, etc), y la guardaremos como un atributo de la sesión (es decir, dicha estructura ENTERA la guardaremos como UN UNICO atributo de sesión). Guardaremos, para cada artículo, su nombre y su precio unitario, para luego mostrar estos dos datos en la página que se genere. Para tomar el nombre del artículo y el precio, notar que cada artículo tiene asociado en la página un formulario con dos campos ocultos *articulo* y *precio*, con estos elementos.

Crear una clase interna en el propio *CarroServlet* (llamadla *ObjetoCarro*, por ejemplo), que tenga como campos los valores que hay que guardar de cada artículo:

```
class ObjetoCarro
```

```
{
 String articulo;
 int precio;

 public ObjetoCarro(String articulo, int precio)
 {
  this.articulo = articulo;
  this.precio = precio;
 }

 public String getArticulo()
 {
  return articulo;
 }

 public int getPrecio()
 {
  return precio;
 }
}
```

Después, en el servlet *CarroServlet*, cada vez que el usuario compre un artículo, creáis un objeto de este tipo con los valores del artículo que haya enviado. Y lo añadís a la sesión, almacenando todos los objetos de tipo *ObjetoCarro* en una lista.

Mejoras para el carro de la compra (0.2 puntos)

Sobre el ejercicio anterior:

- Añadid un enlace u opción para invalidar la sesión al llamar a `org.expertojava.cweb.ejercicios.CarroServlet`. Comprobad que al pincharlo, y luego comprar un artículo, el carro aparecerá sólo con ese artículo.
- Si quisiéramos aplicar reescritura de URLs en el ejercicio anterior para prevenir que las cookies estén deshabilitadas, ¿qué cambios tendríamos que hacer? Dejadlos reflejados en el código.

4. Contexto global de la aplicación web

Vamos a estudiar los elementos que podemos utilizar para establecer una comunicación entre los distintos componentes de una aplicación web, tanto entre distintos servlets como entre servlets y otros elementos de la aplicación.

4.1. Contexto de los servlets

Comenzaremos estudiando el contexto de los servlets (*Servlet Context*). Este objeto de contexto es propio de cada aplicación web, es decir, tendremos un objeto `ServletContext` por aplicación web, por lo que nos servirá para comunicar los servlets de dicha aplicación.

```
public void init(ServletConfig config)
```

En la inicialización del servlet (método `init`), se nos proporcionará un objeto `ServletConfig` como parámetro. Mediante este objeto podemos:

- Obtener el nombre del servlet, que figurará en el descriptor de despliegue de la aplicación web (`web.xml` en Tomcat).

  ```
  String nombre = config.getServletName();
  ```

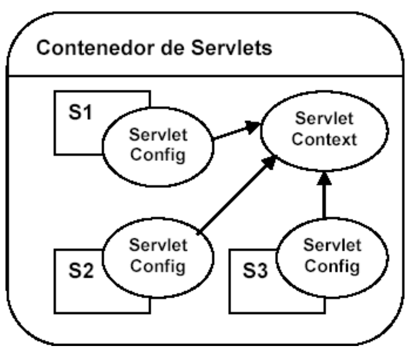
- Obtener los valores de los parámetros de inicialización del servlet, que se hayan establecido en el descriptor de despliegue. Tanto los nombres como los valores de los parámetros son cadenas de texto (`String`).

  ```
  String valor_param = config.getInitParameter(nombre_param);
  Enumeration nombres_params = config.getInitParameterNames();
  ```

- Acceder al objeto de contexto de la aplicación a la que pertenece el servlet.

  ```
  ServletContext context = config.getServletContext();
  ```

  Esta última función es la más importante, ya que nos permite acceder al objeto de contexto global de la aplicación, con el que podremos realizar una serie de operaciones que veremos a continuación.



Tanto el objeto `ServletConfig` como `ServletContext` pueden ser obtenidos directamente desde dentro de nuestro servlet llamando a los métodos `getServletConfig`

y `getServletContext` respectivamente, definidos en `GenericServlet`, y por lo tanto disponibles en cualquier servlet.

Atributos de contexto

Dentro del objeto de contexto de nuestra aplicación podremos establecer una serie de atributos, que serán globales dentro de ella. Estos atributos son un conjunto de pares <*nombre, valor*> que podemos establecer y consultar desde los distintos servlets de nuestra aplicación web. El nombre del atributo será una cadena de texto (`String`), mientras que el valor podrá ser cualquier objeto java (`Object`).

Para consultar el valor de un atributo utilizaremos:

```
Object valor = context.getAttribute(nombre);
```

Daremos valor a un atributo con:

```
context.setAttribute(nombre, valor);
```

Podemos también eliminar un atributo:

```
context.removeAttribute(nombre);
```

Lo cual dejará el atributo a `null`, igual que si nunca le hubiesemos asignado un valor. Por último, con

```
Enumeration enum = context.getAttributeNames();
```

Obtenemos la lista de nombres de atributos definidos en el contexto.

Hay que hacer notar en este punto, que el objeto de contexto a parte de ser propio de cada aplicación web, es propio de cada máquina virtual Java. Cuando trabajemos en un contexto distribuido, cada máquina ejecutará una VM distinta, por lo que tendrán también objetos de contexto diferentes. Esto hará que si los servlets de una aplicación se alojan en máquinas distintas, tendrán contextos distintos y este objeto ya no nos servirá para comunicarnos entre ellos. Veremos más adelante formas alternativas de comunicación para estos casos.

Parámetros de inicialización

El objeto `ServletConfig` nos proporcionaba acceso a los parámetros de inicialización del servlet en el que nos encontramos. Con `ServletContext`, tendremos acceso a los parámetros de inicialización globales de nuestra aplicación web. Los métodos para obtener dichos parámetros son análogos a los que usabamos en `ServletConfig`:

```
String valor_param = context.getInitParameter(nombre_param);
Enumeration nombres_params = context.getInitParameterNames();
```

Acceso a recursos estáticos

Este objeto nos permite además acceder a recursos estáticos alojados en nuestro sitio web. Utilizaremos los métodos:

```
URL url = context.getResource(nombre_recurso);
InputStream in = context.getResourceAsStream(nombre_recurso);
```

El nombre del recurso que proporcionamos será una cadena que comience por `"/"`, lo cual indica el directorio raiz dentro del contexto de nuestra aplicación, por lo tanto serán direcciones relativas a la ruta de nuestra aplicación web.

El primer método nos devuelve la URL de dicho recurso, mientras que el segundo nos devuelve directamente un flujo de entrada para leer dicho recurso.

Hay que señalar que esto nos permitirá leer cualquier recurso de nuestra aplicación como estático. Es decir, si proporcionamos como recurso `"/index.jsp"`, lo que hará será leer el código fuente del JSP, no se obtendrá la salida procesada que genera dicho JSP.

Podemos también obtener una lista de recursos de nuestra aplicación web, con:

```
Set recursos = context.getResourcePaths(String ruta);
```

Nos devolverá el conjunto de todos los recursos que haya en la ruta indicada (relativa al contexto de la aplicación), o en cualquier subdirectorio de ella.

Redirecciones

Si lo que queremos es acceder a recursos dinámicos, el método anterior no nos sirve. Para ello utilizaremos estas redirecciones. Utilizaremos el objeto `RequestDispatcher` que nos proporciona `ServletContext`.

Hemos de distinguir estas redirecciones de la que se producen cuando ejecutamos

```
response.sendRedirect();
```

Con `sendRedirect` lo que estamos haciendo es devolver al cliente una respuesta de redirección. Es decir, será el cliente, quien tras recibir esta respuesta solicite la página a la que debe redirigirse.

Con `RequestDispatcher` es el servidor internamente quien solicita el recurso al que nos redirigimos, y devuelve la salida generada por éste al cliente, pero todo ello de forma transparente al cliente. En cliente no sabrá en ningún momento que se ha producido una redirección.

Para obtener un objeto `RequestDispatcher` podemos usar los siguientes métodos de `ServletContext`:

```
RequestDispatcher rd = context.getRequestDispatcher(ruta);
RequestDispatcher rd = context.getNamedDispatcher(ruta);
```

Como ruta proporcionaremos la ruta relativa al contexto de nuestra aplicación, comenzando por el carácter `"/"`, del recurso al que nos queramos redirigir. También podemos obtener este objeto proporcionando una ruta relativa respecto al recurso actual, utilizando para ello el método `getRequestDispatcher` del objeto `ServletRequest`, en lugar de `ServletContext`:

```
RequestDispatcher rd = request.getRequestDispatcher(ruta);
```

Podemos utilizar el `RequestDispatcher` de dos formas distintas: llamando a su método `include` o a `forward`.

```
rd.include(request, response);
```

El método `include` incluirá el contenido generado por el recurso al que redireccionamos en la respuesta, permitiendo que se escriba este contenido en el objeto `ServletResponse` a continuación de lo que se haya escrito ya por parte de nuestro servlet. Se podrá llamar a este método en cualquier momento. Lo que no podrá hacer el recurso al que redireccionamos es cambiar las cabeceras de la respuesta, ya que lo único que estamos haciendo es incluir contenido en ella. Cualquier intento de cambiar cabeceras en la llamada a include será ignorado.

Si hemos realizado la redirección utilizando un método `getRequestDispatcher` (no mediante `getNamedDispatcher`), en la petición del servlet al que redireccionamos podremos acceder a los siguientes atributos:

```
javax.servlet.include.request_uri
javax.servlet.include.context_path
javax.servlet.include.servlet_path
javax.servlet.include.path_info
javax.servlet.include.query_string
```

Con los que podrá consultar la ruta desde donde fué invocado.

```
rd.forward(request, response);
```

El método `forward` sólo podrá ser invocado cuando todavía no se ha escrito nada en la respuesta del servlet. Esto es así porque esta llamada devolverá únicamente la salida del objeto al que nos redirigimos. Si esto no fuese así, se produciría una excepción `IllegalStateException`. Una vez el método `forward` haya devuelto el control, la salida ya habrá sido escrita completamente en la respuesta.

Si el recurso al que redireccionamos utiliza direcciones relativas, estás direcciones se considerarán relativas al servlet que ha hecho la redirección, por lo que si se encuentran en rutas distintas se producirá un error. Tenemos que hacer que las direcciones sean relativas a la raiz del servidor para que funcione correctamente (direcciones que comiencen por `"/"`).

Otros métodos

La clase `ServletContext` nos proporciona otros métodos de utilidad, que podremos consultar accediendo a su documentación JavaDoc.

Un método de interés es `log`, que nos permite escribir texto en el fichero de log del servlet:

```
context.log(mensaje);
```

Esto será util para tener un registro de eventos que ocurren en nuestra web, o bien para depurar errores.

Listeners de contexto

Existen objetos que permanecen a la escucha de los distintos eventos que pueden ocurrir en el objeto de contexto de servlets, `ServletContext`.

Un primer listener, es el `ServletContextListener`, que nos permitirá dar respuesta a los eventos de creación y destrucción del contexto del servlet. El código para este listener será como sigue a continuación:

```
import javax.servlet.*;

@WebListener
public class MiContextListener implements ServletContextListener {

    public void contextDestroyed(ServletContextEvent sce) {
        // Destruccion del contexto
    }

    public void contextInitialized(ServletContextEvent sce) {
        // Inicialización del contexto
    }
}
```

Esto nos será de gran utilidad si necesitamos inicializar ciertas estructuras de datos que van a utilizar varios servlets. De esta forma el contexto se habrá inicializado antes de que los servlets puedan ejecutarse.

Si lo que queremos es saber cuando se ha añadido, eliminado, o modificado alguno de los atributos del contexto global, podemos utilizar un listener `ServletContextAttributeListener`. Los métodos que deberemos definir en este caso son los siguientes:

```
import javax.servlet.*;

@WebListener
public class MiContextAttributeListener
                    implements ServletContextAttributeListener {

    public void attributeAdded(ServletContextAttributeEvent scae) {
        // Se ha a&ntilde;adido un nuevo atributo
    }

    public void attributeRemoved(ServletContextAttributeEvent scae) {
        // Se ha eliminado un atributo
    }
```

```
    public void attributeReplaced(ServletContextAttributeEvent scae) {
        // Un atributo ha cambiado de valor
    }
}
```

Hemos visto que podemos declarar *listeners* mediante la anotación `@WebListener`. Con esto el servidor de aplicaciones reconocerá estas clases como *listeners* del contenedor web y los registrará de forma automática sin necesidad de hacerlo nosotros. Los *listeners* que podremos declarar de esta forma son:

- `ServletContextListener`
- `ServletContextAttributeListener`
- `ServletRequestListener`
- `ServletRequestAttributeListener`
- `HttpSessionListener`
- `HttpSessionAttributeListener`

Sin embargo, esta anotación sólo está disponible a partir de la API de Servlets 3.0. En versiones anteriores para hacer que estos objetos se registren como listeners y permanezcan a la escucha, deberemos registrarlos como tales en el descriptor de despliegue de la aplicación (`web.xml`). Para ello deberemos añadir un elemento *<listener>* para cada objeto listener que queramos registrar:

```
<listener>
    <listener-class>MiContextListener</listener-class>
</listener>

<listener>
    <listener-class>MiContextAttributeListener</listener-class>
</listener>
```

Esta declaración no es necesaria si estamos utilizando la anotación `@WebListener` en Servlet 3.0.

Declaración dinámica de servlets

En Servlet 3.0 también tenemos la opción de configurar nuestros servlets de forma programática durante la inicialización del contexto, e incluso declarar servlets que no habían sido declarados previamente:

```
@WebListener
public class MiListener implements ServletContextListener {
  public void contextInitialized (ServletContextEvent sce) {
    ServletContext sc = sce.getServletContext();

    // Declara un nuevo servlet y lo configura
    ServletRegistration servletNuevo = sc.addServlet("miServlet",
      "es.ua.jtech.servlet.MiServlet");
    servletNuevo.addMapping("/UrlServlet");
```

```
        // Obtiene un servlet ya declarado para configurarlo
        ServletRegistration sr = sc.addServlet("otroServlet");
        sr.addMapping("/UrlOtroServlet");
        sr.setInitParameter("param1", "valor1");
    }
}
```

Por lo tanto, en Servlet 3.0 podemos declarar los servlets de tres formas distintas:

- En el descriptor de despliegue `web.xml`.
- Mediante anotaciones en la propia clase del servlet.
- De forma programática en la inicialización del contexto.

4.2. Inyección de dependencias

Los objetos de una aplicación normalmente necesitan otros objetos para realizar su tarea. Por ejemplo, un servlet puede necesitar una fuente de datos de la que obtener la información a mostrar al cliente web. Tradicionalmente, este servlet deberá obtener el objeto del que depende, por ejemplo buscando la fuente de datos en JNDI. El patrón de inyección de dependencias (DI) consiste en invertir este comportamiento: no será el servlet quién busque la fuende de datos, sino que la fuente de datos le será inyectada al servlet de forma externa. Las ventajas de este patrón son el bajo acoplamiento existente entre el servlet y los objetos de los que depende, y la facilidad para cambiar la implementación de la dependencia sin tener que modificar el código del servlet. Por ejemplo, esto facilitará las pruebas, ya que podremos sustituir la fuente de datos original por un *mock* sin tener que hacer ningún cambio en el código del servlet.

En Java SE 6 se integra la inyección de dependencias mediante la especificación *Contexts and Dependency Injection* (CDI, JSR-299). La implementación de referencia de CDI se denomina Weld (http://seamframework.org/Weld).

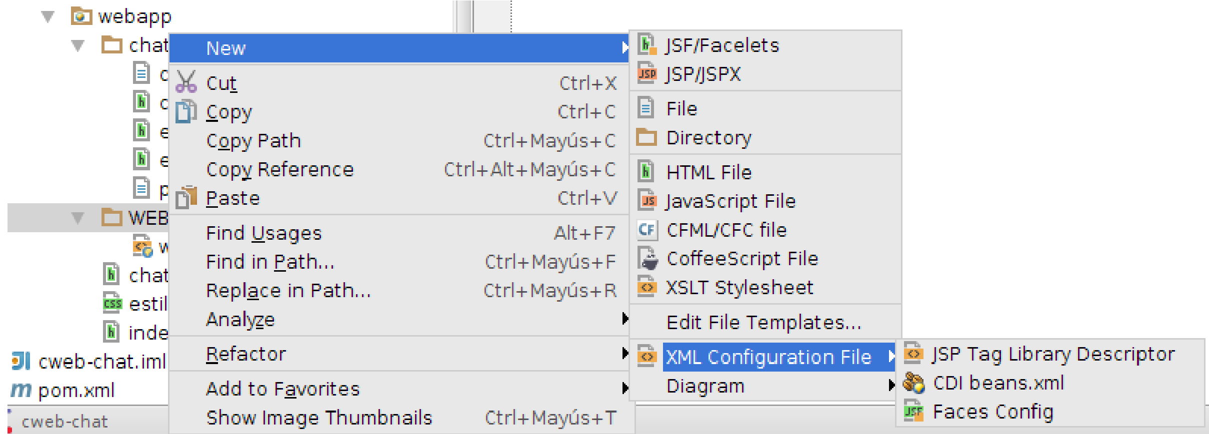
Configuración de Weld

Weld ya se encuentra integrado en la API de Java EE 7, por lo que no es necesario añadir esta librería como dependencia de forma explícita, a no ser que estuviésemos utilizando un contenedor web que no soporte la especificación completa de Java EE.

Para poder utilizar objetos CDI deberemos añadir un fichero `beans.xml` al directorio `WEB-INF` de nuestra aplicación web. Dicho fichero puede estar vacío, pero al menos debe existir. Si no queremos dejarlo vacío, podemos utilizar la siguiente plantilla inicial:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

Podemos crear este fichero desde IntelliJ pulsando con el botón derecho sobre el directorio `WEB-INF` y seleccionando *New > XML Configuration File > CDI beans.xml*.



Managed beans

Podemos inyectar distintos tipos de objetos, como fuentes de datos, contextos de persistencia JPA, o casi cualquier objeto Java plano (POJOs). Nos referiremos a los objetos que inyectamos como *managed beans*.

Casi cualquier clase Java puede ser un *managed bean*, sin ser necesario anotarla de ninguna forma. Para que una clase pueda comportarse como *managed bean* debe cumplir lo siguiente:

- No puede ser una clase interna, a no ser que sea de tipo `static`.
- Tiene un constructor sin parámetros.
- No tiene constructor sin parámetros, pero tiene un constructor anotado con `@Inject`.

Por ejemplo, el siguiente POJO podría ser inyectado como *managed bean*:

```
public class HolaMundo {
    public String saluda(String nombre) {
        return "Hola " + nombre;
    }
}
```

Inyección de objetos

Para inyectar un *bean* utilizamos la etiqueta `@Inject`. Vamos a ver el caso en el que se inyecta en un servlet, aunque podría inyectarse en cualquier otra clase (incluso dentro de otro *managed bean*):

```
@WebServlet(urlPatterns = "/miServlet")
public class MiServlet extends HttpServlet {
    @Inject
    private HolaMundo holaMundo;
}
```

El tiempo de vida del objeto inyectado dependerá del ámbito en el que se defina.

Ámbito de los beans

Los *managed beans* pueden existir en distintos ámbitos. Según el ámbito especificado mantendrá su estado durante un tiempo diferente:

Ámbito	Descripción
`@RequestScoped`	Se mantiene durante el tiempo que dure la petición (`request`) actual. Sólo será accesible por la petición actual.
`@SessionScoped`	Se mantiene durante el tiempo que dure la sesión (`session`) actual. Será accesible por todas las peticiones que se hagan dentro de la sesión actual.
`@ApplicationScoped`	Se mantiene durante toda la vida de la aplicación web (contexto). Será accesible por todas las peticiones de todas las sesiones.
`@Dependent`	Es el valor por defecto. Se mantiene con vida mientras exista el objeto en el que se ha inyectado.

Por ejemplo, podemos etiquetar el *bean* anterior para que sólo se mantenga con vida durante el tiempo que dure la petición de la siguiente forma:

```
@RequestScoped
public class HolaMundo {
    public String saluda(String nombre) {
        return "Hola " + nombre;
    }
}
```

Para que un *managed bean* anotado con `@SessionScoped` funcione correctamente deberemos hacer que sea `Serializable` . De no ser así, obtendremos el mensaje de error *Managed bean declaring a passivating scope must be passivation capable*.

Clasificadores

Podemos crear varias versiones de un *bean*, y distinguirlas mediante lo que se conoce como clasificadores (`qualifiers`). Podemos definir clasificadores de la siguiente forma:

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Dia {}

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Tarde {}

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Noche {}
```

Podríamos crear distintas versiones del *bean* `HolaMundo` , y etiquetar cada una de ellas con un clasificador distinto:

```
@Dia
public class HolaMundoDia extends HolaMundo {
    public String saluda(String nombre) {
        return "Buenos días " + nombre;
    }
}

@Tarde
public class HolaMundoTarde extends HolaMundo {
    public String saluda(String nombre) {
        return "Buenas tardes " + nombre;
    }
}

@Noche
public class HolaMundoNoche extends HolaMundo {
    public String saluda(String nombre) {
        return "Buenas noches " + nombre;
    }
}
```

Si un *bean* no lleva clasificador, se entiende que por defecto tiene el clasificador `@Default`. También podemos etiquetar alguna de las versiones de forma explícita con este clasificador por defecto.

Cuando inyectamos el *bean*, podemos indicar qué versión queremos inyectar mediante el clasificador:

```
@WebServlet(urlPatterns = "/miServlet")
public class ItemServlet extends HttpServlet {
    @Inject @Tarde
    private HolaMundo holaMundo;
}
```

Si no indicamos clasificador, se buscará la versión `@Default`, de la misma forma que si hiciésemos:

```
@WebServlet(urlPatterns = "/miServlet")
public class ItemServlet extends HttpServlet {
    @Inject @Default
    private HolaMundo holaMundo;
}
```

Productores

Podemos definir métodos que produzcan objetos para ser inyectados. De esta forma podemos definir la forma de obtener los objetos a inyectar. Hasta ahora, al inyectar un objeto lo que se hacía era construir una nueva instancia del objeto utilizando alguno de sus constructores, e inyectar dicha instancia. Utilizando un método productor podremos construir los objetos de otras formas.

Por ejemplo, nos puede ser útil para definir una fuente de datos:

```
public @interface FuenteDatos {}
public class ProductorConexiones {
    @Produces @FuenteDatos
    public Connection getConnection() throws Exception {
        Context ctx = new InitialContext();
        DataSource ds = (DataSource) ctx.lookup("jdbc/miBD");
        return ds.getConnection();
    }
}
```

Podemos inyectar las conexiones producidas por la anterior fuente de datos:

```
@Inject @FuenteDatos Connection conexion;
```

De esta forma inyectamos objetos `Connection` generados por una fuente de datos, que no son instanciados utilizando un constructor, sino una factoría, y que por lo tanto no podían ser inyectados como se ha visto anteriormente.

Definición de alternativas

Podemos definir distintas implementaciones alternativas para los *beans* a inyectar, de forma que en tiempo de despliegue podamos cambiar la implementación que se va a utilizar. Esto es especialmente útil para la implementación de pruebas. Por ejemplo, si tenemos el siguiente *bean*:

```
public class FuenteDatos implements IFuenteDatos
```

Podemos definir una implementación alternativa que contenga un `mock` para realizar pruebas:

```
@Alternative
public class MockFuenteDatos implements IFuenteDatos
```

Por defecto siempre se utilizará la implementación original, a no ser que en el fichero `beans.xml` indiquemos el nombre de la implementación alternativa:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
                 http://java.sun.com/xml/ns/javaee
                 http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
    <alternatives>
        <class>es.ua.jtech.MockFuenteDatos</class>
    </alternatives>
</beans>
```

En este caso, para usar el *mock* deberemos incluir el fichero `beans.xml` que así lo indique. También podemos crear la clase alternativa como una especialización:

```
@Specializes
public class MockFuenteDatos extends FuenteDatos
```

En este caso la especialización siempre sustituirá a la original. De esta forma, si incluimos la especialización en el carpeta de fuente de prueba (*test*), cuando incluyamos dichas clases en la aplicación siempre se usará el *mock*, mientras que en el caso de compilar únicamente las clases de la carpeta *main* para poner la aplicación en producción, se utilizará la implementación original. Esta es una forma sencilla de sustituir clases por *mock* en nuestras pruebas.

4.3. Ejercicios

Ejemplo de contexto (0 puntos)

Vamos a probar la aplicación `cweb-contexto` incluida en los ejercicios de la sesión.

a) Desplegar la aplicación web en Tomcat, y acceder a la dirección:

```
http://localhost:8080/cweb-contexto
```

Veremos la aplicación web ya en marcha.

b) La aplicación web nos permite visualizar los atributos de contexto definidos y sus valores, y añadir nuevos atributos. A parte de los atributos que nosotros añadimos manualmente, ¿hay más atributos de contexto definidos?

c) Podemos añadir nuevos atributos de contexto. Daremos un nombre del atributo, y un texto que contendrá como valor. Además como valor también se introducirá el identificador de sesión del navegador que haya creado dicho atributo. Abrir distintos navegadores y añadir atributos de contexto desde cada uno. Comprobar que en cada navegador vemos tanto los atributos creados en su sesión, como lo atributos creados creados en las sesiones de otros navegadores (el identificador de sesión será distinto).

d) Si nos fijamos en el paquete `org.expertojava.cweb.contexto.listener`, veremos que se ha añadido un listener sobre los atributos del contexto. Este listener imprime mensajes en el *log* indicando cuando se añade, elimina o reemplaza un atributo de contexto. Comprobar en el fichero de logs correspondiente que se han registrado los cambios en los atributos que hayamos hecho.

Chat con servlets (0.7 punto)

Vamos a realizar una aplicación de chat utilizando servlets. En el directorio `cweb-chat` de los fuentes de la sesión podrás encontrar la base sobre la que construiremos el chat. Cada mensaje de chat estará encapsulado en la clase `Mensaje`, y la lista de mensajes actual del chat se encontrará en la clase `ColaMensajes`.

Además se proporcionan los ficheros HTML necesarios para la aplicación. El fichero `index.html` contiene el formulario de login para que un usuario introduzca el *nick* con el que entrará en el chat (no se solicita ningún password para validar). El login se hará efectivo por el servlet `LoginUsuarioServlet` también proporcionado, que introducirá el *nick* del usuario en la información de sesión y nos redirigirá al chat. En el subdirectorio `chat` tendremos los ficheros estáticos del chat:

`chatFrames.html`	Página principal de los frames de la aplicación chat. Mostrará un frame con el formulario para enviar mensajes, y otro con la lista de mensajes enviados.
`enviaMensaje.html`	Formulario para enviar mensajes al chat.
`error.html`	Página que nos muestra un mensaje de error cuando se intenta enviar un mensaje sin haber hecho *login*.

`cabecera.htmlf`	Cabecera de la tabla de mensajes, a incluir al comienzo de la página de lista de mensajes.
`pie.htmlf`	Pie de la tabla de mensajes, a incluir al final de la página de lista de mensajes.

Ahora deberemos implementar los servlets para el envio de mensajes y para la consulta de la lista de mensajes enviados. Se pide:

a) La cola de mensajes será el objeto común al que acceden los servlets para el envio y la consulta de estos mensajes. Por lo tanto el objeto deberá añadirse como atributo del contexto. Esto lo tendremos que hacer antes de que cualquier servlet se haya ejecutado. Para ello debemos crear un objeto `ServletContextListener` que en la creación del contexto inicialice la cola de mensajes (`ColaMensajes`) y la introduzca como atributo en el contexto global (atributo `org.expertojava.cweb.chat.mensajes`).

b) Una vez tenemos creada la cola de mensajes, deberemos implementar el servlet `EnviaMensajeServlet` , que tome un mensaje como parámetro (el nombre del parámetro es `texto`), y lo añada a la lista de mensajes con el nick del usuario actual (obtenido del atributo `org.expertojava.cweb.chat.nick` de la sesión). Una vez enviado el mensaje, mostraremos en la salida el contenido de `enviaMensaje.html` , mediante un objeto `RequestDispatcher` . Si no hubiese ningún usuario en la sesión, no se registrará el mensaje y se deberá redirigir la salida a `error.html`

c) Por último, deberemos implementar el servlet `ListaMensajesServlet` que mostrará todos los mensajes del chat. Este servlet debe:

- Para que la lista de mensajes se actualice periodicamente en el cliente, haremos que se recargue cada 5 segundos. Añadir la cabecera HTTP correspondiente a la respuesta para que esto ocurra (cabecera `Refresh` , indicando como valor el número de segundos que tardará en recargar).
- Incluir el contenido del fichero estático `cabecera.htmlf` al comienzo del documento generado, y `pie.htmlf` al final, para enmarcar la zona donde aparecen los mensajes del chat.

Se debe obtener el `PrintWriter` para escribir la respuesta antes de hacer el primer `include` , ya que de lo contrario se obtendría una excepción de tipo `IllegalStateException` . Esto de debido a que el `include` ya habría obtenido previamente el flujo de salida de la respuesta.

- Obtener el *nick* del usuario actual de la sesión. Los mensajes enviados con este *nick* se mostrarán en negrita, el resto se mostrarán de forma normal.

d) Comprobar que el chat funciona correctamente. Conectar desde varios clientes a un mismo servidor.

Inyección de dependencias (0.3 puntos)

Utiliza inyección de dependencias para gestionar la cola de mensajes del chat anterior. En este caso:

- Deberás configurar correctamente los ficheros `web.xml` y `beans.xml` .

- Deberás crear un *bean* que gestione la cola de mensajes del chat.
- Deberás especificar el ámbito adecuado en el *bean* anterior.
- Ya no será necesario utilizar un *listener* del contexto para inicializarla, se inicializará la primera vez que se inyecte.
- Ya no será necesario acceder al atributo de contexto con la cola de mensajes. La cola será inyectada en los servlets.

5. WebSocket

Hasta el momento hemos visto los servlets como un componente web que encapsula el mecanismo petición/respuesta definido en el protocolo HTTP. Sin embargo, en muchas ocasiones nos interesa mantener un canal de comunicación abierto con el servidor, para así poder recibir información del mismo sin tener que interrogarlo mediante peticiones continuamete.

Pongamos por ejemplo el caso de una aplicación de chat, en la que el cliente debe estar pendiente de actualizar el listado de mensajes cuando otros usuarios realicen una publicación. Con protocolo HTTP tendríamos que estar continuamente realizando peticiones (por ejemplo mediante AJAX) para comprobar si hay nuevos mensajes, y en tal caso actualizar la lista. Si la frecuencia de peticiones es alta estaremos malgastando ancho de banda, pero si es baja tendremos un mayor retraso en la actualización de los mensajes. Este es un escenario donde es conveniente utilizar WebSocket en lugar de HTTP.

5.1. WebSocket en Java EE

La API JSR 356 (Java API for WebSocket) nos permite crear *endpoints* WebSocket dentro de una aplicación web Java EE. Esta API se define dentro del paquete `javax.websocket`, cuya clase principal es `Endpoint`, la cual nos permitirá crear un *endpoint* de tipo WebSocket, aunque también podremos crear *endpoints* añadiendo anotaciones a cualquier clase Java.

Endpoints programados

Se crean mediante una subclase de `Endpoint`, en la que tendremos que sobrescribir los métodos de su ciclo de vida:

- `onOpen`: Es el **único método obligatorio** de implementar. Indica que se ha abierto el canal de comunicación.
- `onClose`: Indica que se ha cerrado el canal de comunicación.
- `onError`: Indica un error en la conexión

```
public class MiEndpoint extends Endpoint { ❶
   @Override
   public void onOpen(final Session session, EndpointConfig config) { ❷
      session.addMessageHandler(new MessageHandler.Whole<String>() { ❸
         @Override
         public void onMessage(String msg) { ❹
            // Interactuar con el objeto session para intercambiar datos
            ...
         }
      });
   }
}
```

❶ La clase debe heredar de `Endpoint`
❷ Debemos sobrescribir de forma obligatoria al menos el método `onOpen`
❸ Normalmente definiremos un manejador de mensajes sobre la sesión (`MessageHandler`)

❹ En el manejador de mensajes debemos definir el método `onMessage` que nos notifica la llegada de un mensaje

Todos los métodos anteriores reciben como parámetro un objeto `Session` que representa la conversación con el cliente. Normalmente definiremos un `MessageHandler` sobre la sesión para gestionar el intercambio de mensajes. Este manejador nos permite definir un método `onMessage` que nos notificará la llegada de un mensaje desde el cliente.

El despliegue de este tipo de *endpoints* se hace también de forma programada. Para ello deberemos introducir el siguiente código:

```
ServerEndpointConfig.Builder.create(MiEndpoint.class, "/socket").build();
```

Con esto el *endpoint* quedará publicado en la siguiente dirección:

ws://localhost:8080/miaplicacion/socket

Endpoints mediante anotaciones

Una forma más sencilla de crear *endpoints* consiste en hacerlo mediante anotaciones. Podemos crear un *endpoint* equivalente al anterior mediante anotaciones con el siguiente código:

```
@ServerEndpoint("/socket") ❶
public class EchoEndpoint {
   @OnMessage ❷
   public void onMessage(Session session, String msg) {
      // Interactuar con el objeto session para intercambiar datos
      ...
   }
}
```

❶ Un *endpoint* se declara con la anotación `ServerEndpoint`, que además nos permite especificar la dirección de despliegue

❷ El método `onMessage` se puede definir mediante una anotación directamente. No es necesario crear un manejador de mensajes.

Lo más destacable es que ya no es necesario realizar el despliegue de forma programada. Con añadir la anotación `ServetEndpoint` y como atributo suyo la dirección de despliegue el *endpoint* quedará desplegado en el servidor.

También es importante destacar que disponemos de las siguientes anotaciones para los métodos:

Anotación	Descripción
`@OnOpen`	Se llama al abrirse el canal de datos
`@OnClose`	Se llama al cerrarse el canal de datos
`@OnError`	Se llama al producirse un error en la comunicación
`@OnMessage`	Se llama al recibirse un mensaje

A continuación se muestra un ejemplo de *endpoint* con los 4 tipos de métodos:

```
@ServerEndpoint("/socket")
public class EchoEndpoint {

    @OnOpen
    public void open(Session session,
                     EndpointConfig conf) {
    }

    @OnClose
    public void close(Session session,
                      CloseReason reason) {
    }

    @OnError
    public void error(Session session,
                      Throwable error) {
    }

    @OnMessage
    public void onMessage(Session session,
                          String msg) {
    }
}
```

Podemos observar que en todos ellos siempre se proporciona como primer parámetro el objeto `Session`. El segundo parámetro dependerá del método.

Mantenimiento del estado

Al contrario que los *servlets*, los *endpoints* WebSocket se instancian una vez por cada cliente conectado. De esta forma podemos utilizar la misma clase del *endpoint* para mantener el estado del cliente. Tenemos dos formas de guardar datos de estado:

- Utilizar variables de instancia de la clase que implementa el *endpoint*.
- Utilizar el mapa `session.getUserProperties()` para guardar la información en forma de parejas *<clave, valor>*.

5.2. Intercambio de mensajes

Vamos a pasar a estudiar la forma de intercambiar información mediante WebSocket. Las operaciones básicas de intercambio de datos que encontramos en WebSocket son:

- Envío de datos (de texto o binarios)
- *Ping*: Mensaje de control, puede contener datos
- *Pong*: Respuesta al *ping*, puede contener datos

Envío de mensajes

Para enviar un mensaje necesitaremos siempre un objeto `Session`. Todos los métodos anotados nos proporcionan este objeto como parámetro, por lo que al recibir un mensaje de entrada en `onMessage` podremos responder utilizando el objeto `Session` que proporciona como parámetro. En caso de que necesitemos enviar mensajes que no sean respuesta a un

mensaje de entrada deberemos guardar el objeto `Session` en una variable de instancia de nuestra clase en `onOpen` para así tenerlo disponible en cualquier momento.

A partir del objeto `Session` deberemos obtener un objeto `RemoteEndpoint` para enviar el mensaje al *endpoint* remoto. Encontramos dos tipos de `RemoteEndpoint`:

- **Básico**: Se define mediante la clase `RemoteEndpoint.Basic`, y se obtiene con `session.getRemoteBasic()`.
- **Asíncrono**: Se define mediante la clase `RemoteEndpoint.Async`, y se obtiene con `session.getRemoteAsync()`. Permite realizar las operaciones de forma no bloqueante.

Contamos con las siguientes operaciones para intercambiar datos:

Operación	Descripción
sendText(String)	Envia un mensaje de texto al *endpoint* remoto de forma bloqueante o no bloqueante (según el tipo de `RemoteEndpoint`)
sendBinary(ByteBuffer)	Envia un mensaje binario al *endpoint* remoto de forma bloqueante o no bloqueante (según el tipo de `RemoteEndpoint`)
sendPing(ByteBuffer)	Envía un *ping* al *endpoint* remoto
sendPong(ByteBuffer)	Contesta con un *pong* al *endpoint* remoto

Por ejemplo, podremos enviar un mensaje de texto de la siguiente forma:

```
@ServerEndpoint("/socket")
public class MiEndpoint {
    @OnMessage
    public void onMessage(Session session, String msg) {
        session.getBasicRemote().sendText("Recibido " + msg); ❶
    }
}
```

❶ Envía un mensaje de texto al *endpoint* remoto.

Podemos también enviar un mensaje a todos los *endpoints* remotos conectados actualmente a nuestro *endpoint*. Para ello podemos utilizar el método `getOpenSessions` del objeto `Session`, que nos proporciona la lista de todas las sesiones abiertas. Esto será útil por ejemplo en una aplicación de chat, en la que al recibir un mensaje de un cliente deberemos difundirlo a todos:

```
@ServerEndpoint("/chat")
public class ChatEndpoint {
    @OnMessage
    public void onMessage(Session session, String msg) {
        try {
            for (Session s : session.getOpenSessions()) {
                if (s.isOpen())
                    s.getBasicRemote().sendText(msg);
            }
```

```
        } catch (IOException e) { }
    }
}
```

Recepción de mensajes

Podemos recibir tres tipos de mensajes, según el tipo de parámetros del método etiquetado como `@OnMessage`:

- Mensajes de texto (`String` , `Reader`)
- Mensajes binarios (`byte[]` , `ByteBuffer` , `InputStream`)
- Mensajes *pong* (`PongMessage`)

A continuación se muestra un ejemplo de *endpoint* que puede recibir los tres tipos de mensajes:

```
@ServerEndpoint("/socket")
public class ReceiveEndpoint {
   @OnMessage
   public void texto(Session session, String msg) { ❶
   }

   @OnMessage
   public void binario(Session session, ByteBuffer msg) { ❷
   }

   @OnMessage
   public void pong(Session session, PongMessage msg) { ❸
   }
}
```

❶ Recepción de un mensaje de texto
❷ Recepción de un mensaje binario
❸ Recepción de una respuesta *pong*

5.3. Conversión entre Java y mensajes WebSocket

Aunque los mensajes que se intercambian se limitan a ser de tipo texto o tipo binario, podemos definir un mapeo entre nuestros objetos Java y una determinada codificación (JSON, XML, etc) de forma que se realice una conversión automática entre el mensaje WebSocket y una clase Java. Esto lo haremos definiendo objetos `Encoder` que definan la forma de realizar la conversión de Java a WebSocket, y objetos `Decoder` que realicen la conversión en el sentido inverso.

Imaginemos que tenemos una aplicación que trabaja con películas, que constan de título, director y duración. Podemos tener una clase Java como la siguiente para encapsular los datos de cada película:

```
public class Pelicula {
   String titulo;
   String director;
   int duracion;
```

```
    // Getters y setters
    ...
}
```

Encoders

Podríamos definir un `Encoder` que transforme un objeto de nuestra clase `Pelicula` en texto (`Encoder.Text`) o en binario (`Encoder.Binary`). Vamos a ver un ejemplo en el que transformamos la película a texto con formato *<titulo>;<director>;<duracion>*, por lo que utilizaremos un objeto de tipo `Encoder.Text` :

```
public class PeliculaAJsonEncoder implements Encoder.Text<Pelicula> {
    @Override
    public void init(EndpointConfig ec) { }

    @Override
    public void destroy() { }

    @Override
    public String encode(Pelicula p) throws EncodeException {
        String msg = p.getTitulo() + ";" +
                     p.getDirector() + ";" +
                     p.getDuracion();
        return msg;
    }
}
```

Decoders

Con un `Decoder` realizaremos la operación inversa: transformamos un mensaje WebSocket en un objeto Java. Siguiendo con el ejemplo anterior, podríamos transformar un mensaje JSON son los datos de un libro en un objeto `Libro` definiendo el siguiente `Decoder` , también de tipo `Decoder.Text` :

```
public class JsonAPeliculaDecoder implements Decoder.Text<Pelicula> {
    @Override
    public void init(EndpointConfig ec) { }

    @Override
    public void destroy() { }

    @Override
    public Pelicula decode(String string) throws DecodeException { ❶
        String [] items = string.split(";");
        Pelicula p = new Pelicula();
        p.setTitulo(items[0]);
        p.setDirector(items[1]);
        p.setDuracion(Integer.parseInt(items[2]));

        return p;
    }

    @Override
```

```
    public boolean willDecode(String string) { ❷
        boolean mensajeValido = string.matches("[A-Za-z0-9]+;[A-Za-z0-9]+;
[0-9]+");
        return mensajeValido;
    }
}
```

❶ Debemos transformar la cadena de texto en nuestro objeto Java
❷ Debemos comprobar si la cadena de texto tiene el formato correcto

Sólo podemos definir un único `Decoder` para todos los mensajes de texto. Si fuese posible recibir más de un tipo de objeto deberemos crear una superclase para todos los tipos de mensaje, y crearemos el `Decoder` para el tipo de la superclase. Dentro del `Decoder` deberemos distinguir de qué tipo es el mensaje y devolver la instancia adecuada. Por ejemplo, si además de `Pelicula` tuviesemos `Libro` y `Disco`, tendríamos que hacer que todas ellas heredasen de una misma clase, por ejemplo `Articulo` (`Pelicula extends Articulo`). El *decoder* tendría que definirse en este caso del tipo `Decoder.Text<Articulo>`, y dentro del método `decode` en función del tipo de mensaje detectado instanciaríamos y devolveríamos la subclase correcta.

Uso de *encoders* y *decoders*

Una vez definidos el `Encoder` y el `Decoder`, deberemos añadirlos a la anotación `ServerEndpoint` para que la operación de conversión se realice de forma automática:

```
@ServerEndpoint(
    value = "/socket",
    encoders = { PeliculaAJsonEncoder.class } ❶
    decoders = { JsonAPeliculaDecoder.class } ❷
)
public class MiEndpoint {
    @OnMessage
    public void libro(Session session, Pelicula p) { ❸
        Pelicula nuevaPelicula = new Pelicula();
        nuevaPelicula.setTitulo("Copia de " + p.getTitulo());
        session.getBasicRemote.sendObject(nuevaPelicula); ❹
    }
}
```

❶ Declaramos el `Encoder` en la anotación `ServerEndpoint`
❷ Declaramos el `Decoder` en la anotación `ServerEndpoint`
❸ Podemos utilizar `Libro` como tipo de datos del mensaje recibido en `@OnMessage`
❹ Podemos enviar directamente objetos `Libro` con `sendObject`

5.4. Parámetros del *path* y de la *query*

Al mapear el *endpoint* a una URL podemos especificar segmentos variables del *path*:

```
@ServerEndpoint("/tiempo/{cp}")
```

```
public class TiempoEndpoint {
    ...
}
```

De esta forma podremos acceder al *endpoint* con diferentes URLs, siempre que cumplan el patrón anterior (el segmento `{cp}` puede tomar cualquier valor):

ws://localhost:8080/miaplicacion/tiempo/03001

ws://localhost:8080/miaplicacion/tiempo/03004

ws://localhost:8080/miaplicacion/tiempo/03690

El valor introducido podrá ser inyectado como parámetro de los métodos de nuestro *endpoint* añadiendo la anotación `@PathParam(<nombre_segmento>)`. Podrá inyectarse en los métodos de tipo `onOpen`, `onClose` y `onMessage`. A continuación se muestra un ejemplo de inyección en `onOpen`:

```
@ServerEndpoint("/tiempo/{cp}")
public class ChatEndpoint {
    String cp;
    Session session;

    @OnOpen
    public void open(Session session,
                     EndpointConfig c,
                     @PathParam("cp") String cp) { ❶
        this.cp = cp; ❷
        this.session = session; ❸
    }
}
```

❶ Inyección del valor introducido en el segmento variable `{cp}`

❷ Almacenamos el valor del código postal para recordarlo cuando vayamos a enviar una actualización de los datos del tiempo al cliente

❸ Almacenamos la sesión para poder enviar mensajes al cliente sin la necesidad de que exista un mensaje de entrada

Este tipo de parámetros se conoce como parámetros del *path*. Podríamos también añadir parámetros en la *query* con el siguiente formato:

ws://localhost:8080/miaplicacion/tiempo?**hora=18**

En este caso la forma de obtenerlos en el *endpoint* es distinta. Podremos acceder a estos parámetros a través del objeto `Session` con `getRequestParameterMap`. Esta función nos devuelve un `Map` con todos los parámetros recibidos, que podrían ser multivaluados. El parámetro `cp` anterior se obtendría de la siguiente forma:

```
String hora = session.getRequestParameterMap().get("hora").get(0);
```

Tenemos que poner `.get(0)` debido a que para cada parámetro nos da una lista de objetos `String`, para así permitir los parámetros con múltiples valores. En conveniente que nos aseguremos de que el parámetro es distinto de `null` y que la lista no está vacía.

```
if(session.getRequestParameterMap().get("cp")!=null &&
   session.getRequestParameterMap().get("cp").size() > 0) {
     String cp = session.getRequestParameterMap().get("cp").get(0);
}
```

Podríamos combinar los dos tipos de parámetros en una misma URL, por ejemplo:

ws://localhost:8080/miaplicacion/tiempo/03001?hora=18

5.5. Cliente JavaScript

La gran mayoría de navegadores actuales soportan WebSocket. Encontramos una API JavaScript que nos permite conectar a este tipo de componentes. Deberemos crear un objecto JavaScript de tipo `WebSocket` a partir de la URL de nuestro *endpoint*:

```
websocket = new WebSocket("ws://localhost:8080/miaplicacion/socket");
```

Una vez creado el objeto, deberemos especificar un *callback* en su atributo `onmessage` para recibir una notificación cuando recibamos un nuevo mensaje:

```
websocket.onmessage = onMessage;
```

A continuación se muestra un ejemplo completo, en el que suponemos que se reciben datos de una película con el formato *<titulo>;<director>;<duracion>* de los ejemplos anteriores. Se trocea la cadena y se muestra cada elemento en el documento web:

```
var websocket;

function connect() {
   websocket = new WebSocket("ws://localhost:8080/miaplicacion/socket");
   websocket.onmessage = onMessage;
}

function onMessage(event) {
   var items = event.data.split(";");
   document.getElementById("titulo").innerHTML = items[0];
   document.getElementById("director").innerHTML = items[1];
}

window.addEventListener("load", connect, false);
```

A través del objeto `WebSocket` podemos también enviar mensajes con `send`:

```
websocket.send("Mensaje");
```

5.6. Ejercicios

Chat básico con WebSocket (0.5 puntos)

Vamos a crear una versión alternativa del chat utilizando WebSocket. Para empezar simplemente implementaremos un chat que nos permita intercambiar mensajes con el resto de clientes conectados al servidor sin incluir información del emisor de cada mensaje.

Tenemos en la aplicación `cweb-chat` una página `chatws.html` que contiene el código JavaScript que implementa el chat. Deberemos crear el *endpoint* en el lado del servidor:

- Creamos un *endpoint* mapeado a la dirección `/ChatWS`.
- En el *endpoint* definimos un método `OnMessage` que difunda el mismo mensaje que reciba a todas las sesiones abiertas actualmente.
- El mensaje a difundir debe tener el siguiente formato:

```
<nick>;<mensaje-rebido>
```

  De momento utilizaremos como *nick* la cadena fija `"Anonimo"`. En el próximo ejercicio implementaremos *nicks* y salas de chat.
- Prueba el chat y comprueba que funciona correctamente desde diferentes navegadores.

Chat con nombres de usuario y salas (0.5 puntos)

En este ejercicio vamos a mejorar el chat anterior añandiendo distintas salas de chat y asignando a cada cliente un *nickname*. La sala se especificará mediante un segmento de la URL (*path param*), mientras que el *nickname* llegará como parámetro de la *query*:

ws://localhost:8080/cweb-chat/ChatWS/SalaA?nick=Pepe

Deberemos:

- En primer lugar en el fichero `chatws.html` comentaremos la siguiente línea del código JavaScript:

```
window.addEventListener("load", connect, false);
```

  Con esto la conexión con el *endpoint* dejará de funcionar al hacerse la petición a una ruta distinta a la que está mapeado (no está preparado para recibir el segmento con el nombre de la sala).
- Ahora deberemos mapear correctamente el *endpoint* a una dirección que acepte el *path param* con el nombre de la sala como segmento de ruta. Comprueba que la conexión funciona correctamente tras establecer este mapeo.
- Implementa soporte para establecer el *nick*. Define para ello en el *endpoint* un método `OnOpen` que lea el parámetro de la *query* `nick` y guarde el *nick* en una variable de instancia. Cuando difunda los mensajes en `OnMessage` incluiremos el *nick* correcto en lugar de `"Anonimo"`.
- Por último, implementaremos la posibilidad de tener múltiples salas de chat, cada una de ellas identificadas por un nombre. Sólo veremos los mensajes de los usuario que estén en nuestra misma sala. Para ello:

En primer lugar en `OnOpen` inyectaremos el *path param* como parámetro.
Guardaremos el nombre de la sala en el mapa `session.getUserProperties()`.
Al difundir los mensajes comprobaremos si en nuestra la propiedad *sala* coincide con cada una de las otras sesiones abiertas. Sólo enviaremos mensajes a las sesiones en las que coincida.

6. Seguridad en aplicaciones web

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:

- **Autentificación** y **autorización**: La autentificación se refiere a identificar a los actores que se conectan, lo cual se hará normalmente aportando unas credenciales (*login* y *password*), mientras que la autorización se refiere a distinguir las operaciones que cada actor puede realizar.
- **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.

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 autentificado 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 (fichero *web.xml*), 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.

Vamos a centrarnos en el estudio de la **autentificación** y **autorización** mediante **seguridad declarativa** en servidores de aplicaciones Java EE.

6.1. Mecanismos de autentificación

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

Autentificaciones elementales

El protocolo HTTP incorpora un mecanismo de autentificación básico (**basic**) basado en cabeceras de autentificación 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 autentificación no proporciona confidencialidad ni integridad, sólo se emplea una codificación Base64.

Una variante de esto es la autentificación **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 autentificación.

Certificados digitales y SSL

Las aplicaciones reales pueden requerir un nivel de seguridad mayor que el proporcionado por las autentificaciones *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 desencriptar 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 desencriptar 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 desencriptan bien, indica que son correctos). Sin embargo, no proporciona **autentificación** (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 desencriptarlos. 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 autentificación 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 autentificación del cliente. Funcionan de la misma forma que los explicados para el servidor, pero residiendo en el cliente.

6.2. Usuarios y roles

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

Los mecanismos de autentificación en WildFly se basan en el concepto de *realm*. Un *realm* es un conjunto de usuarios, cada uno con un *login* y *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`). El *login* y el *password* se utilizará para la autentificación de los usuarios, y los roles para su autorización.

Podemos encontrar distintos tipos de *realms*, que básicamente se diferencian en dónde están almacenados los datos de *logins*, *passwords* y *roles*. Por defecto en WildFly tenemos dos realms que almacenan estos datos en ficheros de texto de tipo `.properties`. Existen otros *realms* que leer los datos de bases de datos con JDBC, de un directorio LDAP o mediante JAAS, el API estándar de Java para autentificación.

Los dos *realms* que tenemos por defecto en WildFly son `ManagementRealm` y `ApplicationRealm`. `ManagementRealm` se utiliza para la aplicación de administración del servidor, por lo que sólo nos permite controlar la autentificación (no se indican roles porque el único rol que tienen los usuarios de este conjunto es el de administrar el servidor). Por otro lado, `ApplicationRealm` nos permite además controlar la autorización, mediante la asignación de roles a usuarios.

No será necesario editar manualmente los ficheros de usuarios de estos *realms* por defecto, ya que se proporciona una herramienta para añadir nuevos usuarios a ellos. Este herramienta se encuentra en el directorio `$WILDFLY_HOME/bin`, y se ejecuta de la siguiente forma:

```
$ ./addUser.sh
```

Nos preguntará en primer lugar en cuál de los dos *realms* por defecto queremos introducir el usuario, y a continuación nos irá pidiendo los datos del nuevo usuario. En las aplicaciones que incorporen seguridad declarativa se nos permitirá entrar con cualquiera de los usuarios del `ApplicationRealm`. Las operaciones que nos permita hacer dependerán de los roles asignados.

6.3. Autentificación en aplicaciones web Java EE

En las aplicaciones web Java EE tenemos distintos tipos de autentificación que podemos emplear:

- **Autentificación basic**: Con HTTP se proporciona un mecanismo de autentificación básico, basado en cabeceras de autentificación para solicitar datos del usuario (el servidor) y para enviar los datos del usuario (el cliente). Esta autentificación no proporciona confidencialidad ni integridad, sólo se emplea una codificación Base64.
- **Autentificación digest**: Existe una variante de lo anterior, la autentificación **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 autentificación.

- **Autentificación basada en formularios**: Con este tipo de autentificación, 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 autentificación y una página de error. Tiene el mismo inconveniente que la autentificación *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 autentificación.

Autentificación basada en formularios

Veremos ahora con más profundidad la autentificación 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 autentificación incorrecta. Así, un usuario que intente acceder a la parte restringida es redirigido automáticamente a la página del formulario, si no ha sido autentificado 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 autentificación 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 autentificación 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 autentificación 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 autentificación basada en formularios (como es el caso)
  - `BASIC`: para autentificación BASIC
  - `DIGEST`: para autentificación 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 autentificación)

`<form-error-page>` (para la página de error).

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

```
<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 autentificación, 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 autentificación

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)

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 a 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:

```
<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.

Autentificación *basic*

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

1. Establecer los logins, passwords y roles

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

2. Indicar al servlet que se empleará autentificación 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:

```
<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 autentificación

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

6.4. 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. La anotación principal que utilizaremos es `@ServletSecurity`, que toma dos parámetros:

- `value`: Define la restricción de seguridad, mediante una anotación de tipo `@HttpConstraint`.
- `httpMethodConstraints`: Permite definir una lista de restricciones sobre los métodos HTTP que pueden acceder al servlet. Se definen mediante una anotación de tipo `@HttpMethodConstraint`. De esta manera no damos una restricción general para todos los métodos, sino que podemos personalizar las restricciones que se aplicarán para cada uno de ellos.

La anotación `@HttpConstraint` acepta los siguientes parámetros:

- `rolesAllowed`: Nos permite definir la lista de roles que pueden acceder al servlet.
- `transportGuarantee`: Puede tomar los valores `TransportGuarantee.NONE` o `TransportGuarantee.CONFIDENTIAL`. Con el segundo de ellos sólo estaremos permitiendo acceder al servlet si la conexión se realiza mediante SSL.
- `value`: Nos permite establecer la política de acceso a llevar a cabo independientemente del rol del usuario. Las opciones son admitir todas las peticiones (`EmptyRoleSemantic.PERMIT`) o denegarlas (`EmptyRoleSemantic.DENY`). Definiremos estas políticas cuando no se especifique una lista de roles.

Un caso común es aquel en el que queremos permitir acceso al servlet a unos roles determinados:

```
@WebServlet("/MiServlet")
@ServletSecurity(@HttpConstraint(rolesAllowed={"rol1","rol2"}))
public class MiServlet extends HttpServlet { ... }
```

Si queremos permitir el acceso a cualquier usuario (aunque no esté autentificado), pero siempre mediante SSL, podemos indicarlo de la siguiente forma:

```
@WebServlet("/MiServlet")
@ServletSecurity(@HttpConstraint(value=EmptyRoleSemantic.PERMIT,
              transportGuarantee=TransportGuarantee.CONFIDENTIAL)
public class MiServlet extends HttpServlet { ... }
```

Como alternativa, podemos definir diferentes restricciones para cada método HTTP. Para ello utilizaremos la anotación `@HttpMethodConstraint`, que podrá tomar los siguientes parámetros:

* `value`: Indica el método para el que vamos a definir las restricciones de acceso. Por ejemplo `"GET"`, `"POST"`, etc.
* `rolesAllowed`: Define la lista de roles que pueden acceder al servlet mediante el método indicado.
* `transportGuarantee`: Indica si sólo se permite acceder al método indicado mediante SSL. Se define de la misma forma que en el caso de `@HttpConstraint`.
* `emptyRoleSemantic`: Nos permite establecer la política de acceso a llevar a cabo independientemente del rol del usuario para el método especificado. Se define de la misma forma que en el caso de `@HttpConstraint`.

Si sólo se especifica el método (sin añadir más parámetros), se considera que siempre se permite el acceso mediante dicho método. Por ejemplo, podemos definir políticas diferentes para los métodos `GET`, `POST` y `PUT`:

```
@WebServlet("/MiServlet")
@ServletSecurity(httpMethodConstraints={
   @HttpMethodConstraint("GET"),
   @HttpMethodConstraint(value="POST",rolesAllowed="admin"),
   @HttpMethodConstraint(value="PUT",
                         emptyRoleSemantic=EmptyRoleSemantic.DENY)})
public class MiServlet extends HttpServlet { ... }
```

En este ejemplo, se permite acceder a todos los usuarios con `GET`, pero con `POST` sólo podrán acceder los que tengan rol `admin`, y con `PUT` no podrá acceder nadie.

También podemos combinar una política particular para una serie de métodos, y una política general para el resto:

```
@ServletSecurity(
   value=@HttpConstraint(EmptyRoleSemantic.PERMIT),
   httpMethodConstraints={
     @HttpMethodConstraint(value="POST",rolesAllowed="admin")
     @HttpMethodConstraint(value="PUT",
         transportGuarantee=TransportGuarantee.CONFIDENTIAL)})
public class MiServlet extends HttpServlet { ... }
```

En este caso sólo los usuarios con rol `admin` podrán acceder mediante `POST`, y sólo se podrán establecer conexiones `PUT` mediante SSL, pero para el resto de métodos se permitirá acceder a cualquier usuario sin necesidad de utilizar SSL.

6.5. Acceso a la información de seguridad

Es muy probable que necesitemos acceder desde el código de la aplicación a información del contexto de seguridad del servidor, como puede ser el nombre del usuario autentificado actualmente, o sus roles. Podemos acceder a esta información a través del objeto `HttpServletRequest`.

En primer lugar, podemos obtener los datos del usuario autentificado actualmente con el método `getUserPrincipal()` de la petición. Esto nos devolverá un objeto de tipo `Principal`, del que podremos sacar el nombre del usuario. De forma alternativa, este nombre también se puede obtener mediante el método `getRemoteUser()` del mismo objeto.

```
Principal p = request.getUserPrincipal();
if(p!=null) {
    out.println("El usuario autentificado es " + p.getName());
} else {
    out.println("No hay ningun usuario autentificado");
}
```

También puede interesarnos comprobar si el usuario actual pertenece a un determinado rol, para así saber si debemos darle permiso o no para realizar una operación dada. Esto lo podemos realizar con el método `isUserInRole(rol)` del objeto petición.

```
if(request.isUserInRole("admin")) {
    usuarioDao.altaUsuario(usuario);
} else {
    out.println("Solo los administradores pueden realizar altas");
}
```

Otros métodos que nos aportan información sobre el contexto de seguridad son `getAuthType()`, que nos dice el tipo de autentificación que estamos utilizando (`BASIC`, `DIGEST`, `FORM`, `CLIENT-CERT`), y `isSecure()` que nos indica si estamos realizando una conexión segura (SSL) o no.

Para finalizar, si estamos utilizando seguridad basada en formulario podremos cerrar la sesión de forma sencilla llamando al método `invalidate()` del objeto `HttpSession`.

6.6. Ejercicios

Seguridad básica (0.3 puntos)

En las plantillas de la sesión tenemos un proyecto llamado `cweb-seguridad`, en el que hay un directorio `restringido` que deberemos proteger para que sólo los usuarios autentificados puedan acceder a su contenido. Se pide:

a) Crear en el *realm* `ApplicationRealm` los roles `admin` y `registrado`, y un usuario que tenga esos roles.

b) Añadir al fichero `web.xml` la configuración necesaria para proteger el directorio `restringido` y todo su contenido mediante autentificación de tipo `BASIC` para que sólo los dos roles que hemos creado puedan acceder.

c) Entrar en `restringido/index.jsp` para comprobar que está protegido, y que introduciendo el usuario que hemos creado nos deja acceder correctamente. Podemos encontrar una enlace a dicho recurso protegido desde la página principal `index.html`.

Seguridad basada en formularios (0.4 puntos)

Vamos a cambiar el tipo de seguridad por autentificación basada en formularios. Se pide:

a) Crear los ficheros `login.jsp` y `error.jsp` (puedes utilizar el código de los apuntes).

b) Modificar `web.xml` para cambiar el tipo de seguridad por autentificación basada en formularios, que utilice las páginas creadas en el apartado anterior.

c) Probar a acceder ahora al área restringida y comprobar que los formularios funcionan correctamente. Ahora podemos aprovechar la página `logout.jsp` de la plantilla para cerrar la sesión y así poder probar otro usuario.

Seguridad mediante anotaciones (0.3 puntos)

En el proyecto tenemos un servlet de nombre `SeguridadServlet`. Vamos a añadir anotaciones de seguridad para protegerlo. Se pide:

a) Añadir la anotación necesaria para que sólo los roles `registrado` y `admin` puedan acceder al servlet. Comprobar que las anotaciones funcionan correctamente.

b) Comprobar dentro del servlet si el usuario es administrador, y en tal caso mostrar un mensaje en la página que lo indique.

7. Filtros y Wrappers

Hasta ahora hemos visto la forma en la que los servlets nos permiten encapsular el mecanismo de petición/respuesta. Se identifica al servlet como un recurso dentro del sitio web, y cuando desde el cliente solicitamos dicho recurso, se ejecutará el código que hayamos definido dentro del método de servicio del servlet.

La limitación de los servlets es justamente esa, que un servlet se invocará sólo cuando solicitemos dicho servlet desde el cliente. Pero, ¿y si queremos procesar cualquier petición que se haga a cierta parte o toda nuestra aplicación web?

Si sólo contamos con servlets, para solucionar esto podríamos optar por alguna de las siguientes opciones por ejemplo:

- Crear un servlet central que invocaremos siempre desde el cliente pasándole como parámetro el recurso que deseamos obtener. Tiene el inconveniente de que no es transparente para desarrollador del contenido de nuestra aplicación web, ya que deberá definir todos los enlaces del sitio para que vayan al servlet.
- Introducir al comienzo de todos los servlets de nuestra aplicación una llamada a cierta función que haga el procesamiento que queremos realizar. Esto no nos serviría para el contenido estático de la aplicación web. Además tampoco es transparente ya que el desarrollador de los servlets deberá realizar una llamada a este código. Otro inconveniente es que estaremos repitiendo código común en varios elementos, lo cual va en contra de la modularidad.
- Configurar el servidor web (si nos lo permite) para que cualquier petición de recurso sea redirigida a un servlet que la procese. Esta sería la solución más apropiada, pero tendremos el problema de que si el servlet internamente quiere hacer la petición del recurso al servidor, volverá a redireccionarlo a si mismo, por lo que podemos entrar en un bucle infinito. Por lo tanto, surgirán problemas con la identificación de los recursos.

Como vemos, por ahora este problema no tiene ninguna solución totalmente satisfactoria. Para ello, a partir de la versión 2.3 de servlets, aparecen los denominados filtros.

7.1. Filtros

¿Qué es un filtro?

Un filtro es un componente que intercepta cualquier petición que se realice a un determinado grupo de recursos de nuestra aplicación web, y la respuesta que se vaya a devolver al cliente por parte del servidor.

Normalmente los filtros no generarán por si mismos la respuesta, como es el caso de los servlets, sino que simplemente la modificarán si es necesario. Podrán modificar tanto la petición HTTP, como la respuesta o las cabeceras de la misma.

Una ventaja importante de los filtros es que nos ayudarán a modularizar la aplicación, ya que son componentes independientes que actuarán sobre cualquier grupo de recursos, no teniendo dichos recursos porque conocer la existencia de estos filtros. De esta forma este filtrado de las peticiones y respuestas a nuestro servidor se realiza de un forma totalmente transparente en todos los niveles, tanto para el cliente como para los desarrolladores del contenido del sitio web (servlets, JSPs, páginas estática, y cualquier otro recurso).

Esta independencia implica por lo tanto que los filtros podrán ser reutilizados para cualquier elemento del sitio web, sin necesidad de incluir código común en todos los elementos que queramos que realicen dicha funcionalidad.

Funcionalidades de los filtros

Un filtro podrá acceder a la petición de un determinado recurso antes de que dicho recurso sea invocado, momento en el que podremos procesar o modificar dicha petición.

Una vez se ha invocado la petición, podremos procesar o modificar la respuesta que nos ha devuelto el servidor.

Además, podremos tener múltiples filtros actuando sobre determinados grupos de recursos. De esta forma un recurso podrá no ser filtrado, o ser filtrado por uno o más filtros. Cuando tenemos varios filtros, se organizarán en forma de cadena en el orden que nosotros especifiquemos, y cada uno procesará el resultado del anterior.

Aplicaciones de los filtros

Hemos descrito lo que es un filtro, pero entenderemos más claramente los filtros si vemos una serie de posibles aplicaciones que les podemos dar:

- Autentificación de usuarios: Podemos definir un filtro que actúe sobre cierta zona restringida de nuestra aplicación web. Si el usuario está registrado el filtro dejará ver el contenido tal cual, si no le redirigirá a la página con el formulario de registro de usuarios.
- Transformación con hojas XSL-T: Si tenemos una serie de páginas escritas en XML, y una serie de hojas de transformación XSL-T para generar código para distintos navegadores, podremos definir un filtro que actúe sobre el conjunto de documentos XML, y aplique una transformación según el tipo de navegador que hizo la petición. Devolverá al cliente la respuesta transformada, adaptada al navegador adecuado.
- Transformación de imágenes: Igual que transformamos documentos XML, también podemos aplicar los filtros a determinados formatos de imágenes, y transformar dichas imágenes dinámicamente a un formato más adecuado.
- Encriptación de datos: Podemos utilizar un filtro para que encripte la salida de cualquier recurso al que se acceda. El cliente deberá ser capaz de desencriptarlo para poder visualizar dicho contenido.
- Compresión de datos: De forma similar al punto anterior, podemos comprimir los datos que genera el servidor.
- Registro de acceso a recursos: Se puede contabilizar mediante un filtro la cantidad de accesos a cada recurso de nuestra web. Como todas las peticiones pasan a través de él, simplemente tendrá que incrementar la cantidad de visitas al recurso que se solicite en cada momento.
- Log de accesos: Podemos también elaborar un fichero de log de accesos a la web, para conocer los datos de todos los accesos que se han realizado.

Configuración de un filtro

Para que un filtro intercepte las peticiones a determinados recursos, deberemos configurar la aplicación web para que esto sea así. La forma de configurar los filtros es similar a la configuración de los servlets.

Los filtros, al igual que los servlets, serán clases Java que definamos, y que tendremos normalmente en el directorio `WEB-INF/classes` de nuestra aplicación web, o subdirectorios

de este si está en algún subpaquete. La configuración de los filtros deberá establecerse en el fichero de configuración de nuestra aplicación web, `WEB-INF/web.xml`.

Primero deberemos declarar los filtros incluidos en nuestra aplicación web. Para ello se utilizará la anotación `@WebFilter`:

```
@WebFilter("/ruta/*")
public class FiltroEjemplo implements Filter {
    ...
}
```

Esto siempre que estemos utilizando la API de Servlet 3.0. Si trabajásemos con una versión anterior, debereos utilizar el elemento `filter` que se define de en el descriptor de despliegue `web.xml`:

```
<filter>
  <filter-name>Filtro de ejemplo</filter-name>
  <filter-class>FiltroEjemplo</filter-class>
  <init-param>
    <param-name>fichero_log</param-name>
    <param-value>log.txt</param-name>
  </init-param>
</filter>
```

Es muy similar a la forma de declarar un servlet. Asignamos un nombre al filtro, que será asociado a la clase en la que está implementado dicho filtro. En este caso la clase es `FiltroEjemplo`, por lo que tendremos que tener el fichero `FiltroEjemplo.class` en el directorio `WEB-INF/classes` de nuestra aplicación.

A continuación podemos declarar una serie de parámetros de entrada para el filtro, de forma que para variar estos datos no tengamos que modificar y recompilar la clase del filtro, sino que simplemente deberemos modificar el valor del parámetro en este fichero de configuración. Podremos no tener ningún parámetro, tener uno, o tantos como queramos.

Una vez declarados los filtros deberemos mapearlos a los recursos. Las peticiones que se hagan al servidor a estos recursos, serán interceptadas por nuestro filtro. Podemos mapear filtros a recursos de distintas formas, con la etiqueta `filter-mapping`:

```
<filter-mapping>
  <filter-name>Filtro de ejemplo</filter-name>
  <servlet-name>Servlet interceptado</servlet-name>
</filter-mapping>

<filter-mapping>
  <filter-name>Filtro de ejemplo</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

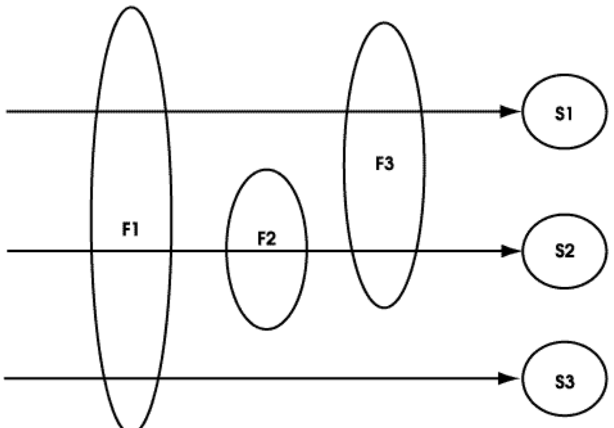
La primera forma nos sirve para mapearlo a un servlet, dado el nombre del servlet al que lo vamos a asociar. La segunda forma asocia el filtro a todos los elementos cuya URL cumpla el patrón dado:

```
/*                   Se asocia con todos los elementos
                     de nuestra aplicación web.

/zona_restringida/* Se asocia con todos los elementos
                     en el directorio de nombre
                     zona_restringida, y con los de sus
                     subdirectorios.
/web/*               Se asocia con todos los elementos
                     en el directorio de nombre
                     web, y con los de sus subdirectorios.
...
```

El valor por defecto de la anotación `@WebFilter` (`value`) se refiere a los patrones de las URLs sobre las que se mapea el filtro (es equivalente a su parámetro `urlPatterns`). Si queremos mapear servlets tenemos el parámetro `servletNames`

```
@WebFilter(servletName={ "MiServlet1", "MiServlet2", "MiServlet3" })
```

Podemos asociar varios filtros a un mismo recurso, si dicho recurso aparece mapeado para varios filtros. En este caso tendremos una cadena de varios filtros cuando se produzca una petición a este recurso.



Implementación básica de un filtro

Los filtros se definen mediante la interfaz `Filter`, contenida en el paquete `javax.servlet`. Por lo tanto, para crear un filtro deberemos crear una clase que implemente dicha interfaz:

```
import javax.servlet.*;
import javax.servlet.http.*;

class MiFiltro implements Filter {
  FilterConfig config;
```

Dentro de este clase, el método básico que deberemos implementar será el método `doFilter`, al que se llamará cada vez que dicho filtro intercepte una petición a recursos:

```
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
      ...
```

Vemos que a este método se le pasa como parámetro la petición y la respuesta, de forma que podamos procesarlas o modificarlas según la funcionalidad que queramos que implemente el filtro. Hemos de fijarnos que toma una petición y respuesta genérica, no se limita únicamente a peticiones y respuestas HTTP.

Además también se nos proporciona un objeto que representa la cadena de filtros. Con él podremos pasar la petición y la respuesta interceptadas al siguiente filtro de la cadena, o bien al recurso destino en caso de que ya no hubiese más filtros. Esto lo haremos con una llamada a:

```
      ...
      chain.doFilter(request, response);
      ... // En este punto el servidor ya habrá producido
          //la respuesta en response

    }
```

Justo después de haber llamado a este método, ya se habrá producido la respuesta, ya que con él estamos indicando que se ejecuten todos los filtros que siguen al nuestro en la cadena, y en último lugar el recurso solicitado.

Por lo tanto, todas las modificaciones que queramos hacer en la petición que va a llegar al recurso las deberemos hacer antes de la llamada a este método, mientras que todo procesamiento que queramos hacer de la respuesta se hará después de esta llamada, que será cuando se haya generado.

También podemos hacer que no se llegue a llamar, si queremos que nuestro filtro de la respuesta por si solo, sin acceder al recurso que se había pedido. Esto lo haremos por ejemplo cuando queramos prohibir el acceso a un recurso.

Otros métodos que debemos definir en un filtro son:

```
    public void init(FilterConfig config) throws ServletException {
      // Código de inicialización del filtro
      this.config = config;
      ...

    }

    public void destroy() {
      // Libera recursos del filtro
      config = null;
      ...
    }

    ...

  }
```

Que serán llamados en la inicialización y en la destrucción de este componente respectivamente.

Acceso al contexto

Acabamos de ver que cuando se inicializa el filtro se llama a su método `init`. En esta llamada se proporciona un objeto `FilterConfig` que contiene información sobre los parámetros del filtro, que vimos en el apartado de configuración, y además nos permite acceder a la información global de contexto.

Para leer los parámetros del filtro especificados en el descriptor de despliegue de la aplicación web (fichero `web.xml` en Tomcat como hemos visto), este objeto proporciona el siguiente método:

```
String valor = config.getInitParameter(nombre_param);
```

Esta llamada nos devolverá una cadena con el valor del parámetro, o null en el caso de que el parámetro indicado no existiese. Si queremos obtener la lista de parámetros definidos en el descriptor de despliegue, podemos usar el siguiente método:

```
Enumeration parametros = config.getInitParameterNames();
```

Con esto obtendremos una enumeración de todos los nombres de parámetros definidos.

Este objeto también nos permite obtener el nombre del filtro, que se habrá definido en el descriptor de despliegue, con el método:

```
String nombre = config.getFilterName();
```

Este objeto además nos permitirá acceder al objeto de contexto global del contenedor de servlets, mediante el método:

```
ServletContext context = config.getServletContext();
```

Obtenemos este objeto con el cual podremos acceder a los atributos globales definidos en nuestra aplicación web, y además nos proporciona una serie de métodos que nos permitirán realizar en filtros las mismas operaciones que podíamos hacer en los servlets.

Será importante acceder a este objeto desde los filtros, ya que si queremos realizar redirecciones, o acceso a recursos estáticos por ejemplo, necesitaremos contar con dicho objeto.

Ciclo de vida de un filtro

Justo después del despliegue de la aplicación web, y antes de que se produzca cualquier petición a un recurso, el contenedor localizará los filtros que deben ser aplicados a cada recurso. Instanciará los filtros que hayamos declarado, y tras ello llamará al método `init` de cada filtro para inicializarlo.

Si hacemos que este método `init` lance una excepción `UnavailableException` estaremos indicando que el filtro no puede funcionar correctamente. Esta excepción tiene un

método `isPermament` que indicará si el fallo es permanente o puede recuperarse pasado un tiempo. De no ser permanente el contenedor intentará volver a instanciar el filtro más adelante. Podemos establecer en la excepción un tiempo estimado que puede tardar en estar disponible, para informar al contenedor de cuando puede volver a intentar instanciarlo.

Al método `init` se le proporcionará el objeto `FilterConfig`, con la información de los parámetros y nombre del filtro obtenidos del descriptor de despliegue, además de una referencia al objeto `ServletContext` de la aplicación web, como hemos visto en el apartado anterior.

Una vez terminada la fase de inicialización, el servidor ya podrá empezar a recibir peticiones. Cuando se produzca una petición, el contenedor localizará el primer filtro asociado a dicho recurso, y llamará a su método `doFilter` proporcionando los objetos `ServletRequest`, `ServletResponse`, y `FilterChain`. Una vez hecho esto será responsabilidad de nuestro filtro tratar estos objetos, y decidir si pasar el procesamiento al siguiente filtro de la cadena.

Cuando lleguemos al ultimo filtro de la cadena, al llamar a `doChain` se invocará directamente el recurso que se solicitaba en la petición.

Si durante `doFilter` lanzamos una excepción `UnavailableException`, el contenedor no intentará seguir procesando la cadena de filtros. Si hemos indicado que es no permanente, tras un rato reintentará procesar la cadena entera.

Antes de poder hacer que el filtro deje de estar en servicio, llamará a su método `destroy` para que libere los recursos que sea necesario.

7.2. Wrappers

Hasta ahora hemos visto como interceptar la petición que se realiza a un determinado recurso de nuestra web mediante filtros, pero, ¿y si queremos interceptar la respuesta que nos devuelve el servidor para analizarla o modificarla?

Cuando desde nuestro filtro pasemos el procesamiento de la petición al siguiente elemento de la cadena (`doFilter`), delegaremos en este siguiente elemento el procesamiento de la petición y la generación de la respuesta. Supongamos que este elemento es el recurso final que se había solicitado. En este caso el contenido de este recurso será escrito en el objeto respuesta, lo cual producirá que dicho contenido sea devuelto al cliente.

Sin embargo, nosotros no queremos que sea devuelto directamente al cliente, sino que queremos procesarla previamente en nuestro filtro antes de devolverla. Con este objeto `ServletResponse` (`HttpServletResponse`) no podremos hacer esto, ya que cuando se escribe en él lo que se hace es devolver la respuesta al cliente, y una vez escrita no podemos acceder nuevamente a ella ni modificarla.

La solución a nuestro problema es sustituir el objeto respuesta que proporcionamos al siguiente elemento de la cadena por un objeto de respuesta creado por nosotros.

¿Qué es un wrapper?

Un *wrapper* es un objeto que envuelve al objeto original, de forma que no se acceda directamente al objeto original sino al *wrapper*. El *wrapper* implementará la misma interfaz del objeto al que envuelve, de forma que externamente se trabajará con él de la misma forma, por lo que podemos sustituir el original por el *wrapper* siendo esto transparente a los sucesivos elementos que vayan a manipular este objeto.

Cuando se llame a un método del *wrapper* podrá, o bien redirigir la llamada al correspondiente método del objeto original al que envuelve, o bien tratar por si mismo la llamada a dicho método. De esta forma, podremos redefinir el comportamiento que tendrán determinadas operaciones.

Encontramos para nuestro fin *wrappers* para la petición y la respuesta: `ServletRequestWrapper` (`HttpServletRequestWrapper`) y `ServletResponseWrapper` (`HttpServletResponseWrapper`). Con ellos podremos crear implementaciones propias del objeto petición y respuesta que envuelvan a los originales, pudiendo de esta forma redefinir el comportamiento de determinadas operaciones.

Nos centraremos en el *wrapper* de la respuesta. Con él podemos evitar que la respuesta se envie directamente al cliente. En lugar de esto, cuando se escriba la salida en este objeto *wrapper* de la respuesta podemos hacer que guarde dicha salida en un buffer interno. Una vez procesados todos los elementos de la cadena que están después de nuestro filtro (tras llamar a `doFilter`), se habrá escrito la salida generada en el *buffer* del *wrapper*. En este momento podemos analizar esta salida, modificarla si es necesario, y enviarla a través del objeto respuesta original.

Implementación de un wrapper

Para implementar un wrapper deberemos crearnos una subclase de la clase del *wrapper* adecuado para nuestro caso (petición o respuesta), y redefinir en esta subclase las operaciones cuyo comportamiento queramos cambiar. El funcionamiento por defecto de las operaciones que no redefinamos será redirigir la petición al método correspondiente del objeto (petición o respuesta) original.

Vamos a ver esto con un ejemplo de implementación de un wrapper de la respuesta que guarda en un buffer la respuesta generada por el servidor, para poder ser procesada por nuestro filtro.

Puesto que queremos envolver la respuesta, tendremos que crearnos una subclase de `ServletResponseWrapper`:

```
public class GenericResponseWrapper extends HttpServletResponseWrapper {
```

Dentro de esta clase deberemos tener el buffer donde vayamos a escribir la salida. Dado que en la salida se puede escribir tanto como flujo de bytes como de caractéres, para que sea más genérico convendrá crear el buffer como array de bytes, de forma que se pueda escribir en él de las dos formas:

```
  private ByteArrayOutputStream output;
```

En el constructor de la clase simplemente deberemos proporcionar el objeto respuesta original (al cual estaremos envolviendo). Lo que hacemos aquí es utilizar el constructor de la superclase proporcionándole la respuesta original, de forma que se encargue de redirigir a él las operaciones predeterminadas. Además deberemos crear nuestro buffer de bytes donde se escribirá la respuesta:

```
  public GenericResponseWrapper(HttpServletResponse response) {
    super(response);
    output = new ByteArrayOutputStream();
  }
```

Proporcionaremos además un método para obtener los datos escritos en el buffer:

```
   public byte[] getData() {
     return output.toByteArray();
   }
```

Cuando alguien quiera devolver una respuesta al cliente lo que hará será obtener el flujo de salida del objeto respuesta y escribir en él. Por defecto este flujo envia los datos al cliente. Sin embargo podemos evitar que esto ocurra haciendo que los flujos que devuelva sirvan para escribir en el buffer, y no para enviar la respuesta al cliente. Se puede enviar la respuesta de dos formas: mediante un flujo de bytes (`getOutputStream`), o mediante un flujo de carácteres (`getWriter`), por lo que deberemos redefinir ambos métodos.

```
   public ServletOutputStream getOutputStream() {
     return new FilterServletOutputStream(output);
   }

   public PrintWriter getWriter() {
     return new PrintWriter(getOutputStream(), true);
   }
 }
```

En el caso del flujo de bytes, deberemos devolverlo como un `ServletOutputStream` . Por lo tanto tendremos que crearnos un tipo propio de `ServletOutputStream` que escriba en nuestro buffer:

```
 public class FilterServletOutputStream extends ServletOutputStream {
     private DataOutputStream stream;

     public FilterServletOutputStream(OutputStream output) {
         stream = new DataOutputStream(output);
     }

     public void write(int b) throws IOException {
         stream.write(b);
     }

     public void write(byte[] b) throws IOException {
         stream.write(b);
     }

     public void write(byte[] b, int off, int len) throws IOException {
         stream.write(b, off, len);
     }
 }
```

Este será el flujo que utilicemos para escribir la respuesta en forma de bytes en nuestro buffer interno.

Aunque a primera vista parezca compleja la creación de dicho *wrapper*, tiene la ventaja de ser reutilizable para cualquier aplicación en la que necesitemos interceptar la respuesta generada por el servidor.

Utilización de un wrapper

Para utilizar el *wrapper* que hemos creado, deberemos instanciarlo a partir del objeto de respuesta original que le ha sido proporcionado a nuestro filtro. Esto lo haremos antes de que se haya generado el contenido del recurso solicitado, es decir, antes de llamar a `doFilter`.

```
public void doFilter(ServletRequest request,
      ServletResponse response, FilterChain chain)
{
  ...
  GenericResponseWrapper wrapper =
      new GenericReponseWrapper(response);
```

Una vez hemos creado nuestro propio objeto respuesta que envuelve a la respuesta original, podemos utilizarlo para que el servidor escriba el contenido del recurso solicitado en él. Para esto realizaremos la llamada a `doFilter` proporcionando como respuesta este *wrapper* que hemos creado:

```
  chain.doFilter(request, wrapper);
```

Una vez ejecutado este método se habrá generado la respuesta en el objeto de respuesta proporcionado, en este caso habrá sido en nuestro *wrapper*. Por lo tanto podemos obtener y procesar la respuesta según la función de nuestro filtro:

```
  byte [] datos = wrapper.getData();

  ... // Procesar datos segun la funcion del filtro
```

Por último, para que el cliente pueda ver esta respuesta, deberemos escribirla en el objeto respuesta original:

```
  OutputStream out = response.getOutputStream();
  out.write(datos);
  out.close();
}
```

Con esto vemos que habremos podido procesar la salida generada en nuestro filtro, y enviarla al cliente para que pueda ser visualizada correctamente.

7.3. Ejemplos

Vamos a ver a continuación una serie de ejemplos de usos comunes de los filtros, y cómo implementaríamos dichos filtros, utilizando distintos elementos que hemos visto durante el curso.

Acceso restringido

Una primera aplicación sencilla de los filtros es prohibir el acceso a cierta parte de nuestra web. Cuando un usuario intente acceder a dicha parte, se comprobará si este usuario está

registrado. Si lo está se le dejará pasar normalmente, pero si no se prohibirá el acceso, redireccionando a la página de login de usuarios.

```
public class RestringirAcceso implements Filter {
```

Cuando se invoca el filtro querrá decir que un usuario intenta acceder a la zona restringida.

```
    public void doFilter(ServletRequest request,
      ServletResponse response, FilterChain chain)
    {
        // Se intenta acceder a la zona restringida
```

Comprobamos si el usuario está registrado en el servidor. Para ello utilizamos la información de sesión, donde almacenaremos el login del usuario en caso de estar registrado.

```
        // Solo podemos comprobar la sesión
        // en el caso de tener una petición HTTP

        if(request instanceof HttpServletRequest &&
            response instanceof HttpServletResponse)
        {

            HttpServletRequest http_request =
                       (HttpServletRequest)request;
            HttpServletResponse http_response =
                       (HttpServletResponse)response;

            // * Comprobamos si el usuario se ha registrado *
            // En nuestra aplicación si el usuario
            // se ha registrado habremos establecido
            // el atributo usuario de la sesion al
            // nombre del usuario, si no será null.

            if(http_request.getSession().getAttribute("usuario")!=null)
```

Si hay un login almacenado, procesamos la petición de forma normal.

```
            {
                // Continuamos de forma normal con la petición
                chain.doFilter(request, response);
            }
```

Si no, redireccionamos a la página de login, para que el usuario se registre.

```
            else
            {
                // Redireccionamos a la página de login
                http_response.sendRedirect("/ejemplo/login.jsp");
            }
        } else {
            // Si no es una petición HTTP
```

```
            // simplemente procesamos la petición

            chain.doFilter(request, response);
        }
    }
}
```

Ranking de páginas más visitadas

Otra posible aplicación es registrar el número de visitas que se hacen a cada página, de forma que podremos obtener un listado de las páginas favoritas de los usuarios dentro de nuestro sitio web. Para ello instalaremos un filtro que intercepte las peticiones a cualquier página. Cada vez que el filtro se invoque, querrá decir que se ha visitado una página. Lo que deberemos hacer en este momento es:

Determinar la dirección de la página que se ha solicitado

```
public class Ranking implements Filter {

    public void doFilter(ServletRequest request,
      ServletResponse response, FilterChain chain)
    {
        // Solo podemos ver el recurso solititado en el
        // caso de tener una petición HTTP

        if(request instanceof HttpServletRequest)
        {
            HttpServletRequest http_request =
                            (HttpServletRequest)request;

            // Miramos que recurso está siendo solicitado
            String uri = http_request.getRequestURI();
```

Tendremos una base de datos con una entrada para cada página, donde se contabilizan el número de visitas. Si no existe entrada para la página visitada, la crearemos con una visita.

```
            PaginasDAO dao = new PaginasDAO();

            if(dao.existePagina(uri)
            {
                // La página ya esta registrada en la BD
                // y solo tenemos que incrementar su contador
                dao.incrementaContador(uri);
            }
```

Si ya existe entrada para esta página en la BD, incrementaremos el número de visitas.

```
            else
            {
                // La página se está visitando por primera vez
                // Debemos registrarla en la BD
                // con contador a 1 (1 visita)
```

```
                dao.insertaPagina(uri);
            }
        }
```

Procesamos la petición de forma normal.

```
        chain.doFilter(request, response);
    }
```

Extracción automática de información

Imaginemos que en el ranking queremos, además de la dirección, registrar el título de la página. A partir de la información de la petición y la respuesta ordinaria no podemos obtener dicha información, ya que se refiere al contenido de la página. Para ello tendremos que utilizar un wrapper, que obtenga la respuesta generada por el servidor, de manera que podamos analizarla y extraer de ella el título de la página.

```
public class RankingTitulo implements Filter {

    public void doFilter(ServletRequest request,
      ServletResponse response, FilterChain chain)
    {
        // Solo podemos ver el recurso solititado en el caso
        // de tener una petición HTTP

        if(request instanceof HttpServletRequest &&
            response instanceof HttpServletResponse)
        {

            HttpServletRequest http_request =
                       (HttpServletRequest)request;
            HttpServletResponse http_response =
                       (HttpServletResponse)response;

            // Miramos que recurso está siendo solicitado
            String uri = http_request.getRequestURI();

            PaginasDAO dao = new PaginasDAO();

            if(dao.existePagina(uri)
            {
                // La página ya esta registrada en la BD
                // y solo tenemos que incrementar su contador
                dao.incrementaContador(uri);
            }
```

Cuando se visite una página por primera vez, para registrarla en la base de datos tendremos que obtener la información del título. Creamos un wrapper, y procesamos la petición utilizando dicho wrapper como objeto respuesta.

```
            else
```

```
            {
                // La página se está visitando por primera vez
                // Debemos obtener su titulo para registrarla en la BD

                // Envolvemos la respuesta con nuestro wrapper generico
                GenericResponseWrapper wrapper =
                       new GenericResponseWrapper(http_response);

                // Procesamos la petición
                chain.doFilter(http_request, wrapper);
```

Una vez hecho esto, tendremos en el wrapper el contenido de la página generado. Podemos obtenerlo y analizarlo, buscando en él la etiqueta `<title>`.

```
                // En este momento ya diponemos
                // de la respuesta en el wrapper

                // La analizamos para obtener el
                // valor de su etiqueta <title>

                byte [] datos = wrapper.getData();
                HtmlParser parser = new HtmlParser(datos);
                String titulo = parser.getTitle();
```

Una vez obtenido el título, podremos registrar en la base de datos la entrada de la página.

```
                // Ahora podemos registrar ya la página con sus datos
                dao.insertaPagina(uri, titulo);
```

Por último, tendremos que hacer que la respuesta del wrapper pase al cliente, enviándola al objeto respuesta original.

```
                // Por último, debemos devolver
                // la respuesta al cliente de forma
                // que pueda visualizar el recurso solicitado
                OutputStream out = response.getOutputStream();
                out.write(datos);
                out.close();
            }
        } else {
            // Si no es HTTP procesamos la petición de forma ordinaria
            chain.doFilter(request, response);
        }
    }

}
```

7.4. Ejercicios

Filtro de acceso restringido (0 puntos)

Vamos a probar la aplicación `cweb-filtros` en la que tenemos un filtro `RestringirAcceso` que prohibe el acceso de los usuarios no registrados al contenido del directorio `restringido`. Se pide:

a) Desplegar la aplicación en Tomcat. Una vez instalada comprobamos que la aplicación se ha instalado correctamente. Tras esto intentamos acceder a uno de los recursos del directorio `restringido`:

http://localhost:8080/cweb-filtros/restringido/index.html

http://localhost:8080/cweb-filtros/restringido/index.jsp

¿Qué ocurre? ¿Por qué?

b) Validarse como usuario y volver a intentar acceder al directorio `restringido`. ¿Ahora que ocurre? ¿Qué ventajas tiene implementar esta restricción de acceso mediante un filtro?

Restringir el acceso al chat (0.5 puntos)

Recordemos que en sesiones anteriores realizamos un chat mediante servlets, en el que había que registrarse como usuario antes de entrar a hablar. Sin embargo, si se introduce directamente la dirección:

http://localhost:8080/cweb-chat/chat/chatFrames.html

Podemos acceder directamente al chat sin estar registrados. Se pide:

a) Implementar un filtro en el chat que restrinja el acceso al chat si no se ha registrado un *nick* en la sesión.

b) ¿A qué recursos deberá afectar este filtro? Introducir la configuración necesaria para que el filtro intercepte los intentos de acceso a los recursos restringidos.

Wrapper de ejemplo (0 puntos)

La aplicación `cweb-wrapper` incorpora un filtro que utiliza un *wrapper* para analizar la respuesta generada. Toma esta respuesta del *wrapper* (asumimos que es contenido HTML), y la analiza utilizando la librería *htmlparser*, para extraer su título. Una vez tiene el título registra que se ha accedido a dicha página en el *log* y devuelve la respuesta al cliente. Se pide:

a) Desplegar la aplicación en Tomcat. Probar a acceder a varios ficheros HTML estáticos de los incluidos dentro de la aplicación. Comprobar que en el *log* se ha registrado el acceso indicando el título de las páginas.

b) Si tuviesemos recursos que no fuesen HTML (como por ejemplo imágenes) dentro de la ruta a la que afecta el filtro, ¿que ocurriría?. Intentar acceder a una imagen dentro de la aplicación y ver el error que se produce. ¿A qué se debe esto? ¿Como podríamos solucionarlo?

Registro de accesos (0.5 puntos)

Vamos a realizar una aplicación que contabilice el número de accesos a las páginas mediante un filtro. Esta aplicación tiene el nombre `cweb-ranking`. Para ello tendremos una lista de páginas en memoria en la que figurará:

`ruta`	Ruta de la página visitada
`titulo`	Título de la página
`accesos`	Número de accesos realizados

El servlet `RankingPaginasServlet` nos genera un listado de las páginas junto al número de visitas que han recibido.

Se pide:

a) Desarrollar el filtro `AccesoPaginaFilter` que actue sobre todos los recursos estáticos, y que contabilice el número de visitas que se realiza a ellos almacenando esta información en la lista de páginas. Deberá cumplir las siguientes características:

- Cuando una página sea visitada por primera vez, se deberá registrar una nueva en el `Map` de páginas en memoria, con la URL de la página visitada como clave y el contador de visitas inicializado a `1`. Por el momento no obtendremos el título de la página, ya que para esto se requiere utilizar un *wrapper*, así que en este campo introduciremos siempre el valor *(Título desconocido)*.
- Cuando la página ya estuviese registrada en la lista lo que haremos será incrementar el número de visitas en 1.

b) Actualizar el filtro `AccesoPaginaFilter` para que obtenga el título de la página a la que se ha accedido, utilizando para ello el *wrapper* genérico que se proporciona. Deberá cumplir las siguientes características:

- Sólo registrará los recursos cuyo contenido sea html (tipo de contenido `text/html`).
- Cuando una página sea visitada por primera vez, se deberá extraer su título de la etiqueta `<title>`, para lo cual deberá usarse un *wrapper*. En este caso insertaremos los datos de la página en la lista anotando una visita.
- Cuando la página ya estuviese registrada en la lista, lo que haremos será incrementar el número de visitas en 1.

c) Si la página HTML está almacenada en la caché del navegador, el acceso no se contabilizará correctamente. ¿Qué cabecera HTTP podriamos utilizar para solucionar este problema? ¿Donde podriamos establecer esta cabecera? Establecer las cabeceras necesarias para evitar el uso de la caché y comprobar el correcto funcionamiento de la aplicación.

Para asegurarnos de que funciona correctamente con los diferentes navegadores, podemos incluir una serie de cabeceras destinadas a evitar que la página se almacene en la caché:

```
Cache-control: no-cache
Cache-control: no-store
Pragma: no-cache
Expires: 0
```

8. Facelets, JSTL y lenguajes de expresiones

8.1. JavaServer Faces y Facelets

JavaServer Faces (JSF) es un *framework* para la creación de aplicaciones web en el lado del servidor. Dentro de este *framework* encontramos:

- Una API que nos proporciona **componentes** que podemos utilizar en la interfaz de nuestra aplicación y nos permite gestionar sus eventos y su validación en el lado del servidor. Ejemplos de componentes son los botones, cuadros de texto, imágenes, mensajes de texto, etc.
- Librerías de **etiquetas** con las que podemos añadir los componentes anteriores a una página web sin necesidad de introducir código Java. Por ejemplo, encontramos etiquetas como `<h:form>`, `<h:inputText>`, `<h:graphicImage>`, `<h:selectOneMenu>`, etc.

Una página JSF se representa mediante un árbol de componentes, al que se llama **vista**.

Los **componentes** que constituyen la vista son elementos reutilizables y configurables como por ejemplo botones, campos de texto, tablas, etc. Todos los componentes se implementan en clases que heredan de `UIComponentBase`. Estas clases implementan la funcionalidad del componente, pero no la forma de mostrarlo. Esta separación permite crear de forma sencilla varias formas de mostrar un componente reutilizando su funcionalidad. La forma de mostrar un componente se implementa mediante un objeto de tipo `Renderer`.

Podemos tener diferentes **etiquetas** para un mismo componente, de manera que cada una de ellas lo muestre de forma distinta. Por ejemplo, el componente `UISelectOne` tiene la funcionalidad de permitir seleccionar un único elemento de una lista, pero puede presentarse mediante tres *renderers* alternativos, por lo que tenemos disponibles tres etiquetas para este mismo componente:

- `<h:selectOneListbox>`: Se muestra como un cuadro de lista donde aparecen todas las opciones.
- `<h:selectOneMenu>`: Se muestra como un menú desplegable.
- `<h:selectOneRadio>`: Se muestra como botones de radio.

Cada etiqueta de la librería HTML de JSF relaciona un componente con un *renderer* determinado para mostrarlo.

Una ventaja de JSF es que separa claramente la presentación de la lógica. Existen diferentes opciones para crear la vista en JSF. Se podría utilizar cualquier tipo de componente en el que podamos incluir las etiquetas que proporciona, como son por ejemplo los JSPs, pero la opción recomendada es la tecnología de *Facelets*. Vamos a centrarnos en estudiar los *Facelets* y los distintos elementos que podemos incluir en ellos.

8.2. Introducción a los Facelets

Los *Facelets* consisten en páginas creadas en XHTML, dentro de las cuales podemos utilizar los siguientes elementos:

- **Librerías de tags de Facelets, JSF y JSTL**: Se utilizan fundamentalmente para crear los diferentes componentes de la interfaz de la página, además de especificar la forma de validarlos o de presentar los datos.

- **Lenguaje de expresiones (EL)**: Nos permite relacionar los componentes anteriores con datos de nuestra aplicación, como por ejemplo aquellos contenidos en *managed beans*, o aquellos que recibimos como parámetros de la petición. Podremos utilizar este lenguaje en el cuerpo de la página o en los atributos de las etiquetas anteriores para consultar o guardar datos en *managed beans*, realizar operaciones con ellos, etc.
- **Plantillas de componentes y páginas**: Podemos definir la estructura de contenido de nuestras páginas mediante una plantilla, y aplicar esta plantilla a todas ellas. Por ejemplo podremos definir en la plantilla bloques para la cabecera, pie, menú lateral, etc.

En los siguientes apartados estudiaremos con detalle cada uno de los componentes anteriores.

Para crear una aplicación que utilice *Facelets* deberemos realizar lo siguiente:

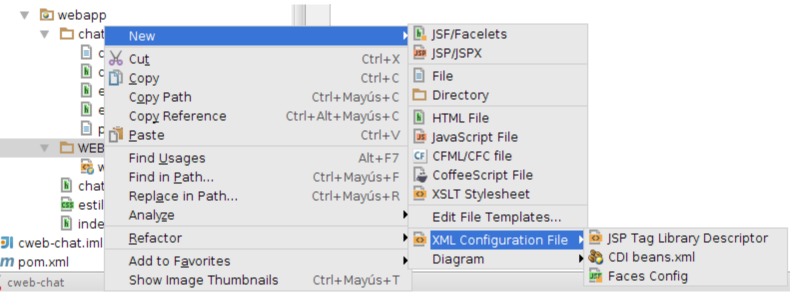
- Mapear el servet de JSF (`FacesServlet`) a un determinado patrón de URL (por ejemplo `*.xhtml`).
- Crear *managed beans*. Desde los *Facelets* accederemos a los datos de nuestra aplicación a través de ellos.
- Crear páginas utilizando las etiquetas de componentes. Los *Facelets* serán estas página XHTML que utilizarán dichas etiquetas y accederán a los datos mediante *managed beans*.

Mapeo del *servlet* de JSF

Para poder usar *Facelets* deberemos mapear el *servlet* de JSF a un determinado patrón de URL para que se encargue de procesar todas las páginas que se ajusten a dicho patrón. Normalmente quedará mapeado al patrón `*.xhtml` , para que todas las páginas que tengan dicha extensión sean procesadas como *Facelets*. De no añadir este mapeo se tratarían como páginas estáticas.

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

Esto puede hacerse de forma automática desde IntelliJ. Bastará con situarse sobre el directorio `WEB-INF` , pulsar con el botón derecho y seleccionar *New > XML Configuration File > Faces Config*. Con esto se creará en `WEB-INF` en fichero de configuración de JSF (`faces-config.xml`), y añadirá al descriptor de despliegue la configuración del *Faces Servlet* que se encargará de procesar los *Facelets*.



Estructura básica de un *Facelet*

A continuación mostramos la estructura básica que tiene un *Facelet*:

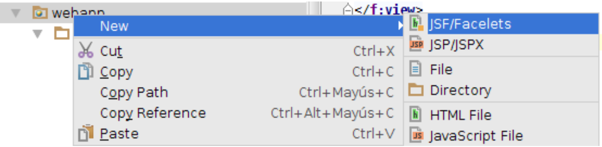
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"> ❶

    <h:head> ❷
        <title>Mi primer Facelet</title>
    </h:head>
    <h:body> ❸
        <!-- Contenido de la página -->
    </h:body>
</html>
```

❶ Declaración de las librerías de etiquetas a utilizar. En este caso declaramos únicamente la librería HTML de JSF asignándole el prefijo `h`.

❷ En lugar de `<head>` se suele utilizar una etiqueta equivalente de la librería HTML de JSP. Al haber sido declarada con prefijo `h`, dicha etiqueta será `<h:head>`.

❸ Al igual que `<head>`, para la etiqueta `<body>` también utilizamos la etiqueta equivalente de JSF.

Podemos crear un *Facelet* desde IntelliJ pulsando con el botón derecho sobre el directorio donde lo queramos añadir (por ejemplo `webapp`) y seleccionando *New* > *JSF/Facelets*.



Declaración de librerías de etiquetas

En el ejemplo anterior hemos declarado únicamente la librería HTML de JSF. Podríamos añadir a la declaración otras librerías de etiquetas como por ejemplo las que se muestran a continución:

Librería	URI	Prefijo	Descripción
JSF HTML Tag Library	http://xmlns.jcp.org/jsf/html	h	Componentes de la UI de la página
JSF Core Tag Library	http://xmlns.jcp.org/jsf/core	f	Funciones y acciones
JSTL Core Tag Library	http://xmlns.jcp.org/jsp/jstl/core	c	Etiquetas de propósito general de JSTL
JSTL Functions Tag Library	http://xmlns.jcp.org/jsp/jstl/functions	fn	Funciones de JSTL

Cada librería tiene un prefijo definido por convención que utilizaremos normalmente al declararla, pero podríamos cambiarlo. Por ejemplo

```
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">
```

En este caso hemos declarado la librería HTML de JSF con prefijo `h` y la librería Core de JSTL con prefijo `c`. Podríamos también declararlas de la siguiente forma:

```
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:html="http://xmlns.jcp.org/jsf/html"
      xmlns:core="http://xmlns.jcp.org/jsp/jstl/core">
```

En este último caso las etiquetas de la librería HTML de JSF se escribirían con prefijo `html`, como por ejemplo `<html:head>` y `<html:body>`. De la misma forma, las etiquetas de la librería Core de JSTL se escribirían con prefijo `core`.

8.3. Lenguaje de expresiones

Para poder conectar las etiquetas de un *Facelet* con los datos de nuestra aplicación necesitaremos introducir en ellas expresiones que hagan referencia a los *managed beans* que hayamos definido. Estas expresiones se definen mediante el llamado lenguaje de expresiones (EL).

Introducción al lenguaje de expresiones

A partir de la versión 2.0 de JSP se introdujo en las páginas un lenguaje de expresiones que permite hacer referencia a objetos de la aplicación sin necesidad de introducir código Java. En realidad, dicho lenguaje se implantó con las primeras versiones de JSTL, una librería de etiquetas JSP que se considera estándar, y que permitía utilizar este lenguaje de expresiones en los atributos de dichas etiquetas, constituyendo una característica muy importante de JSTL.

Cualquier elemento que pertenezca al lenguaje de expresiones irá englobado dentro de la marca `${…}`. En ella podremos colocar nombres de *managed beans*, parámetros de petición HTTP, elementos de una sesión… etc.

Por ejemplo, si previamente hemos creado un *managed bean* `miBean`, podríamos acceder a él desde la página (JSP o *Facelet*) con algo como:

```
<h3>El valor del bean es ${miBean}</h3>
```

Si lo que queremos es mostrar el parámetro `password` que hemos tomado de un formulario, para sacarlo por pantalla o guardarlo en alguna base de datos, podríamos acceder a él con algo como:

```
Accediendo al parámetro ${param.password}
```

También se pueden utilizar, como veremos, expresiones más complejas, que se evalúan desde el contenedor de *Facelets* o JSP. Por ejemplo, si tenemos un *bean* `edad` para una persona y queremos comprobar si dicha persona es mayor de edad, podríamos poner:

```
${edad > 18}
```

Y luego utilizar el resultado de esta expresión en otras zonas (por ejemplo, etiquetas condicionales de JSTL) para realizar la acción correspondiente.

Se describe a continuación, y a grandes rasgos, el lenguaje de expresiones incluido en JSTL 1.0, y en general a partir de JSP 2.0. El lenguaje está inspirado en los lenguajes ECMAScript y XPath, y está basado en espacios de nombres (atributos *PageContext*), propiedades de elementos, operadores relacionales, lógicos y aritméticos, y un conjunto de objetos implícitos.

Atributos y expresiones

Como hemos comentado anteriormente, podremos invocar a este lenguaje desde cualquier lugar de nuestra página (en JSP 2.0 o *Facelets*), o dentro de un atributo de una etiqueta, mediante el elemento `${…}`:

```
${expresion}
```

Esta expresión podrá estar:

- Por sí sola dentro de un atributo de una etiqueta (por ejemplo de JSTL):

  ```
  <c:set var="miVariable" value="${expresion}"/>
  ```

  En este caso, se evalúa la expresión y el resultado se convierte al tipo de dato del atributo, siguiendo las reglas de conversión internas del lenguaje.

- Combinada con texto dentro de un atributo de una etiqueta (por ejemplo de JSTL):

  ```
  <c:set var="miVariable" value="texto${e1} y ${e2}texto"/>
  ```

  Aquí, las expresiones se evalúan de izquierda a derecha, y se intercalan entre el texto, convirtiéndolas a *String* (siguiendo reglas de conversión internas). Luego, la cadena resultante se convierte al tipo del atributo en el que esté (si está dentro de algún atributo).

- Fuera de atributos, dentro del contenido HTML de la página JSP:

  ```
  <h3>Hola, esto es una página</h3>
  <p>Y aquí ponemos una expresión ${expresion}, para mostrar su valor</p>
  ```

Para cadenas que contengan la secuencia '${' sin que sea propiamente una expresión, se encapsula esa secuencia así: `${'${'}`. Por ejemplo:

```
Cadena con ${'${'}expr}
```

Mostraría: "*Cadena con ${expr}*"

Operadores

- **Operadores `[ ]` y `.`**: se unifican los operadores `[ ]` y `.` de forma que son equivalentes:

```
${expr.campo}
${expr["campo"]}
```

- **Operadores aritméticos**:
- `+`, `-`, `*`, `/`: suma, resta, multiplicación y división
- `div`: división entera
- `%`, `mod`: resto (se mantienen los dos por compatibilidad con XPath y ECMAScript)
- `-`: cambio de signo
- **Operadores relacionales**:
- `>`, `gt`: mayor que
- `<`, `lt`: menor que
- `>=`, `ge`: mayor o igual que
- `#`, `le`: menor o igual que
- `==`, `eq`: igual que
- `!=`, `ne`: distinto que
- **Operadores lógicos**:
- `&&`, `and`: Y lógica
- `||`, `or`: O lógica
- `!`, `not`: NO lógica
- **Operador** `empty` utilizado delante de un elemento, para indicar si el elemento es nulo o vacío (devolvería `true`) o no (devolvería `false`). Por ejemplo:

```
${empty A}
```

PRECEDENCIA

- `[ ]`, `.`
- `( )`
- `-` (cambio de signo), `not`, `!`, `empty`
- `*`, `/`, `div`, `%`, `mod`
- `+`, `-`
- `<`, `>`, `#`, `>=`, `lt`, `gt`, `le`, `ge`
- `==`, `!=`, `eq`, `ne`
- `&&`, `and`

- `||`, `or`

EJEMPLOS

```
// Daría true si el parametro nombre no se ha enviado
${empty param.nombre}

// Devolvería el resultado de la suma de ambas variables
${num1 + num2}

// Devolvería true si valor1 es mayor o igual que valor2
${valor1 >= valor2}

// Daría true si v1 fuese distinto a v2, y v3 menor que v4
${v1 ne v2 and v3 < v4}
```

Nombres de variables

El lenguaje de expresiones evalúa un identificador o nombre de elemento mirando su valor como un atributo, según el comportamiento del método `PageContext.findAttribute(String)`. Por ejemplo, si ponemos:

```
${valor}
```

Se buscará el atributo *valor* en los ámbitos de página (`page`), petición (`request`), sesión (`session`) y aplicación (`application`), y si lo encuentra devuelve su valor. Si no, se devuelve `null`.

Objetos implícitos

Cuando como nombre de atributo se utiliza alguno de los que el lenguaje de expresiones considera como implícitos, se devolverá el objeto asociado. Dichos objetos implícitos son:

- `pageContext`: el objeto `PageContext` actual
- `pageScope`, `requestScope`, `sessionScope`, `applicationScope`: para obtener valores de atributos de página / petición / sesión / aplicación, respectivamente.
- `param`: para obtener el valor de un parámetro de petición. Se obtiene un tipo `String` (utilizando el método `ServletRequest.getParameter(String)`)
- `paramValues`: para obtener los valores de un parámetro de petición. Se obtiene un tipo `String[]` (utilizando el método `ServletRequest.getParameterValues(String)`).
- `header`: para obtener el valor de un parámetro de cabecera. Se obtiene un tipo `String` (utilizando el método `ServletRequest.getHeader(String)`)
- `headerValues`: para obtener los valores de un parámetro de cabecera. Se obtiene un tipo `String[]` (utilizando el método `ServletRequest.getHeaderValues(String)`).
- `cookie`: para obtener el valor de una *cookie*. Se obtiene un objeto `Cookie`. Las cookies se buscan con `HttpServletRequest.getCookies()`
- `initParam`: para obtener el valor de un parámetro de inicialización. Se obtiene un tipo `String` (utilizando el método `ServletContext.getInitParameter(String)`)

EJEMPLOS

```
${sessionScope.profile}
```

Se obtiene el atributo `profile` de la sesión

```
${param.id}
```

Se obtiene el valor del parámetro `id` de la petición, o `null` si no se encuentra.

Palabras reservadas

Se tienen algunos identificadores que no podemos utilizar como nombres de atributos, como son `and`, `eq`, `gt`, `true`, `instanceof`, `or`, `ne`, `le`, `false`, `empty`, `not`, `lt`, `ge`, `null`, `div` y `mod`.

Lenguaje de expresiones y CDI

Podemos hacer que los *beans* CDI sean accesibles desde el lenguaje de expresiones. Por defecto no serán accesibles, para poder acceder a ellos deberemos darles un nombre. Esto lo haremos anotando el *bean* con la etiqueta `@Named`:

```
@Named("hola")
public class HolaMundo {
    public String getSaludo() {
        return "Hola mundo!";
    }
}
```

Podremos utilizar este *bean* desde EL de la siguiente forma:

```
${hola.saludo}
```

8.4. Librerías de etiquetas

Las **librerías de etiquetas** (*tag libs*) son conjuntos de etiquetas HTML personalizadas que permiten encapsular determinadas acciones, mediante un código Java subyacente. Es decir, se define lo que va a ejecutar la etiqueta mediante código Java, y luego se le da un nombre a la etiqueta para llamarla desde los *Facelets* o páginas JSP, estableciendo la relación entre el nombre de la etiqueta y el código Java que la implementa. Por ejemplo, una página JSP que hace uso de librerías de tags podría tener este aspecto:

```
<%@ taglib uri="ejemplo" prefix="ej" %>
<html>
<body>
<h1>Ejemplo de librerias de tags</h1>
<ej:mitag>Hola a todos</ej:mitag>
<br>
<ej:otrotag/>
```

```
</body>
</html>
```

donde se utiliza una librería llamada `ejemplo`, que se simplifica con el prefijo `ej`, de forma que todos los tags de dicha librería se referencian con dicho prefijo y dos puntos, teniendo la forma `ej:tag`. Se utilizan así los tags `mitag` y `otrotag`.

Podríamos incluir dicha librería en un *Facelet* de la siguiente forma:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ej="ejemplo">

    <h:body>
      <h1>Ejemplo de librerias de tags</h1>
      <ej:mitag>Hola a todos</ej:mitag>
      <br/>
      <ej:otrotag/>
    </h:body>
</html>
```

JSTL (*JavaServer Pages Standard Tag Library*) es una librería de tags estándar que encapsula, en forma de *tags*, muchas funcionalidades comunes en aplicaciones JSP, de forma que, en lugar que tener que recurrir a varias librerías de *tags* de distintos distribuidores, sólo necesitaremos tener presente esta librería que, además, por el hecho de ser estándar, funciona de la misma forma en cualquier parte, y los contenedores pueden reconocerla y optimizar sus implementaciones.

JSTL permite realizar tareas como iteraciones, estructuras condicionales, tags de manipulación de documentos XML, tags SQL, etc. También introduce un lenguaje de expresiones que simplifica el desarrollo de las páginas, y proporciona un API para simplificar la configuración de los tags JSTL y el desarrollo de tags personalizados que sean conformes a las convenciones de JSTL.

Librería de etiquetas JSTL

JSTL contiene una gran variedad de *tags* que permiten hacer distintos tipos de tareas, subdivididas en áreas. Así, JSTL proporciona varias *sublibrerías*, para describir cada una de las áreas que abarca, y dar así a cada área su propio espacio de nombres.

En la siguientes tablas se muestran las áreas cubiertas por JSTL (cada una con una librería):

AREA	URI	PREFIJO
Core	`http://java.sun.com/jstl/ea/core`	c
XML	`http://java.sun.com/jstl/ea/xml`	x
Internacionalización (I18N)	`http://java.sun.com/jstl/ea/fmt`	fmt

AREA	URI	PREFIJO
SQL	`http://java.sun.com/jstl/ea/sql`	sql
Functions	`http://java.sun.com/jstl/ea/functions`	fn

Las URIs mostradas en esta tabla pueden ser utilizadas en páginas JSP. En *Facelets*/JSF sólo se han incorporado las librerías *Core* y *Functions*, siendo sus URIs las siguientes:

AREA	URI	PREFIJO
Core	`http://java.sun.com/jstl/ea/core`	c
Functions	`http://xmlns.jcp.org/jsp/jstl/functions`	fn

Cada una de estar librerías se encarga de las siguientes funcionalidades:

- **Core** se utiliza para funciones de propósito general (manejo de expresiones, sentencias de control de flujo, etc).
- **XML** se emplea para procesamiento de ficheros XML.
- La librería de **internationalización** se usa para dar soporte a páginas multilenguaje, y a multiformatos de números, monedas, etc, en función de la región en que se tenga la aplicación.
- **SQL** sirve para acceder y manipular bases de datos relacionales.
- **Functions** contiene una serie de funciones de propósito general, como por ejemplo funciones para la manipulación de cadenas.

Las URIs y prefijos que se indican en la tabla pueden emplearse (aunque no es obligatorio) para utilizar las librerías en nuestros *Facelets*:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:c="http://java.sun.com/jstl/ea/core"
      xmlns:fn="http://xmlns.jcp.org/jsp/jstl/functions">

    <h:body>
        ...
    </h:body>
</html>
```

La librería Core

Los tags `core` incluyen tags de propósito general. En esta librería se tienen etiquetas para:

- Funciones de propósito general: evaluar expresiones, establecer valores de parámetros, etc.

- Funciones de control de flujo: condiciones para ejecutar unos bloques de código u otro, iteradores, etc.
- Funciones de acceso a URLs: para importar URLS en la página actual, etc.

Los tags de esta librería se presentan con el prefijo "**c**".

Tags de propósito general

out

El tag **out** evalúa el resultado de una expresión y lo pone en el objeto `JspWriter` actual. Es equivalente a la sintaxis `<%= … %>` de JSP, y también a poner directamente una expresión `${…}` en su lugar, aunque admite algunos atributos adicionales.

SINTAXIS:

Dar el valor por defecto mediante un atributo `default`:

```
<c:out value="valor"
 [escapeXML="true|false"]
 [default="valor"]/>
```

Dar el valor por defecto mediante el cuerpo del tag:

```
<c:out value="valor"
 [escapeXML="true|false"]>
  Valor por defecto
</c:out>
```

ATRIBUTOS:

- `value`: expresión que se tiene que evaluar.
- `escapeXML`: a `true` (valor por defecto) indica que los caracteres <, >, &, ', " que haya en la cadena resultado se deben convertir a sus códigos correspondientes (<, >, &, ', ", respectivamente).
- `default`: valor por defecto si el resultado es `null`. Se puede indicar por el atributo o por el cuerpo del tag.

EJEMPLO:

```
<c:out value="${datos.ciudad}"
 default="desconocida"/>
```

Sacaría el valor del campo `ciudad` del objeto `datos`, o mostraría “desconocida” si dicho valor es nulo.

set

El tag **set** establece el valor de un atributo en cualquier ámbito (`page`, `request`, `session`, `application`). Si el atributo no existe, se crea.

SINTAXIS:

Dar valor a una variable utilizando el atributo `value`:

```
<c:set value="valor" var="variable"
 [scope="page|request|session|application"]/>
```

Dar valor a una variable utilizando el cuerpo del tag:

```
<c:set var="variable"
 [scope="page|request|session|application"]>
  Valor
</c:set>
```

Dar valor a una propiedad de un objeto utilizando el atributo `value`:

```
<c:set value="valor" target="objeto"
 property="propiedad"/>
```

Dar valor a una propiedad de un objeto utilizando el cuerpo del tag:

```
<c:set target="objeto" property="propiedad">
  Valor
</c:set>
```

ATRIBUTOS:

- `value`: valor que se asigna. Podemos dar el valor con este atributo o con el cuerpo del tag.
- `var`: variable a la que se asigna el valor.
- `scope`: ámbito de la variable a la que se asigna el valor.
- `target`: objeto al que se le modifica una propiedad. Debe ser un objeto JavaBeans con una propiedad `propiedad` que pueda establecerse, o un objeto `java.util.Map`.
- `property`: propiedad a la que se le asigna valor en el objeto `target`.

EJEMPLO:

```
<c:set var="foo" value="2"/>
...
<c:out value="${foo}"/>
```

Asignaría a la variable `foo` el valor "2", y luego mostraría el valor por pantalla.

Otras Etiquetas

Existen otras etiquetas, como `remove` o `catch`, que no se comentan aquí.

Tags de control de flujo

if

El tag **if** permite ejecutar su código si se cumple la condición que contiene su atributo `test`.

SINTAXIS:

Sin cuerpo:

```
<c:if test="condicion" var="variable"
 [scope="page|request|session|application"]/>
```

Con cuerpo:

```
<c:if test="condicion" [var="variable"]
 [scope="page|request|session|application"]>
  Cuerpo
</c:if>
```

ATRIBUTOS:

- `test`: condicion que debe cumplirse para ejecutar el `if`.
- `var`: variable donde se guarda el resultado de evaluar la expresión. El tipo de esta variable debe ser `Boolean`.
- `scope`: ámbito de la variable a la que se asigna el valor de la condición.

EJEMPLO:

```
<c:if test="${visitas > 1000}">
<h1>¡Mas de 1000 visitas!</h1>
</c:if>
```

Sacaría el mensaje “¡Mas de 1000 visitas!” si el contador `visitas` fuese mayor que 1000.

choose

El tag **choose** permite definir varios bloques de código y ejecutar uno de ellos en función de una condición. Dentro del `choose` puede haber espacios en blanco, una o varias etiquetas **when** y cero o una etiquetas **otherwise**.

El funcionamiento es el siguiente: se ejecutará el código de la primera etiqueta **when** que cumpla la condición de su atributo **test**. Si ninguna etiqueta `when` cumple su condición, se ejecutará el código de la etiqueta **otherwise** (esta etiqueta, si aparece, debe ser la última hija de `choose`).

SINTAXIS:

```
<c:choose>
  <c:when test="condicion1">
    codigo1
  </c:when>
  <c:when test="condicion2">
    codigo2
```

```
  </c:when>
  ...
  <c:when test="condicionN">
    codigoN
  </c:when>
  <c:otherwise>
    codigo
  </c:otherwhise>
</c:choose>
```

EJEMPLO:

```
<c:choose>
  <c:when test="${a < 0}">
  <h1>a menor que 0</h1>
  </c:when>
  <c:when test="${a > 10}">
  <h1>a mayor que 10</h1>
  </c:when>
  <c:otherwise>
  <h1>a entre 1 y 10</h1>
  </c:otherwhise>
</c:choose>
```

Sacaría el mensaje “a es menor que 0” si la variable a es menor que 0, el mensaje “a es mayor que 10” si es mayor que 10, y el mensaje “a esta entre 1 y 10” si no se cumple ninguna de las dos anteriores.

forEach

El tag **forEach** permite repetir su código recorriendo un conjunto de objetos, o durante un número determinado de iteraciones.

SINTAXIS:

Para iterar sobre un conjunto de objetos:

```
<c:forEach [var="variable"] items="conjunto"
[varStatus="variableEstado"] [begin="comienzo"]
[end="final"] [step="incremento"]>
  codigo
</c:forEach>
```

Para iterar un determinado número de veces:

```
<c:forEach [var="variable"]
 [varStatus="variableEstado"] begin="comienzo"
 end="final" [step="incremento"]>
  codigo
</c:forEach>
```

ATRIBUTOS:

- `var`: variable donde guardar el elemento actual que se está explorando en la iteración. El tipo de este objeto depende del tipo de conjunto que se esté recorriendo.
- `items`: conjunto de elementos que recorre la iteración. Pueden recorrerse varios tipos:
    - `Array`: tanto de tipos primitivos como de tipos complejos. Para los tipos primitivos, cada dato se convierte en su correspondiente `wrapper` (`Integer` para `int`, `Float` para `float`, etc)
    - `java.util.Collection`: mediante el método `iterator()` se obtiene el conjunto, que se procesa en el orden que devuelve dicho método.
    - `java.util.Iterator`
    - `java.util.Enumeration`
    - `java.util.Map`:el objeto del atributo `var` es entonces de tipo `Map.Entry`, y se obtiene un `Set` con los mapeos. Llamando al método `iterator()` del mismo se obtiene el conjunto a recorrer.
    - `String`: la cadena representa un conjunto de valores separados por comas, que se van recorriendo en el orden en que están.
- `varStatus`: variable donde guardar el estado actual de la iteración. Es del tipo `javax.servlet.jsp.jstl.core.LoopTagStatus`.
- `begin`: indica el valor a partir del cual comenzar la iteración. Si se está recorriendo un conjunto de objetos, indica el índice del primer objeto a explorar (el primero es el 0), y si no, indica el valor inicial del contador. Si se indica este atributo, debe ser mayor o igual que 0.
- `end`: indica el valor donde terminar la iteración. Si se está recorriendo un conjunto de objetos, indica el índice del último objeto a explorar (inclusive), y si no, indica el valor final del contador. Si se indica este atributo, debe ser mayor o igual que `begin`.
- `step`: indica cuántas unidades incrementar el contador cada iteración, para ir de `begin` a `end`. Por defecto es 1 unidad. Si se indica este atributo, debe ser mayor o igual que 1.

EJEMPLO:

```
<c:forEach var="item"
 items="${cart.items}">
  <tr>
    <td>
    <c:out value="${item.valor}"/>
    </td>
  </tr>
</c:forEach>
```

Muestra el valor de todos los `items`.

forTokens

El tag **forTokens** es similar al tag **foreach**, pero permite recorrer una serie de `tokens` (cadenas de caracteres), separadas por el/los delimitador(es) que se indique(n).

SINTAXIS:

La sintaxis es la misma que `foreach`, salvo que se tiene un atributo `delims`, obligatorio.

ATRIBUTOS:

- `var` : igual que para `foreach`
- `items` : cadena que contiene los tokens a recorrer
- `delims` :conjunto de delimitadores que se utilizan para separar los tokens de la cadena de entrada (colocados igual que los utiliza un `StringTokenizer`).
- `varStatus` : igual que para `foreach`
- `begin` : indica el índice del token a partir del cual comenzar la iteración.
- `end` : indica el índice del token donde terminar la iteración.
- `step` : igual que para `foreach` .

EJEMPLO:

```
<c:forTokens var="item"
 items="un#token otro#otromas" delims="# ">
  <tr>
    <td>
    <c:out value="${item}"/>
    </td>
  </tr>
</c:forEach>
```

Definimos dos separadores: el `'#'` y el espacio `' '` . Así habrá 4 iteraciones, recorriendo los tokens `"un"` , `"token"` , `"otro"` y `"otromas"` .

Tags de manejo de URLs

import

El tag **import** permite importar el contenido de una URL.

SINTAXIS:

Para copiar el contenido de la URL en una cadena:

```
<c:import url="url" [context="contexto"]
 [var="variable"]
 [scope="page|request|session|application"]
 [charEncoding="codificacion"]>
  cuerpo para tags "param" opcionales
</c:import>
```

Para copiar el contenido de la URL en un `Reader` :

```
<c:import url="url" [context="contexto"]
 varReader="variableReader"
 [charEncoding="codificacion"]>
  codigo para leer del Reader
</c:import>
```

ATRIBUTOS:

* `url` : URL de la que importar datos
* `context` : contexto para URLs que pertenecen a contextos distintos al actual.
* `var` : variable (`String`) donde guardar el contenido de la URL
* `varReader` : variable (`Reader`) donde guardar el contenido de la URL
* `scope` : ámbito para la variable `var`
* `charEncoding` : codificación de caracteres de la URL

EJEMPLO:

```
<c:import url="http://www.ua.es"
 var="universidad">
  <c:out value="${universidad}"/>
</c:import>
```

Obtiene y muestra el contenido de la URL indicada.

param

El tag `param` se utiliza dentro del tag `import` y de otros tags (`redirect` , `url`) para indicar parámetros de la URL solicitada. Dentro del tag `import` sólo se utiliza si la URL se guarda en una cadena. Para los `Readers` no se emplean parámetros.

SINTAXIS:

Sin cuerpo:

```
<c:param name="nombre" value="valor"/>
```

Con cuerpo:

```
<c:param name="nombre">
  Valor
</c:param>
```

ATRIBUTOS:

* `name` : nombre del parámetro
* `value` : valor del parámetro. Puede indicarse bien mediante este atributo, bien en el cuerpo del tag.

EJEMPLO:

```
<c:import url="http://localhost/mipagina.jsp"
 var="universidad">
  <c:param name="id" value="12"/>
</c:import>
```

Obtiene la página `mipagina.jsp?id=12` (le pasa como parámetro `id` el valor 12).

Otras Etiquetas

Existen otras etiquetas, como `url` o `redirect`, que no se comentan aquí.

Ejemplo

Vemos cómo quedaría el ejemplo visto en la sesión anterior para la librería `request` adaptado a la librería `core`. Partiendo del mismo formulario inicial:

```
<html>
<body>
  <form action="request.jsp">
    Nombre:
    <input type="text" name="nombre">
    <br>
    Descripcion:
    <input type="text" name="descripcion">
    <br>
    <input type="submit" value="Enviar">
  </form>
</body>
</html>
```

Para obtener los parámetros podríamos tener una página como esta:

```
<%@ taglib uri="http://java.sun.com/jstl/ea/core" prefix="c" %>

<html>
<body>
  Nombre: <c:out value="${param.nombre}"/>
  <br>
  <c:if test="${not empty param.descripcion}">
    Descripcion: <c:out value="${param.descripcion}"/>
  </c:if>
</body>
</html>
```

Hemos utilizado en este caso, como ejemplo, los tags `out` e `if` (para comprobar si hay parámetro `descripcion`). En este último caso, utilizamos el operador `empty` en el lenguaje de expresiones, para ver si hay o no valor.

La librería de funciones

JSTL dispone también de un conjunto de **funciones** que pueden emplearse desde dentro del lenguaje de expresiones, y permiten sobre todo manipular cadenas, para sustituir caracteres, concatenar, etc.

Para utilizar estas funciones, cargaríamos la directiva `taglib` correspondiente:

```
<%@ taglib uri="http://java.sun.com/jstl/ea/functions" prefix="fn" %>
```

Y luego utilizaríamos las funciones que haya disponibles, dentro de una expresión del lenguaje:

```
La cadena tiene <c:out value="${fn:length(miCadena)}"/> caracteres
```

Librerías de JSF/*Facelets*

Hemos visto hasta el momento librerías de JSTL que podemos utilizar dentro de los *Facelets*, pero que también encontraremos dentro de otros *frameworks*. Vamos a ver ahora algunas librerías propias de *Facelets*/JSF, como la librería HTML de JSF y la librería UI de *Facelets* que nos permitirá crear plantillas para las páginas.

Librería HTML de JSF

Una de las principales librerías de etiquetas utilizada en *Facelets* es la librería HTML de JSF. Esta librería contiene una serie de componentes de la interfaz que se renderizarán en la página como HTML. Gran parte de los componentes de esta librería son campos de formularios, cuyo valor puede vincularse con *managed beans* y podemos establecer la forma mediante etiquetas de la librería de funciones de JSF. Por ejemplo podríamos tener un campo de texto como el siguiente:

```
<h:inputText id="precio"
             size="4"
             value="#{articulo.precio}"
             title="Cantidad">
    <f:validateLongRange minimum="0"/>
</h:inputText>
```

En el ejemplo anterior se crea un campo de texto que se renderizará como una etiqueta `<input>` en el HTML resultante, y quedará vinculado en el servidor con el precio del *managed bean* `articulo`. Además, se especifica que el campo debe ser validado de forma que nunca tenga un valor inferior a `0`.

Encontramos otras etiquetas como las siguientes dentro de esta librería:

Etiqueta	**Función**
`<h:commandLink>`	Representa un enlace a otra página, se renderiza como `<a>`
`<h:dataTable>`	Tabla que se puede actualizar de forma dinámica, se renderiza como `<table>`
`<h:column>`	Representa una columna en una tabla
`<h:panelGroup>`	Representa una fila en una tabla, se renderiza como `<div>`
`<h:graphicImage>`	Muestra una imagen, se renderiza como `<img>`
`<h:outputText>`	Muestra texto plano
`<h:outputStylesheet>`	Permite especificar una hoja de estilo
`<h:head>`	Equivalente a `<head>`, permite reubicar bloques de *script*
`<h:body>`	Equivalente a `<body>`, permite reubicar bloques de *script*

En todas estas etiquetas encontramos una serie de atributos comunes:

Atributo	Función
`id`	Identifica al componente de forma única.
`rendered`	Podemos especificar una condición mediante EL que indica si el componente se debe mostrar o no.
`style`	Especifica el estilo CSS.
`styleClass`	Especifica una clase de estilo.

Recursos

Hablamos de recursos para referirnos a cualquier artefacto que una página web necesita para mostrarse correctamente, como son las imágenes, ficheros JavaScript, hojas de estilo, etc. En los *Facelets* podremos hacer referencia a estos recursos simplemente indicando el nombre del recurso, siempre que éstos se encuentren en determinadas ubicaciones estándar. Estas ubicaciones son:

- Directorio `resources` en la raíz del contexto.
- Dentro del *classpath*, en `META-INF/resources`.

Si el recurso se encuentra en una de las localizaciones anteriores podemos hacer referencia a él únicamente mediante su nombre en los atributos de las etiquetas de JSF/*Facelets*, como por ejemplo:

```
<h:outputStylesheet library="css" name="estilo.css"/>
```

En este caso podríamos tener la hoja de estilo en `/resources/css/estilo.css`.

Podemos también hacer referencia a la ubicación mediante lenguaje de expresiones:

```
<h:graphicImage value="#{resource['imagenes:logo.gif']}"/>
```

En el caso anterior buscará la imagen en `/resources/imagenes/logo.gif`.

Plantillas

La tecnología de *Facelets* nos permite crear plantillas para las páginas. Estas plantillas definen una determinada disposición de los elementos en el documento, y podrán ser aplicadas a diferentes páginas para así tener una estructura homogénea en todo el sitio web. En estas plantillas por ejemplo podríamos definir un bloque de cabecera, menú lateral, cuerpo de la página, pie de página, etc.

Para la creación de plantillas utilizaremos la librería de Interfaz de Usuario (UI) de *Facelets*:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets" ❶
```

```
        xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8" />
        <h:outputStylesheet library="css" name="estilo.css"/>
        <title>Plantilla con Facelets</title>
    </h:head>

    <h:body>
        <div id="top" class="top">
            <ui:insert name="cabecera">Cabecera</ui:insert> ❷
        </div>
        <div>
        <div id="left">
             <ui:insert name="menu">Menú lateral</ui:insert>
        </div>
        <div id="content" class="content">
             <ui:insert name="cuerpo">Cuerpo de la página</ui:insert>
        </div>
        </div>
    </h:body>
</html>
```

❶ Declaramos la librería UI de *Facelets*

❷ Utilizamos `<ui:insert>` para crear un lugar donde insertar contenido dentro de la plantilla general.

En el ejemplo anterior hemos creado una plantilla de página web donde se definen tres lugares donde vamos a poner insertar contenido: `cabecera`, `menu` y `cuerpo`. Vamos a crear a continuación una página que se ajuste a dicha plantilla e inserte contenido en cada una de las secciones.

Consideremos que el fichero anterior con la plantilla ha sido guardado con el nombre `plantilla.xhtml`. Una página que utilice dicha plantilla podría definirse de la siguiente forma:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:body>
        <ui:composition template="./plantilla.xhtml"> ❶
            <ui:define name="cabecera"> ❷
                Título de la página
            </ui:define>

            <ui:define name="menu">
                <h:outputLabel value="Menú"/>
            </ui:define>

            <ui:define name="cuerpo">
                <h:graphicImage value="#{resource['images:logo.gif']}"/>
                <h:outputText value="2014 (c) DCCIA"/>
```

```
            </ui:define>
        </ui:composition>
    </h:body>
</html>
```

❶ Con la etiqueta `<ui:composition>` indicamos la plantilla que vamos a utilizar.

❷ Con la etiqueta `<ui:define>` indicamos el contenido que vamos a insertar en cada una de las secciones de la plantilla utilizada.

Vemos en el ejemplo anterior que podremos poner una etiqueta `<ui:define>` por cada elemento definido mediante `<ui:insert>` en la plantilla que estamos utilizando.

8.5. Ejercicios

Página de chat con *Facelets* (1 punto)

Vamos a crear una implementación alternativa del listado de mensajes del chat mediante un *Facelet* que utilice la librería JSTL y el lenguaje de expresiones. Los *Facelets* son más apropiados que los *Servlets* para crear la vista de la aplicación, por lo que será conveniente pasar la presentación que actualmente está realizando `ListaMensajesServlet` a un *Facelet*.

Seguiremos los siguientes pasos:

- Añade la **configuración de JSF** al proyecto desde IntelliJ. Se deberá crear el fichero `faces-config.xml` en `WEB-INF` y mapear el *Faces Servlet* a las direcciones de tipo `*.xhtml`.
- Crea desde IntelliJ un nuevo *Facelet* llamado `listaMensajes.xhtml` en el directorio `webapp/chat`.
- Importa en el *Facelet* la **librería JSTL Core**.
- Crearemos un recuadro para el chat siguiendo el mismo estilo de las páginas utilizadas en sesiones anteriores. Esta vez podemos **importar la hoja de estilo** `estilo.css` como un recurso.

Deberemos copiar el fichero `estilo.css` a un directorio donde pueda ser importado como recurso (`webapp/resources`).

- En la cabecera del cuadro del chat **indicaremos el nombre del usuario** que hay actualmente conectado. Podemos poner un mensaje como "Conectado al chat como <nombre>".

El nombre del atributo donde se guarda el *nick* es `org.expertojava.cweb.chat.nick`, lo cual puede causar problemas al utilizarlo dentro del lenguaje de expresiones. Para evitar estos problemas podríamos utilizar una sintaxis como `${sessionScope["nombre_atributo"]}`.

- Utilizaremos un *managed bean* para **mostrar los mensajes** en la página. Deberemos:
  - Si la cola está vacía, mostraremos el mensaje "Todavía no se ha enviado ningún mensaje al chat".
  - Si la cola tiene mensajes, iteraremos por la cola para mostrar todos los mensajes junto al nombre del usuario que los envió.

No podemos utilizar directamente el objeto `ColaMensajes` dentro del lenguaje de expresiones al tratarse de un objeto de tipo `LinkedList`. Podemos crearnos un nuevo *managed bean* que nos dé acceso a dicha lista mediante uno de sus atributos. Por ejemplo, podríamos crear un objeto `Chat` dentro del ámbito de la aplicación con un atributo `cola` en el que inyectaremos el objeto `ColaMensajes`. De esta forma podríamos acceder a ella desde lenguaje de expresiones con `${chat.cola}`.

Todo lo anterior debe realizarse utilizando **únicamente JSTL y lenguaje de expresiones**.