Capítulo 3. Tecnología Java Servlet

1. Introducción

La tecnología Servlet [7] de Java es una poderosa y eficiente solución para crear contenido dinámico en la Web. Durante los últimos años los Servlets se han convertido en un bloque fundamental del servidor Java. El poder de los Servlets viene del uso de Java como plataforma y de la interacción con el contenedor de Servlets. La plataforma Java proporciona al desarrollador una API robusta, de programación orientada a objetos, independiente de la plataforma, con tipos estrictos, recolector de basura, y con todas las características de seguridad de la Máquina Virtual de Java (JVM, Java Virtual Machine). Complementándolo, el contenedor de Servlets proporciona administración de ciclos de vida, un único proceso para compartir y administrar recursos de toda la aplicación, e interacción con el servidor Web. Juntos, consiguen que los Servlets sean la tecnología más deseable para los desarrolladores de servidores en Java.

Los Servlets siempre forman parte de un proyecto mayor llamado Aplicación Web. Una Aplicación Web es una colección completa de recursos para un sitio Web. Nada impide a una Aplicación Web consistir en cero, uno, o múltiples Servlets, pero un contenedor de Servlets administra todos los de la Aplicación Web. Las Aplicaciones Web y sus archivos de configuración están especificados en la norma Servlet.

El principal propósito de la especificación Servlet es definir un mecanismo robusto para enviar contenido al cliente tal y como se define en el modelo Cliente/Servidor. Los Servlets generalmente se suelen usar para generar contenido dinámico en la Web y tienen soporte nativo para HTTP [52].

1.1 Introducción histórica

Tan pronto como la Web se empezó a usar para proporcionar servicios, los proveedores de éstos reconocieron la necesidad de contenido dinámico. Los Applets, uno de los primeros intentos de conseguirlo, se concentran en el uso de la plataforma del cliente para proporcionar al usuario este contenido dinámico.

Al mismo tiempo, los desarrolladores también investigaron el uso del servidor para lograr este propósito. Inicialmente, los scripts *Common Gateway Interface* (*CGI*) fueron la principal tecnología utilizada para generar contenido dinámico. Aunque ampliamente usados, la tecnología de scripts *CGI* tenía algunos inconvenientes, incluyendo dependencia de la plataforma y falta de escalabilidad. Para solventar estas limitaciones, la tecnología de Servlets Java se creó como un medio portable para proporcionar contenido dinámico orientado al usuario.

La especificación original de Servlet fue creada por Sun Microsystems [39], finalizada en Junio de 1997. A partir de la versión 2.3, la especificación Servlet la desarrolla el JCP [14].

La última versión de Servlet que se ha hecho pública es la 2.5, el 10 de Mayo de 2006, aunque la utilizada en el actual proyecto es la 2.4, ya que era la versión disponible

en el momento de empezar su desarrollo. De hecho, en el momento de finalización del proyecto aún no había una versión del contenedor Tomcat [51] que soporte la nueva versión de Servlet 2.5 completamente; por ejemplo; la versión 6.0 de Tomcat está en desarrollo y sólo está disponible su versión beta.

2 Introducción a la Aplicación Web

Una Aplicación Web es una extensión dinámica de un servidor Web o un servidor de aplicaciones. En la plataforma Java 2, los *componentes Web* proporcionan capacidad de extensión dinámica a un servidor Web. Los componentes Web son Servlets Java, páginas *JSP*, o los puntos finales de los servicios Web. La interacción entre un cliente y una Aplicación Web se puede ver en la Figura 3.1:

1) El cliente envía una petición HTTP al servidor Web.
2) El servidor Web que implementa los Servlets Java y la tecnología de Páginas de Servidor Java (JavaServer Pages, JSP) convierte la petición en un objeto `HttpServletRequest`.
3) Éste se entrega a un componente Web, que interactúa con componentes JavaBeans [13]
4) A partir de ahí se accede a los datos, o directamente a una base de datos para generar contenido dinámico.
5) El componente Web genera entonces un `HttpServletResponse` o pasa la petición a algún otro componente Web.
6) De una forma o de otra, se generará un objeto `HttpServletResponse` que el servidor Web convertirá en una respuesta HTTP y la devolverá al cliente.

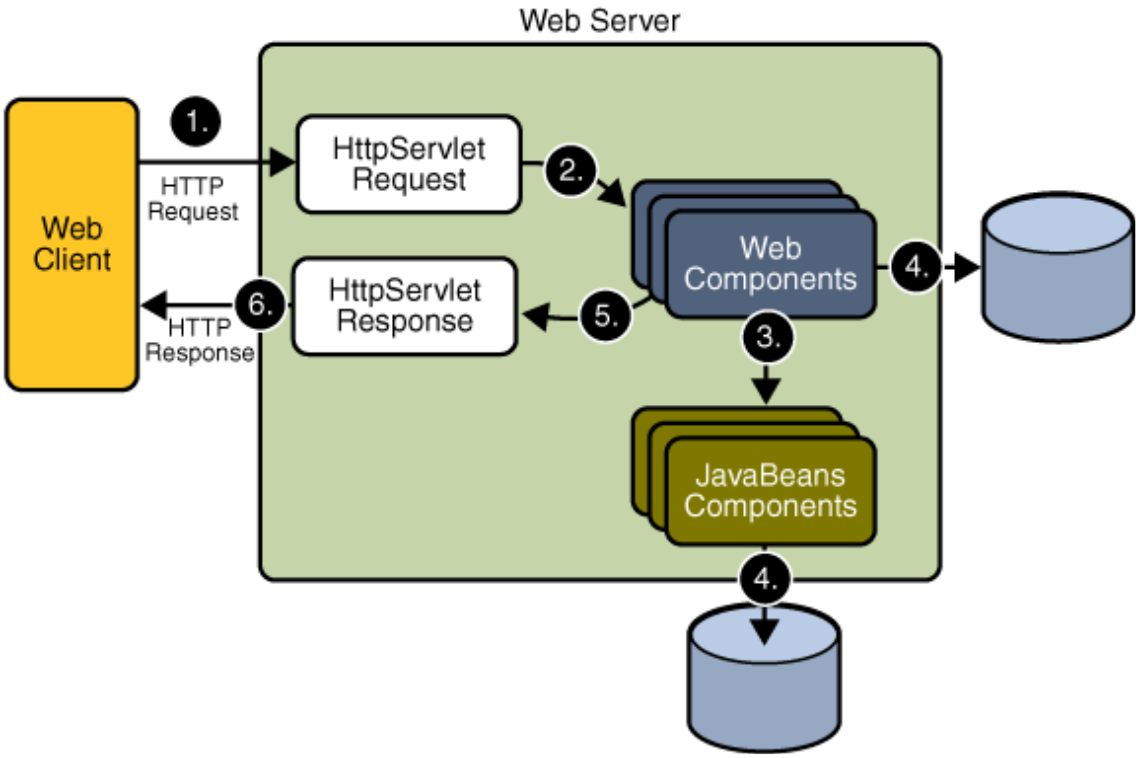

Figura 3.1. Proceso de la petición por parte de una Aplicación Web Java

Los Servlets son clases Java que dinámicamente procesan peticiones y mandan respuestas. Las JSPs son documentos de texto que se ejecutan como Servlets pero que permiten una mayor aproximación a la creación de contenido estático. Aunque se pueden intercambiar, cada uno tiene sus virtudes. Los Servlets encajan mejor para las aplicaciones orientadas a servicio (los puntos finales de servicio se implementan como Servlets) y las funciones de control de una aplicación orientada a presentación, como el envío de peticiones y el tratamiento de datos que no son de texto. Las JSPs son más apropiadas para generar marcas de texto como HTML, y XML [4].

Los componentes Web se ejecutan gracias a los servicios de una plataforma de ejecución llamada *contenedor Web*. Un contenedor Web proporciona servicios como entrega de peticiones, seguridad, concurrencia, y gestión de ciclos de vida. También proporciona a los componentes Web acceso a las APIs.

2.1 Módulos Web

En la arquitectura Java EE [17], los archivos de los componentes Web y el contenido Web estático como las imágenes se llaman *recursos Web*. Un *módulo Web* es la unidad más pequeña desplegable y utilizable de recursos Web, y que en el entorno Java EE se corresponde también con una *Aplicación Web*, tal y como se define en la especificación Servlet de Java.

Además de los componentes Web y los recursos Web, un módulo Web puede contener otros archivos:

- Clases de servicio del servidor (beans de bases de datos, carros de la compra, etc.). A menudo estas clases siguen la arquitectura de componentes JavaBeans [13].
- Clases del lado del cliente (Applets y clases de servicio).

Un módulo Web tiene una estructura específica, mostrada en la Figura 3.2. El directorio superior de un módulo Web es el *directorio raíz* de la aplicación. El directorio raíz es donde se almacenan las páginas JSP, las clases y archivos del lado del cliente, y los recursos de contenido estático, como las imágenes.

El directorio raíz contiene un subdirectorio llamado `/WEB-INF/`, que contiene los siguientes archivos y directorios:

- `web.xml`: El descriptor de despliegue de la aplicación Web.
- Archivos descriptores de biblioteca de etiquetas (Tag Library Descriptors, TLD).
- `classes`: Un directorio que contiene las clases del lado del servidor: Servlets, clases de servicio y componentes JavaBeans.
- `Tags`: Un directorio que contiene archivos de etiquetas `.tag`.
- `lib`: Directorio que contiene los archivos JAR [42] de bibliotecas llamadas por las clases del servidor.

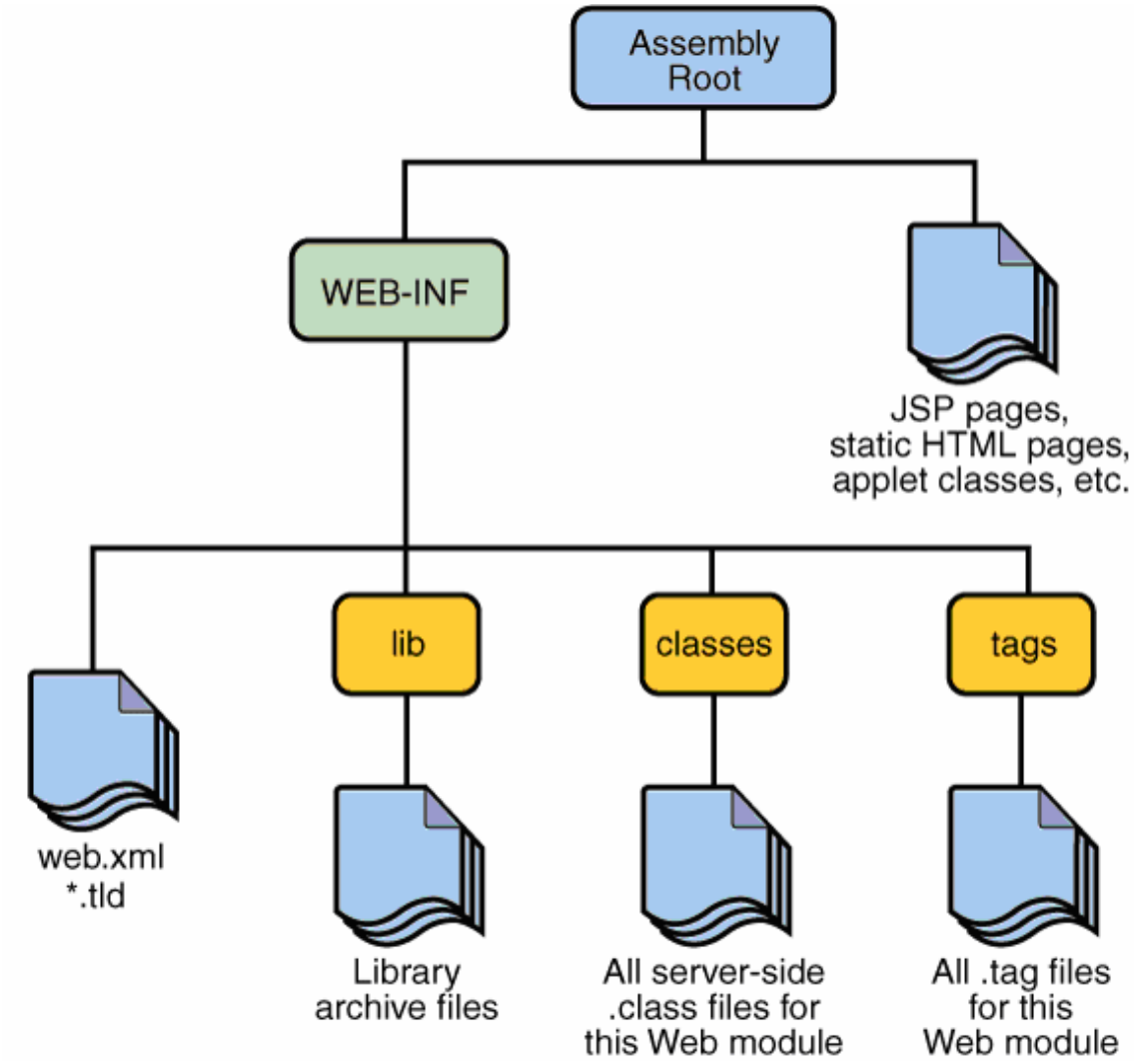

Figura 3.2. Estructura de un Módulo Web

El descriptor de despliegue de la aplicación Web sólo hace falta si un módulo Web contiene algún Servlet, filtro, escuchador, o algún parámetro inicial de la Aplicación; si sólo contiene páginas JSP y archivos estáticos no hará falta incluir el archivo `web.xml`.

En el directorio raíz o en `/WEB-INF/classes/` se pueden crear subdirectorios específicos de la aplicación (es decir, directorios de paquete).

Un módulo Web se puede desplegar como una estructura de archivos sin empaquetar o empaquetado en un archivo JAR [42] conocido como archivo Web (Web archive, WAR), que usa la extensión `.war` para diferenciarlos de los archivos JAR comunes. El módulo Web así descrito es portable y se puede desplegar en cualquier contenedor que cumpla la especificación *Java Servlet*.

Para desplegar un WAR algunos contenedores incluyen algún descriptor de ejecución de despliegue, como el archivo `sun-web.xml` localizado en `/WEB-INF/` para el *Servidor de Aplicaciones de Java*. Es un archivo que contiene información como la ruta de contexto de la aplicación Web y cómo se convierten los nombres portables de

los recursos de la aplicación a recursos del Servidor de Aplicaciones. Aunque estos archivos no suelen ser necesarios, ya que no están descritos por la especificación.

2.2 `/WEB-INF/` y `web.xml`

La especificación Servlet define un archivo de configuración llamado "descriptor de despliegue", que contiene meta-datos para una Aplicación Web y que comprueba el contenedor cuando la carga. Este archivo contiene, como ya se ha dicho, meta-datos, como la página por defecto a mostrar, Servlets a cargar, o restricciones de seguridad que imponer a los archivos. La especificación Servlet también define que todo el directorio `/WEB-INF/` de cualquier Aplicación Web se debe mantener oculto a los usuarios de la aplicación (es decir, no se puede navegar con un explorador Web a http://servidor:puerto/contexto/WEB-INF, ni a ningún subdirectorio por debajo de éste).

El archivo web.xml es un archivo XML [4] cuyo esqueleto es el siguiente:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" version="2.4">

</web-app>
```

Ahí se define la configuración de la Aplicación Web, como se ve a continuación.

2.3 Configuración de Aplicaciones Web

La configuración de una Aplicación Web se realiza añadiendo elementos en el descriptor de despliegue de la aplicación, es decir, en el archivo `web.xml`. A continuación se muestran distintos aspectos de una Aplicación Web que se pueden configurar mediante el mismo archivo. Hay otros aspectos que también se pueden configurar y que, o se verán más adelante en otras secciones, o son más específicos y esta pequeña introducción no los trata porque se escapa del ámbito de la Aplicación Web diseñada en el proyecto.

2.3.1 Mapeo de URLs a componentes Web

Cuando el contenedor recibe una petición debe determinar qué componente debería ocuparse de ella. Esto se realiza mapeando el camino de la URL [20] contenido en la petición a un componente Web. La URL contiene un contexto y un alias:

```
http://servidor:puerto/contexto/alias
```

El *alias* identifica al componente Web que debe manejar la petición. La dirección del alias debe comenzar con una barra hacia la derecha ('`/`') y terminar con una cadena. Ya que los contenedores Web mapean los alias que terminan con `*.jsp` directamente, no se necesita especificar un alias para las JSPs a menos que se prefiera referir a la página mediante otro nombre que no sea su nombre de archivo. Así, para establecer un alias para un Servlet, `web.xml` debe contener lo siguiente:

- Un elemento `servlet` que establece un nombre para una clase Servlet.
- Un elemento `servlet-mapping` que mapea ese nombre establecido para el Servlet a una URL que debe comenzar con el carácter '/'.

2.3.2 Declaración de Archivos de Bienvenida

El mecanismo de los *archivos de bienvenida* permite especificar una lista de archivos que el contenedor Web usará para añadir a la petición de una URL (llamada petición parcial válida) que no se mapee a ningún componente Web.

Por ejemplo, si se define el archivo de bienvenida `welcome.html`, cuando un cliente pida la URL `http://servidor:puerto/apweb/directorio`, donde *`directorio`* no se mapea a un Servlet o JSP, se devolverá al cliente el archivo `servidor:puerto/apweb/directorio/welcome.html`.

Si un contenedor recibe una petición parcial válida, examina la lista de archivos de bienvenida y añade a la petición parcial cada archivo de bienvenida en el orden especificado y comprueba si algún Servlet o contenido estático está mapeado a esa URL. Luego, el contenedor envía la petición al primer recurso que coincide.

Para especificar un archivo de bienvenida se debe anidar un elemento `welcome-file` en un elemento `welcome-file-list` en `web.xml`.

2.3.3 Estableciendo Parámetros Iniciales

Los componentes Web en un módulo Web comparten un objeto que representa el contexto de la Aplicación. Se pueden pasar parámetros iniciales al contexto de la Aplicación Web, a los que tendrán acceso todos los elementos de ésta, o al de un componente Web específico(a un Servlet, por ejemplo), al que sólo tendrá acceso ese componente.

Para añadir un parámetro inicial se necesita lo siguiente en `web.xml`:

- Un elemento `param-name` que especifica el nombre del parámetro.
- Un elemento `param-value` que especifica el valor del parámetro
- Si es un parámetro inicial de contexto irán englobados en un elemento `context-param`; y si es un parámetro inicial de un componente Web, irán englobados en un elemento `init-param` (que irá en el elemento del módulo Web que corresponda, por ejemplo un elemento `servlet`).

2.3.4 Mapeo de Errores a Pantallas de Error

Se puede configurar la Aplicación Web para que si durante su ejecución ocurre un error, muestre una pantalla de error específica de acuerdo al tipo producido. En particular, se puede especificar un mapeo entre códigos de error HTTP [52] o excepciones Java a cualquier componente Web. Se tendrá en `web.xml` un elemento `error-page` para cada error que se quiera mapear a un elemento, y dentro:

- Un elemento `exception-type` especificando la excepción o un elemento `error-code` con el código HTTP causa del error que se quiera capturar.
- Un elemento `location` que especifica el nombre de un recurso Web que será invocado cuando se dé el error. El nombre debe comenzar con '/'.

3 ¿Qué es un Servlet?

Un Servlet es una clase del lenguaje de programación Java usada para extender las capacidades de los servidores que proporcionan acceso a aplicaciones a través del modelo de programación petición-respuesta. Aunque los Servlets pueden responder a cualquier tipo de petición, comúnmente se utilizan para extender las aplicaciones que sirven los servidores Web. Para esas aplicaciones, la tecnología de Servlet de Java define unas clases específicas HTTP.

Los paquetes `javax.servlet` y `javax.servlet.http` proporcionan interfaces y clases para escribir Servlets. Todos los Servlets deben implementar la interfaz `Servlet`, que define los métodos de los ciclos de vida. Cuando se implemente un servicio genérico, se puede usar o extender la clase `GenericServlet` proporcionada por la API Java Servlet. La clase `HttpServlet` proporciona métodos, como `doGet` y `doPost`, para manejar los servicios específicos HTTP.

3.1 Ciclo de vida de un Servlet

La clave para entender la funcionalidad a bajo nivel de los Servlets es comprender el simple ciclo de vida que siguen. El ciclo de vida se ejecuta en el entorno multi-hilo en el que se ejecuta el Servlet y explica algunos de los mecanismos de los que dispone el desarrollador para compartir recursos de servidor.

Este ciclo de vida de los Servlets (ver Figura 3.3) es la primera razón por la que los Servlets (y también las JSP, como se verá más adelante) superan al tradicional CGI. En contraposición al ciclo de vida de un solo uso de CGI, los Servlets siguen una vida de tres fases: *inicialización*, *servicio*, y *destrucción*, con la inicialización y la destrucción ejecutados sólo una vez, y el servicio muchas veces, típicamente.

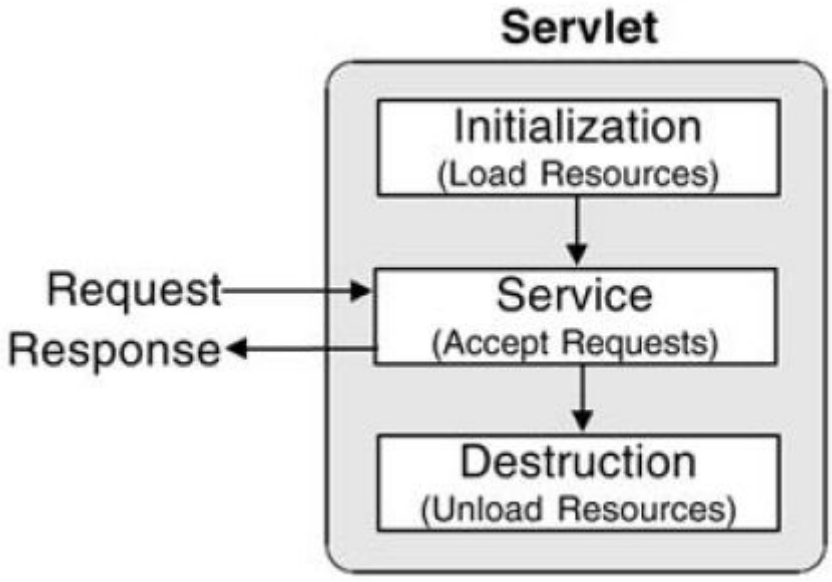

Figura 3.3. Diagrama del Ciclo de Vida de un Servlet

La inicialización es la primera fase del ciclo de vida y representa la creación e inicialización de las fuentes que el Servlet puede necesitar para servir las peticiones. Todos los Servlets deben implementar la interfaz `java.xservlet.Servlet`. Esta interfaz define el método `init()` para seguir la fase de inicialización del ciclo de vida del Servlet. Cuando un contenedor carga un Servlet, invoca el método `init()` antes de servir cualquier petición.

La fase de servicio del ciclo de vida del Servlet representa todas las interacciones con las peticiones hasta que el Servlet es destruido. La interfaz Servlet lo representa mediante el método `service()`. Éste método se invoca una vez por petición y es el responsable de generar las respuestas. La especificación Servlet define que el método `service()` tiene dos parámetros: un objeto `javax.servlet.ServletRequest` y un `javax.servlet.ServletResponse`. Estos dos objetos representan la petición del cliente de contenido dinámico y la respuesta del Servlet al cliente, respectivamente. Por defecto, un Servlet es multi-hilo, lo que significa que típicamente el contendor sólo carga una instancia de un Servlet. La inicialización se realiza sólo una vez, y las posteriores peticiones se manejan concurrentemente por hilos ejecutando este método `service()`.

La fase de destrucción del ciclo de vida del Servlet representa que el contenedor va a dejar de usar el Servlet. La interfaz `Servlet` define el método `destroy()` que corresponde con la fase destrucción del ciclo. Cada vez que un Servlet se va a dejar de usar, el contendor llama al método `destroy()`, permitiendo que el Servlet finalice y limpie cualquier recurso que hubiera creado. Usando de forma adecuada las fases de inicialización, servicio y destrucción del ciclo de vida del Servlet, éste puede utilizar adecuadamente recursos de la aplicación. Durante la inicialización un Servlet carga todo lo que necesita para servir peticiones, luego se usan los recursos durante la fase de servicio y finalmente se limpian en la fase de destrucción.

En la práctica, el contenido Web se accede principalmente vía HTTP, y el Servlet básico no sabe nada de HTTP, pero hay una implementación especial de Servlet, `javax.servlet.http.HttpServlet`, que está especialmente diseñado para él.

3.2 Servlet para el *World Wide Web*

El Protocolo de Transferencia de HiperTexto (HTTP, HyperText Transfer Protocol) [52] se usa para la mayoría de las transacciones en el World Wide Web. Los Servlets soportan las transacciones HTTP gracias a la clase `javax.servlet.http.HttpServlet`.

La implementación del método `service()` de `HttpServlet`, es un poco especial ya que tiene algunas modificaciones. Cada vez que llega una petición que hay que servir, llama a uno de los siete métodos ayudantes. Estos siete métodos se corresponden directamente a los siete métodos HTTP y son los siguientes: `doGet()`, `doPost()`, `doPut()`, `doHead()`, `doOptions()`, `doDelete()`, y `doTrace()`.Se invoca el método apropiado que concuerde con el método de una petición HTTP dada. El ciclo de vida de `HttpServlet` se puede ver en la Figura 3.4:

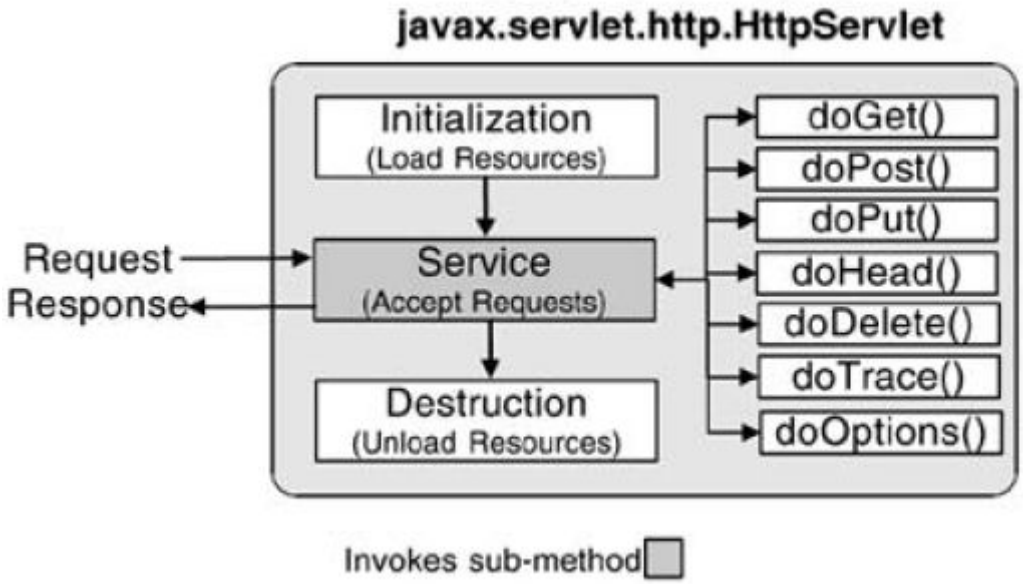

Figura 3.4. Ciclo de Vida de `HttpServlet`

Normalmente sólo se llamará a un método para una petición. Se podría llamar a más de uno si el desarrollador sobrescribe los métodos y hace que se llamen unos a otros. Las fases de inicialización y destrucción son iguales que las descritas anteriormente.

Para codificar un `HttpServlet` el desarrollador sólo tiene que ocuparse de los métodos que se necesitan personalizar, ya que la clase se ocupa de las partes redundantes de la petición y la respuesta HTTP. La manipulación de la petición y la respuesta se hace a través de dos objetos: `javax.servlet.http.HttpServletRequest` y `javax.servlet.http.HttpServletResponse`. Los dos objetos se pasan como parámetros cuando se invocan los métodos de servicio HTTP.

3.3 Poniendo en funcionamiento un Servlet

Un Servlet necesita ser desplegado para que pueda generar respuestas dinámicas. El archivo de la clase del Servlet debe ir en el directorio `/WEB-INF/classes` (o algún subdirectorio de paquete). Y para que se pueda acceder a él se debe declarar una o más URLs en `web.xml` mapeando el nombre del Servlet a una URL:

- Primero se declara el Servlet mediante el elemento `servlet` que le da un nombre (que debe ser único entre todos los nombres de los Servlets) y la localización de la clase Java apropiada, mediante las etiquetas `servlet-name` y `servlet-class` respectivamente.
- Luego se mapea usando el elemento `servlet-mapping` el nombre definido anteriormente a una URL o varias mediante las etiquetas `servlet-name` y `url-pattern` respectivamente.

3.4 Codificando un Servlet

La codificación de un Servlet empieza por la implementación de los métodos que se quieran utilizar, ya sea de una clase `Servlet` o de una `HttpServlet`. Y la

codificación de cada Servlet específico depende, claro, de para lo que se vaya a usar. Mediante los objetos que reciben éstos métodos se puede acceder a todas las propiedades de la petición y de la respuesta (sea un Servlet básico, Http, o algún otro tipo de subclase) de una forma sencilla. Por ejemplo, para enviar información al cliente, mediante el objeto `HttpServletResponse` se puede obtener, gracias al método `getWriter()` o `getOutputStream()`, un flujo de salida para escribir contenido en la respuesta al cliente. Estos dos métodos devuelven los objetos para enviar texto o contenido binario al cliente, respectivamente. Y mediante el objeto `HttpServletRequest` se puede acceder a las cabeceras, parámetros y archivos enviados por el cliente; como por ejemplo, mediante el método `getParameter(String NombreDelParametro)` se accede al valor `String` del parámetro requerido del formulario enviado en la petición.

3.5 Contexto de un Servlet

La interfaz `ServletContext` representa la visión que tiene el Servlet del resto de la Aplicación Web en la que se ejecuta. Mediante esta interfaz el Servlet puede acceder a recursos de la Aplicación Web, traducción de directorios virtuales, un mecanismo común para registrar información y un ámbito de aplicación en el que almacenar objetos. Las implementaciones de la interfaz las proporcionan cada uno de los desarrolladores de los contenedores, aunque su funcionalidad es la misma, tal y como está definida en la interfaz.

3.5.1 Acceso a los parámetros iniciales de la Aplicación Web

Todos los Servlets tienen un objeto específico asociado, `ServletConfig`, que contiene el método `getServletContext()` para acceder al contexto de la aplicación. Mediante el objeto `ServletContext` se puede acceder a los parámetros iniciales de la Aplicación Web, y obtenerlos en un `String`.

3.5.2 Almacenamiento de parámetros para la Aplicación Web

Se pueden almacenar objetos mediante el objeto `ServletContext` que estarán disponibles en toda la Aplicación Web, y que también se pueden recoger mediante este mismo objeto.

3.5.3 Traducción de directorios virtuales

Todos los recursos de la Aplicación Web están abstraídos a un directorio virtual, que empieza con '/', y sigue con una ruta virtual hasta los subdirectorios y recursos. Pero para acceder a la localización física de un recurso hace falta la dirección real. Así, el objeto `ServletContext` proporciona el método `getRealPath(String camino)` para obtener un `String` con la dirección real del recurso.

Es importante que la Aplicación Web sea portable, ya que puede que se ejecute en distintos contenedores, o incluso desde un archivo WAR, así que se deben evitar referencias absolutas a recursos, y acceder a ellos mediante el método anterior, u otros similares del objeto `ServletContext` para conseguir que sea realmente portable.

3.6 Escuchadores de Eventos de Servlet

El contenedor de Servlets se puede utilizar para notificar eventos a una Aplicación Web, y mediante los escuchadores de eventos de Servlets se puede actuar en consecuencia. Así, se define una interfaz de escuchador de eventos para cada evento importante que se pueda producir en relación con la Aplicación Web y los Servlets. Para que sea notificado del evento, se debe realizar una clase que implemente la interfaz de escuchador correcta, y se debe desplegar en `web.xml`. Esta clase contendrá el comportamiento que queremos que se produzca en reacción al evento, como por ejemplo llevar una cuenta del número de sesiones abiertas en la Aplicación, arrancar y parar una conexión con una base de datos, etc.

Se definen las interfaces correspondientes a los siguientes eventos:

- Ciclos de vida de las peticiones que se realizan a la Aplicación Web.
- Cambios en los atributos de la petición.
- Cambios en el contexto o el ciclo de vida de la Aplicación Web.
- Cambios en los atributos del contexto de la Aplicación Web.
- Cambios en los ciclos de vida de las sesiones HTTP.
- Cambios en los objetos almacenados en el ámbito de sesión HTTP.
- Y de cambios en la serialización de la sesión.

Así, de la clase escuchadora que se desee usar, sólo deberán implementarse los métodos según los eventos que se quieran manejar, y declarar la clase en el archivo de despliegue de la Aplicación Web.

4 JavaServer Pages (JSP)

Las Páginas de Servidor Java, o JSPs, y los Servlets son tecnologías complementarias para producir páginas Web dinámicas usando Java. Mientras que los Servlets son los cimientos sobre los que se sustenta Java en el lado del servidor, no es siempre la mejor solución en cuanto a tiempo de desarrollo. Codificar, desplegar y depurar un Servlet puede convertirse en una tarea tediosa. Encontrar un fallo gramatical de las marcas requiere leer el código que las escribe en el Servlet (normalmente múltiples llamadas `print()`), recompilar el Servlet y recargar la Aplicación Web. Y ese tipo de errores son comunes. JSP complementa a los Servlets ayudando a resolver estos problemas y simplificando el despliegue del Servlet.

4.1 Introducción a JSP

La primera especificación se terminó en 1999. Originalmente JSP se modeló como otra tecnología para crear contenido dinámico incrustado entre las marcas estáticas en el lado del servidor. Cuando se realiza una petición a una página JSP, el contenedor interpreta la JSP, ejecuta cualquier código incrustado, y envía los resultados en la respuesta. Este tipo de funcionalidad no era algo terriblemente nuevo, pero aún así fue, y aún es, una buena mejora a los Servlets.

Las JSP se han revisado en diversas ocasiones desde la salida de la versión original, añadiéndole diversas funcionalidades. Actualmente la última versión es la 2.1,

aunque la que se ha utilizado en este proyecto es la versión 2.0, ya que era la única disponible cuando se comenzó su desarrollo. De hecho, el contenedor Tomcat [51] aún no tiene una versión que soporte la nueva versión de JSP; la versión de Tomcat 6.0 que debe soportarlo está aún en desarrollo, estando sólo disponible su versión beta.

4.2 Ciclo de vida de las JSPs

Las JSP siguen un ciclo de vida de tres fases: inicialización, servicio y destrucción, tal y como se muestra en la Figura 3.5. Este ciclo de vida es idéntico al descrito para los Servlets.

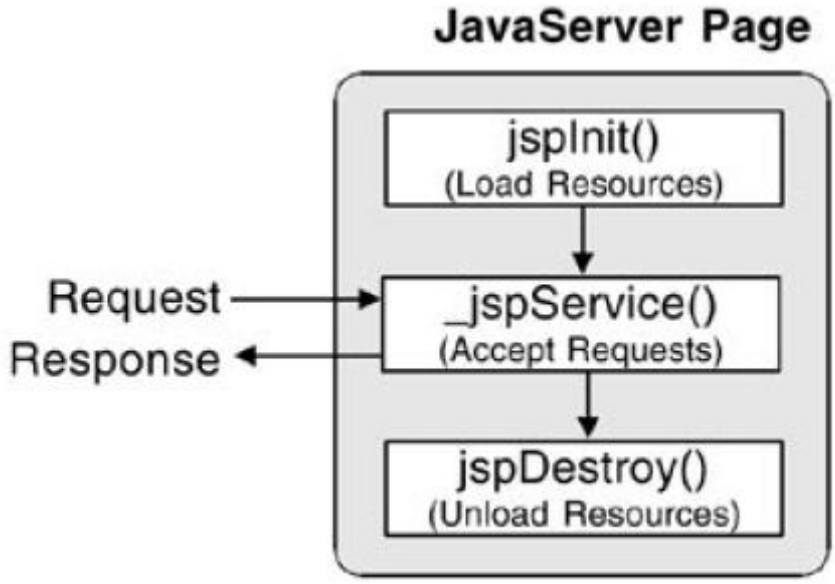

Figura 3.5. Ciclo de vida de las JSP

El ciclo de vida, aunque los nombres son distintos, se corresponde con el del Servlet, permitiendo a la JSP cargar recursos, proporcionar servicio a peticiones de múltiples clientes y destruir los recursos cargados cuando la JSP quede fuera de servicio.

Las JSPs están específicamente diseñadas para simplificar la tarea de crear objetos `HttpServlet` que produzcan texto gracias a la eliminación de las partes redundantes de la codificación de un Servlet. Todas las JSPs están diseñadas para ser usadas con HTTP [52] y generar contenido dinámico para la World Wide Web. El único método `_jspService()` de la JSP es el responsable de generar las respuestas a los siete métodos HTTP. Así, en la mayoría de los casos prácticos, un desarrollador no necesita saber nada de HTTP ni nada más que Java básico para codificar una JSP dinámica.

4.3 Crear una JSP

En realidad, una JSP no es una forma más simple de un Servlet, es un método simple para crear un Servlet productor de texto; aunque JSP y Servlet son dos tecnologías muy diferentes.

Codificar una JSP puede ser tan fácil como poner el código HTML de una página Web estática en un archivo que tenga extensión "`jsp`". Comparado con tener que estar usando los métodos de la familia `print()` continuamente, la aproximación de las JSP es evidentemente más fácil. Es por esta razón por la que las JSP simples se consideran un método rápido para crear Servlets productores de texto. Desplegar una JSP también es más simple; una Aplicación Web automáticamente despliega cualquier JSP a una URL igual que el nombre de la JSP.

En realidad, cualquier JSP es compilada en su Servlet equivalente. Este proceso se realiza en lo que se llama la *fase de traducción* del despliegue de la JSP y lo realiza el contenedor automáticamente y de forma totalmente transparente. Aunque este proceso depende del fabricante del contenedor, el ciclo de vida de una JSP siempre debe ser así. La localización de las JSP compiladas a Servlets depende del contenedor; por ejemplo, Tomcat las guarda en el directorio `/work` de la Aplicación Web. La principal diferencia entre los Servlets y las JSPs es la sintaxis que ofrecen para conseguir la misma funcionalidad. Así, las JSPs casi siempre son más útiles para crear Servlets productores de texto, pero los Servlets normales están mejor preparados para enviar flujos de bytes al cliente o cuando se necesita control total sobre el código Java fuente.

4.4 Sintaxis y Semántica JSP

Las JSPs no siguen la sintaxis y la semántica definida en la especificación Java 2. El código de las JSPs traducidas sí es sólo Java, pero al crear una JSP se siguen las reglas de la especificación JSP. Estas reglas han ido creciendo con cada nueva versión de la especificación.

4.4.1 Elementos y Datos Plantilla

Todo el contenido de una JSP se divide en dos categorías genéricas llamadas *elementos* y *datos plantilla*. Los elementos son las porciones dinámicas de una JSP. Los datos plantilla es el resto del texto en una página JSP que se va a enviar directamente al cliente.

Los datos plantilla están formados por cualquier texto plano, que no sea un elemento; así, cualquier código HTML que tenga la JSP, toda la información de la estructura, contenido estático, etc. forma parte de los datos plantilla, y se envían directamente al cliente. Los datos plantilla no cambian. Por otro lado, los elementos no se envían directamente al cliente. Un elemento se interpreta por parte del contendor JSP y define que se debe tomar una acción especial para generar una respuesta. Los elementos son los que hacen que las JSP sean dinámicas. Éstos se dividen a su vez en tres categorías diferentes: elementos de script, directivas y acciones.

4.4.1.1 Scripts

La forma más simple de hacer una JSP dinámica es incrustar trozos de código Java directamente entre bloques de datos plantilla a través del uso de elementos de script. Las JSPs no limitan los elementos de script a código Java, pero la especificación sólo habla de Java como lenguaje de script, y todos los contenedores deben soportar por defecto Java. Hay tres tipos de elementos de scripts permitidos en las JSPs: *scriptlets*, *expresiones* y *declaraciones*.

- **Scriptlets**: Los scriptlets proporcionan un método para insertar directamente trozos de código Java entre trozos de datos plantilla. Se define con un comienzo, `<%`, y un final, `%>`, con el código Java en medio. Usando Java, el script es idéntico al código Java normal pero sin la necesidad de declaración de clase ni de método. Los scriptlets son ideales para proporcionar funcionalidad de bajo nivel como iteración, bucles y declaraciones condicionales, pero también proporciona un método para incrustar bloques complejos de código en una JSP. Aunque cuantos más trozos de código Java complejos haya en una JSP más difícil será comprenderla y mantenerla. Los scriptlets no se envían al cliente; sólo se envía el resultado del scriptlet.

- **Expresiones**: Proporcionan un método fácil para enviar cadenas dinámicas al cliente. Una expresión comienza con `<%=` y termina con `%>`, con la expresión en medio. Una expresión siempre envía una cadena al cliente, pero el objeto producido como resultado de una expresión no tiene que ser necesariamente de tipo `String`. Automáticamente se llama al método `toString()` del objeto resultado de una expresión, y si el resultado es un tipo primitivo, se representa su valor como una cadena.

- **Declaraciones**: Una declaración se usa como un scriptlet para incrustar código en una JSP, pero el código incrustado aparecerá fuera del método `_jspService()`. Así, las declaraciones se pueden usar para declarar nuevos métodos y variables de clase globales. Una declaración comienza con `<%!` y termina con `%>`.

Hay que tener cuidado al utilizar los scriptlets y las declaraciones, ya que no tienen seguridad de hilo. Las expresiones no tienen este problema.

El poder de las JSPs viene de la facilidad en la creación de Servlets productores de texto. Una JSP tiene un fácil mantenimiento si principalmente tiene marcas, que son fácilmente editables por los autores de páginas. Muchos desarrolladores sostienen la postura de que los elementos de script destruyen esta propiedad de las JSP y que deberían ser sustituidas por *acciones personalizadas* y por el *lenguaje de expresión de JSP* (nuevo en JSP 2.0) (ambas se verán posteriormente). Aunque las acciones personalizadas son más pesadas que un elemento de script, un exceso de estos elementos puede hacer que la JSP sea ilegible.

4.4.1.2 Directivas

Las directivas son mensajes al contenedor JSP. No mandan nada al cliente, pero se usan para definir atributos de la página, qué bibliotecas de etiquetas personalizadas incluir y qué otras páginas incluir. Usan la siguiente sintaxis,

```
<%@ directiva {atributo="valor"}* %>
```

Pudiendo contener espacios en blanco adicionales. Hay tres tipos diferentes de directivas que se pueden usar en una página: `page`, `taglib`, e `include` (y otras tres que sólo se pueden usar en archivos de etiquetas: `tag`, `attribute`, y `variable`).

- **`<%@ page %>`**: Esta directiva proporciona información específica de la página al contenedor JSP, incluyendo configuraciones como el tipo de contenido que va a producir la JSP, el lenguaje de script de la página, y bibliotecas de código a importar, entre otras. Todo esto se realiza mediante distintos atributos de la directiva.

• **<%@ include%>** y **<jsp:include/>**: Esta directiva se usa para incluir texto y/o código en tiempo de traducción de la JSP. Siempre sigue la misma sintaxis, `<%@ include file="URLrelativa" %>`, con la URL relativa del archivo a insertar. Los archivos a incluir deben formar parte de la Aplicación Web. Como tiene lugar en tiempo de traducción, es equivalente a incluir el código fuente directamente en la JSP antes de su compilación y no da lugar a pérdida de rendimiento en su ejecución. Mediante esta directiva se puede incluir el mismo trozo de código repetitivo colocado en un archivo a parte en muchas páginas.

Para incluir cosas que no sea en tiempo de ejecución JSP también define la acción `jsp:include` (las acciones se verán posteriormente) Esta acción es similar a la directiva, pero tiene lugar durante la ejecución. Aunque no es eficiente, la acción se asegura de que se incluye la última versión del archivo. Esto puede ser útil si el archivo a incluir cambia a menudo.

• **<%@ taglib %>**: Esta directiva informa al contenedor sobre qué parte de las marcas de la página debe ser considerada como código personalizado (acciones personalizadas, se verán posteriormente) y dónde encontrar ese código. Su sintaxis es siempre la misma, `<%@ taglib uri="uri" prefix="prefijoDeLaEtiqueta" %>`, donde `uri` representa una localización que entiende el contenedor y `prefix` informa de cuáles de las marcas son acciones personalizadas asociadas a esa localización.

4.4.1.3 Acciones JSP

Las acciones proporcionan un método conveniente para enlazar código con unas simples marcas que aparecen en una JSP. La funcionalidad es idéntica a los elementos de script pero con la ventaja de abstraer completamente cualquier código (que iría en el script) que normalmente se mezclaría con el resto de la JSP. De esta forma ayudan a facilitar el mantenimiento y la eficiencia de una JSP. Hay dos tipos de acciones disponibles para una JSP: estándar y personalizadas, aunque todas siguen la misma sintaxis, `<prefijo:elemento {atributo="valor"}* />`, que es compatible con una etiqueta XML [4]. Las acciones estándar están definidas en la especificación JSP y están disponibles para su uso con cualquier contenedor JSP. Las acciones personalizadas es un mecanismo definido en la especificación JSP para mejorarlas permitiendo a los desarrolladores de JSP crear sus propias acciones.

Las acciones estándar incluyen funcionalidades comúnmente usadas en las JSP y permiten: usar fácilmente Applets de Java, incluir archivos en tiempo de ejecución, manipular JavaBeans [13], enviar peticiones a otros recursos de la Aplicación Web y ayudar a la ejecución de las etiquetas personalizadas.

4.4.2 Dos tipos de sintaxis

Los contenedores JSP deben soportar dos tipos de sintaxis: normal y compatible con XML [4]. La normal está pensada para ser fácil para el autor. La compatible con XML toma la normal y la modifica para que sea conforme con XML, y está pensada para que sea más fácil su uso con herramientas de desarrollo. La compatible con XML se introdujo en la versión 1.2 de JSP, y desde el punto de vista del desarrollador es mucho más engorrosa y no merece la pena, ya que es más compleja. La versión 2.0 de JSP remedió algo el problema proporcionando una sintaxis XML más flexible, aunque

sigue siendo para el desarrollador mucho más complejo y tedioso hacer una JSP compatible con XML.

4.4.3 Configuración de JSP

Las directivas son la forma más fácil de configurar una JSP. Pero tienen el problema de que se deben especificar para todas las JSP de la Aplicación. Para simplificarlo, está disponible el elemento `jsp-config` para usarlo en `web.xml`. Hay dos subelementos: `taglib` y `jsp-property-group`. El primer elemento se usa para configurar una biblioteca de etiquetas personalizadas para su uso en una JSP. La segunda permite la configuración de forma similar a las directivas, pero se puede aplicar a un grupo de JSP.

4.5 Objetos Implícitos

JSP tiene disponibles una serie de *objetos implícitos* que le permiten fácilmente manipular la petición, la respuesta o la sesión. Los siguientes objetos están siempre disponibles para usar en elementos de script ya que se declaran automáticamente en una JSP y tienen su equivalente en el Servlet:

- **`config`**: Al igual que con los Servlets, se pueden declarar parámetros iniciales a las JSPs mediante el descriptor de despliegue de la Aplicación Web, y acceder a ellos mediante este objeto, instancia de `javax.servlet.ServletConfig`.
- **`request`**: Representa la petición del cliente, y hace referencia al objeto pasado al método del Servlet que sirve esa petición. Es una instancia de `javax.servlet.http.HttpServletRequest`.
- **`response`**: Representa la respuesta para el cliente y hace referencia al objeto pasado al método del Servlet que debe enviar esa respuesta. Es una instancia de `javax.servlet.http.HttpServletResponse`.
- **`session`**: Es un objeto que representa el contexto de la sesión de cada cliente. Es una instancia de `javax.servlet.http.HttpSession`, y es equivalente al objeto devuelto por la llamada a `HttpServletRequest.getSession()`.
- **`application`**: Representa la visión del Servlet de la Aplicación Web. Es una instancia de `javax.servlet.ServletContext` y es equivalente al objeto devuelto por la llamada al método `ServletConfig.getServletContext()`.

A continuación se muestran algunos objetos implícitos más que no tienen su equivalente en el Servlet:

- **`pageContext`**: Representa el contexto de una única JSP, incluyendo todos los demás objetos implícitos, métodos para incluir y enviar a otros recursos de la Aplicación Web, y ámbito para almacenar objetos de la página. La principal utilidad es que contiene referencias a los demás objetos implícitos y así es útil como parámetro para otro método. Es una instancia de `javax.servlet.jsp.PageContext`.
- **`out`**: Es una instancia de `javax.servlet.jsp.JspWriter` y proporciona un método conveniente para escribir texto con búfer.
- **`exception`**: Objeto sólo disponible cuando la página se ha definido como una página de error.

5 Excepciones y Errores en Servlets y JSP

Siempre que se lance una excepción desde un Servlet o una JSP, ésta será pasada al contenedor. Lo que haga éste con ella depende de cada contenedor, pero, usualmente, el contenedor trata de informar del error mediante un mensaje seguido de la traza de la pila de la excepción. Aunque toda esta información puede no ser adecuada para el usuario de la Aplicación Web, y, de hecho, normalmente no lo será.

Hay dos formas de manejar los errores, considerando toda la aplicación, o elemento a elemento, cada una con sus ventajas.

5.1 Administración de las excepciones individualmente

Las excepciones se pueden controlar en los Servlets y las JSPs mediante el uso de las sentencias Java `try-catch-finally` libremente. Pero además, las JSP pueden usar la directiva `page` para especificar una página a la que pasar las excepciones que no sean capturadas. Mediante su atributo `errorPage` se puede asignar una URL que puede representar una JSP o un Servlet especialmente diseñado para ser una página de error. Una JSP diseñada para ser una página de error puede establecer el valor del atributo `isErrorPage` de la directiva `page` a "`true`"; con esto se consigue que la variable de script `exception` esté automáticamente disponible para representar a la excepción pasada.

Así se puede asegurar que las trazas de error y los mensajes específicos nunca alcanzarán al usuario. La JSP diseñada para ser una página de error se puede realizar de forma que se adecue al resto de la Aplicación Web y sea más "agradable" al usuario.

En la página de error definida se puede tener acceso a la instancia del objeto `Throwable`, que está en ámbito de la petición bajo el nombre de `javax.servlet.jsp.jspException`, aunque si la página es una JSP definida como página de error, automáticamente tendrá disponible la variable de script implícita `exception` referenciando a ese mismo objeto `Throwable` de la excepción.

5.2 Administración global de las excepciones

Para controlar las excepciones de toda la Aplicación Web se debe hacer uso del archivo `web.xml`, modificando el comportamiento del manejador de excepciones por defecto de la Aplicación Web. Se definen las páginas de error para toda la Aplicación en el descriptor de despliegue de la aplicación, `web.xml`, mediante el elemento `error-page`, definiendo manejadores de error en función de la excepción lanzada o el código de error HTTP producido por la respuesta. El elemento `error-page` contiene subelementos que especifican la URL relativa de la página de error y o bien el código HTTP de error de respuesta o bien el tipo de `Throwable` que está asociado con la página de error (una clase Java cualificada).

6 Lenguaje de Expresión de JSP 2.0

El lenguaje de expresión de JSP, comúnmente conocido como JSP EL (JSP Expression Language), es superior a las expresiones de JSP ya que proporciona una

sintaxis clara y un lenguaje especialmente diseñado para JSP. Además, funciona bien con las JSPs en sintaxis XML, ya que el JSP EL no requiere el uso de caracteres que forman parte de las marcas XML.

Además, el JSP EL se puede usar en cualquier lugar de una JSP. Esto hace que sea una alternativa simple y poderosa a los elementos de script.

6.1 Sintaxis del JSP EL

El JSP EL está diseñador para ser simple, robusto, y con mínimo impacto con las versiones anteriores. El EL maneja expresiones y literales. Las expresiones siempre están encerradas en los caracteres `${ }`. Por ejemplo:

```
<c:if test="${value < 10}" />
```

Los valores que no empiezan con "`${`" se tratan como un literal. El valor del literal se convierte al tipo esperado dependiendo de la entrada en el TLD (Tag Library Descriptor, Descriptor de la Biblioteca de Etiquetas, descrito posteriormente) de la etiqueta:

```
<c:if test="true" />
```

En los casos en los que el valor del literal contiene los caracteres "`${`", se deben poner los caracteres de escape "`$\{`".

El JSP EL proporciona un método intuitivo, que no es Java, para escribir expresiones y que permite un lógica simple y acceso a las variables de cualquier ámbito. Mediante su uso se pueden evitar las expresiones JSP logrando un diseño más limpio y claro.

Ya que antes de JSP 2.0 la sintaxis `${ }` no estaba reservada, para evitar incompatibilidades con versiones anteriores, el EL se puede deshabilitar en una página usando una directiva de página, o configurarse para toda la aplicación mediante una entrada de configuración de las JSPs en `web.xml`.

6.1.1 Atributos

A los atributos en el EL se accede por su nombre, añadiendo opcionalmente un ámbito. Los miembros, los métodos "getter" y las tablas se acceden de la misma forma, mediante un '.', que pueden estar anidados sin límite de profundidad. Aunque también se puede seguir accediendo a ellos mediante las llamadas a métodos o los corchetes en el caso de las tablas. El EL busca el identificador que tiene que evaluar en los distintos ámbitos disponibles en el orden siguiente: página (`page`), petición (`request`), sesión (`session`), y aplicación (`application`), y si no se encuentra, devuelve `null`.

Además, el EL define una serie de objetos implícitos que permiten acceder a objetos de cualquier ámbito, obtener un acceso a los `String` de los parámetros y sus valores recibidos en la petición, leer los datos de la cabecera de la petición, y acceder a las cookies del sistema.

6.1.2 Literales

El EL define los literales *booleanos*, *long* (como en Java), *float* (como en Java), *String* (como en Java), y *null* (como en Java).

6.1.3 Operadores

El EL proporciona los operadores básicos aritméticos típicos (como en Java), operadores lógicos básicos correspondientes a los de Java, y soporte para comparación idéntico a Java, que además permite el uso de las abreviaturas `lt`, `gt`, `lte`, o `gte`. También se define un orden de preferencia de los operadores como en Java.

7 Biblioteca Estándar de Etiquetas de JSP

La especificación de la JSTL (JavaServer Pages Standard Tag Library, Biblioteca Estándar de Etiquetas de JSP) 1.1.2 es una Petición de Especificación Java (JSR) oficial realizada a través del Proceso de la Comunidad Java (JCP [14]) de Sun. La versión 1.0 de este juego extendido de acciones estándar apareció con la versión 2.0 de JSP.

Usar las bibliotecas estándar de etiquetas es simple, sólo hay que importarlas en las páginas JSP mediante la directiva `taglib`. Por ejemplo, para importar la biblioteca JSTL "core" en una página, se debe incluir la línea siguiente al comienzo de la página JSP:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

A pesar de que la funcionalidad que realiza JSTL no es nueva, y las acciones que realizan se pueden hacer mediante elementos de scripts y expresiones, su uso es muy aconsejable. JSTL es estándar, y es una buena implementación de muchas funciones útiles. Al ser estándar no se tienen que aprender nuevas formas de realizar dichas funciones o diseñar nuevas etiquetas que las realicen.

A continuación se comentan las bibliotecas estándar de JSTL, extendiéndose más en las utilizadas en el proyecto.

7.1 Etiquetas Core

El juego de etiquetas "core" incluye etiquetas muy útiles. Todas comparten la misma URI, `http://java.sun.com/jsp/jstl/core`, y se pueden dividir en varios grupos.

7.1.1 Etiquetas de propósito general

Usadas para tareas simples, y consiste en las acciones:

- **`<c:out>`**: evalúa una expresión y envía el resultado a la salida de la página. En JSP 2.0 es irrelevante pues poniendo una EL directamente se consigue el mismo efecto.
- **`<c:set>`**: Evalúa una expresión y el resultado lo establece en un JavaBean [13] o un objeto `java.util.Map`, o establece una variable en un determinado ámbito.
- **`<c:remove>`**: Elimina una variable de algún ámbito.

- **<c:catch>**: Permite capturar una excepción evitando que termine de ejecutarse el resto de la JSP.

7.1.2 Etiquetas de iteración

Permiten iterar sobre colecciones de objetos, permitiendo evitar la inclusión de bucles Java en un scriptlet. Al eliminar los scriptlets a favor de las etiquetas personalizadas aumenta la cantidad de etiquetas de la página y disminuye el código Java, dando como resultado una JSP más fácil de leer. Así la página será más fácil de interpretar y mantener, sobre todo si van a ser varios los desarrolladores que van a tener que hacerlo, permitiendo que incluso sea comprensible por personas no acostumbradas a Java y sí a los lenguajes de marcas.

Sólo hay dos tipos de etiquetas de iteración, y son las siguientes.

- **<c:forEach>**: Proporciona iteración sobre una colección de objetos, en particular sobre una tabla, `java.util.Collection`, `java.util.Iterator`, `java.util.Map`, o un `java.util.Enumeration`, exponiendo cada elemento o primitiva de la colección para su uso en el cuerpo de la etiqueta. Se puede usar con dos sintaxis distintas:

Sintaxis 1: Itera sobre una colección de objetos

```
<c:forEach [var="nombreVariable"] items="coleccion"
            [varStatus="nombreVariableDeEstado"]
            [begin="comienzo"] [end="fin"] [step="paso"]>
      contenido del cuerpo
</c:forEach>
```

Sintaxis 2: Itera un número fijo de veces

```
<c:forEach [var=" nombreVariable"]
            [varStatus=" nombreVariableDeEstado"]
            begin="comienzo" end="fin" [step="paso"]>
      contenido del cuerpo
</c:forEach>
```

- **<c:forTokens>**: Lee un `String` y lo divide en trozos en función de un delimitador especificado, y se itera sobre cada uno de los trozos de forma similar a `forEach`.

7.1.3 Etiquetas Condicionales

Soporte de declaraciones simples como los `switch` e `if` de Java. Permite sustituir los scriptlets que se necesitan comúnmente para realizar esa misma función, permitiendo un código de marcas más limpio en la JSP. Hay dos grupos de etiquetas condicionales:

- **<c:if>**: Proporciona ejecución de su cuerpo en función de que su atributo "`test`" se evalúe a `true` o no.
- **<c:choose>**, **<c:when>**, y **<c:otherwise>**: La etiqueta `choose` realiza ejecución condicional de bloques de subetiquetas `when`. Evalúa el cuerpo de la primera etiqueta `when` cuya condición se evalúe a `true`. Si ninguna condición se cumple, se evalúa el cuerpo de la etiqueta `otherwise`, si está presente.

7.1.4 Manipulación de URL

Proporcionan un soporte de manipulación de URL [20] más robusto que el de JSP por defecto.

- **<c:import>** y **<c:param>**: `import` proporciona la funcionalidad de la acción `include` pero permitiendo además la inclusión de URL absolutas. Importa una URL que se puede enviar a la salida estándar de la JSP o a un Reader indicado por los atributos. La etiqueta `param` complementa a `import` permitiendo pasarle parámetros a la URL especificada, en forma de pares nombre, valor.
- **<c:url>**: Codifica URLs con información y parámetros de sesión. Se utiliza cuando no se pueden utilizar cookies.
- **<c:redirect>**: Codifica una redirección del lado del cliente. Es equivalente a llamar al método `HttpServletResponse sendRedirect()`, pudiendo tener etiquetas hijas `param` para especificar parámetros.

7.2 Formateo de Texto con Capacidad de Internacionalización

Colección de etiquetas para formatear texto diseñadas para sitios Web con internacionalización (i18n), implementación de una de las soluciones más comunes usadas en i18n.

7.3 Manipulación XML

Proporciona una forma flexible de manipular XML. Puede ser muy útil para crear XML para tecnologías como Servicios Web. Proporciona algunas funciones de manipulación XML y el equivalente XML de las etiquetas "core" y condicionales de JSTL. Se dividen en tres grupos:

- **Etiquetas XML core**: Etiquetas equivalentes a las JSTL "core", pero con soporte para EL y XPath [15].
- **Etiquetas de Control de Flujo**: Implementación XPath de las etiquetas `choose`, `when`, `otherwise`, `if` y `forEach`.
- **Etiquetas de Transformación**: Proporciona una forma de aplicar transformaciones XSLT [16] a contenido XML.

7.4 Etiquetas SQL

Proporcionan una interfaz simple para ejecutar peticiones SQL a una base de datos a través de una JSP.

8. Bibliotecas de Etiquetas Personalizadas

Las etiquetas personalizadas y los elementos de script realizan la misma funcionalidad. Los scripts incrustan código directamente con fragmentos de marcas estáticas. Y, por el contrario, las etiquetas personalizadas proporcionan un método para separar limpiamente la lógica del contenido. Un scriptlet mezcla código de modificación de datos y el código para representar los datos. Esto provoca un aumento de la complejidad de la JSP. Sin embargo, utilizando etiquetas personalizadas, la lógica se puede modificar sin tener que tocar la JSP.

También las etiquetas personalizadas son muy fáciles de usar tanto por programadores como por no programadores. Incluso un diseñador HTML sin experiencia con Java puede fácilmente utilizarlas. Por eso las etiquetas personalizadas

son una buena forma de abstraer una lógica compleja en una capa de presentación sencilla en lenguaje HTML o XML.

Otra característica es la portabilidad de las etiquetas. Se pueden empaquetar en un archivo JAR [42] y desplegarlo en muchas Aplicaciones Web. Así se permite reutilizar el código, otra gran ventaja comparada con scriptlet.

Una buena forma de pensar en una etiqueta personalizada es que es un componente, escrito en Java o JSP, que encapsula algún comportamiento. Para el autor de la página, la etiqueta parece HTML, o, al menos, XML. Para un desarrollador de etiquetas, ésta es o una clase Java o un fragmento de JSP.

Con la versión de JSP 2.0 se introdujo una nueva forma de escribir etiquetas personalizadas, que permite que se escriban utilizando código Java (como siempre) o como una JSP (la forma nueva), y que son más simples que las antiguas etiquetas personalizadas. Aunque ambos tipos se pueden combinar de cualquier manera en la JSP, las antiguas y las nuevas.

8.1 Descriptores de Bibliotecas de Etiquetas (Tag Library Descriptors, TLDs)

Un descriptor de biblioteca de etiquetas es un mecanismo que relaciona el código de una etiqueta con las marcas simples que aparecen en una JSP. Concretamente, es un archivo XML que aparece normalmente en el directorio `/WEB-INF` de la Aplicación Web con la extensión "`tld`". También se puede encontrar empaquetado con el resto de la biblioteca de etiquetas en el JAR.

Un ejemplo de TLD puede ser el siguiente:

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
          http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
          version="2.0">
     <tlib-version>1.0</tlib-version>
     <short-name>Util TLD</short-name>
     <jsp-version>2.0</jsp-version>
     <uri>http://www.QTI.us.es/util</uri>

     <tag>
          <name>adminMail</name>
          <tag-class>utilidades.etiquetas.util.AdminMailTag</tag-class>
          <body-content>empty</body-content>
     </tag>

     <tag>
          <name>reqURI</name>
          <tag-class>utilidades.etiquetas.util.RequestedURITag</tag-class>
          <body-content>empty</body-content>
     </tag>

     <tag>
          <name>logger</name>
          <tag-class>utilidades.etiquetas.util.logging.LoggerTag</tag-class>
          <body-content>empty</body-content>
          <attribute>
               <name>nivel</name>
               <required>true</required>
```

```
            <type>java.lang.String</type>
        </attribute>
        <attribute>
            <name>mensaje</name>
            <required>true</required>
            <type>java.lang.String</type>
        </attribute>
    </tag>
</taglib>
```

No es más que una colección de elementos `tag`. Así, para cada una se informa de su nombre (que debe ser único dentro de las etiquetas de la biblioteca), la clase Java que maneja la llamada (un nombre completo de una clase Java existente), el tipo de código que se permite dentro del cuerpo de la etiqueta (vacío, JSP, sin scripts o a interpretar por la etiqueta), y una descripción de los atributos de la etiqueta, si los tiene.

Antes de las declaraciones de las etiquetas de la biblioteca y tras el elemento raíz `taglib`, se pueden encontrar la declaración de la versión de JSP, de la biblioteca de etiquetas, el nombre corto del juego de etiquetas personalizadas, y una URI [9] ficticia definida para hacer referencia a la biblioteca.

8.1.1 Usando el descriptor de la biblioteca de etiquetas

Se debe referenciar la TLD a utilizar mediante la directiva `taglib` en la JSP utilizando el atributo `uri`. Esta URI puede ser una relativa a la localización real de la TLD, aunque es poco recomendable ya que cualquier cambio en la localización de la TLD o la JSP hará que se tenga que cambiar también el código de todas las JSP que referencien a la TLD; además, si la etiqueta se empaqueta en un archivo JAR esa URI es inútil.

Una forma más adecuada de realizar la referencia es definir una URI abstracta para toda la aplicación, y hacerla corresponder a esa TLD. Esta correspondencia se realiza en el descriptor de despliegue de la Aplicación Web, `web.xml`. Así, una JSP que quiera utilizarla sólo tiene que referenciar esta URI abstracta, que se encarga de resolver la Aplicación Web. No hay ningún problema si se cambia de sitio la JSP, todas las JSPs usan la misma URI, y si se cambia la posición de la TLD sólo hay que modificar el descriptor de despliegue de la Aplicación Web. Aunque para ahorra aún más trabajo, se puede añadir a la propia TLD, antes de las declaraciones de las etiquetas, una entrada `<uri>`, donde se define una URI abstracta para la TLD. Así, sólo haciendo referencia a estas URIs y poniendo el archivo TLD en el directorio `/WEB-INF` o algún subdirectorio de éste, o, si está en un JAR, dentro del directorio de JARs en el directorio `/META-INF` (ver especificación de los archivos JAR [42]), ya estará disponible para las JSPs con sólo referenciar su URI abstracta.

8.2 Etiquetas Personalizadas Simples de JSP 2.0

Implementan la interfaz `javax.servlet.jsp.tagext.SimpleTagSupport` que sigue el ciclo de vida simple presentado en la Figura 3.6. Todas las etiquetas simples deben implementar esta interfaz. El ciclo de vida tiene dos partes: en la inicialización se establecen el cuerpo de la etiqueta y cuál es la superior (para permitir colaboración entre etiquetas anidadas); y el método `doTag()` se invoca cuando entra en acción la etiqueta, y es el único método de interés y por el que se tiene que preocupar el desarrollador.

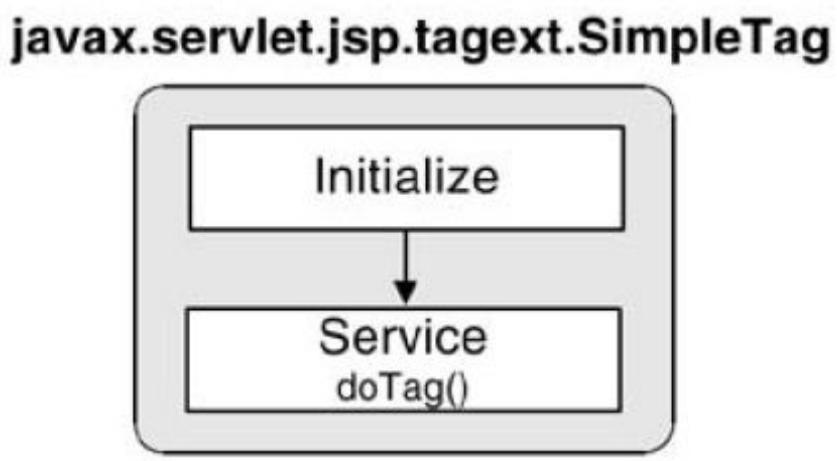

Figura 3.6. Ciclo de vida de una Etiqueta Simple

Al implementar el método `doTag()`, se puede acceder al `JspWriter` para la página, acceder a la petición y a la respuesta, a la etiqueta padre, si la hubiera, etc. Así, se permite declarar atributos de las etiquetas, evaluar su cuerpo e iterar sobre él, expandiendo y convirtiendo a las etiquetas personalizadas en una gran herramienta para realizar muchos procesos complejos que quedarán "ocultos" en la página JSP que use la etiqueta.

8.3 Archivos `.tag` para hacer etiquetas personalizadas

Con la versión de JSP 2.0 se introdujeron los archivos `.tag`, que permiten prescindir de Java para hacer etiquetas personalizadas. Estas etiquetas permiten que se escriban en JSP, permitiendo que el autor no tenga que saber programar Java. Así, estas etiquetas son más útiles para las que deben contener muchos datos de salida para la JSP, ya que no hace falta poner multitud de `out.print(...)`.

El archivo es un archivo fuente JSP con algunas directivas adicionales y algunas acciones más disponibles. Para usar la etiqueta sólo hay que colocar el archivo `.tag` en el directorio `/WEB-INF/tags`, o algún subdirectorio de éste. Y para usarlo en la JSP sólo hay que referenciarlo mediante el atributo `tagdir` de la directiva `taglib`. Igualmente, se pueden incorporar a un TLD para formar parte de la biblioteca, y empaquetarse dentro de un archivo JAR.

El archivo debe ser configurado mediante una serie de directivas disponibles. Las directivas `taglib` e `include` son idénticas a las de las JSPs. Además incluye directivas para definir el nombre y el tipo de contenido de la etiqueta; definir los atributos que acepta la etiqueta, sus tipos, si pueden ser dinámicos, su descripción, etc.; y definir variables para el archivo. También incluye una acción de JSP que sólo se puede utilizar en estos archivos para ejecutar el cuerpo de la etiqueta.

8.4 Manejadores de Etiquetas Clásicas de JSP

Estas etiquetas son las que existían antes de la versión 2.0 de JSP, y aún están disponibles. Hay tres tipos de etiquetas: "básicas", de "iteración", y de "cuerpo", cada

una representada por las interfaces `javax.servlet.jsp.Tag`, `javax.servlet.jsp.IterationTag`, y `javax.servlet.jsp.BodyTag` respectivamente. Cada una sigue un ciclo de vida y presenta unos métodos que se deben implementar representando sus ciclos de vida. Son interfaces mucho más complejas que la nueva etiqueta simple de JSP 2.0, y realizan la misma función.

9 Filtros

Los Filtros son componentes que se sitúan entre la petición y el punto final de esa petición (recurso dinámico o estático de cualquier tipo, existente o no). Un Filtro es capaz de:

- Leer la petición antes de llegar al punto final.
- Envolver los datos de la petición antes de pasarlos.
- Envolver los datos de la respuesta antes de devolverla.
- Manipular los datos de respuesta antes de devolverlos.
- Devolver errores al cliente.
- Enviar la petición a otro lugar e ignorar la URL original.
- Generar su propia respuesta antes de enviarla al cliente.

9.1 ¿Qué es un Filtro?

Físicamente un Filtro es un componente que intercepta la petición enviada a un recurso de la Aplicación Web. Forman una cadena, siendo el último eslabón el recurso solicitado. El Filtro puede elegir pasar la petición al siguiente eslabón de la cadena o no, y también ve la respuesta antes de que sea devuelta al cliente. Este comportamiento se puede ver en la Figura 3.6.

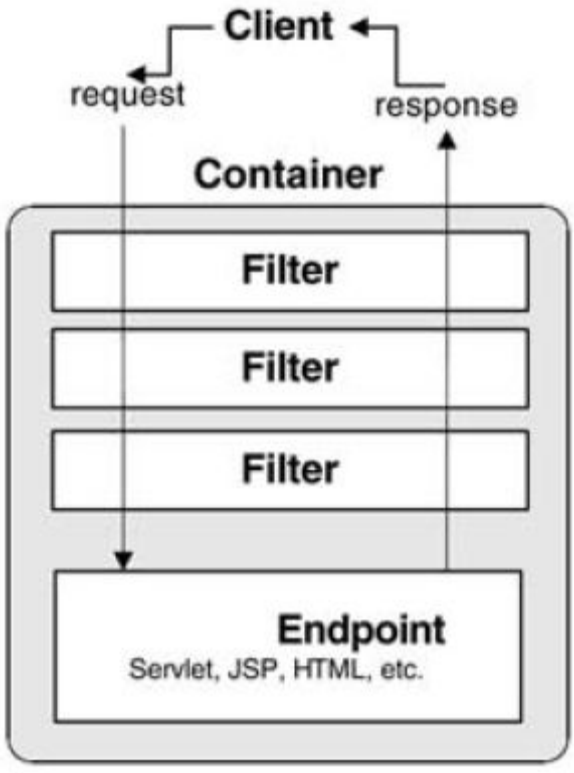

Figura 3.7. Paso de la petición por los Filtros hasta el punto final

De esta forma permiten añadir cualquier número de capas de preprocesado y postprocesado a la petición y la respuesta de una forma mucho más adecuada y fácil que con Servlets.

9.2 Ciclo de Vida de un Filtro

Conceptualmente es igual al del Servlet (ver Figura 3.3): inicialización (cuando se carga el Filtro para ser usado), servicio (cuando se aplica el Filtro a una petición y respuesta) y destrucción(al descargar el Filtro y dejarse de usar). La diferencia es que el método que corresponde con la fase de servicio de llama `doFilter()`, aunque también recibe la petición y la respuesta del cliente.

9.3 Codificar un Filtro

Un Filtro debe implementar la interfaz `javax.servlet.Filter`, que define su ciclo de vida, definiendo tres métodos: `void init(FilterConfig config) throws ServletException`, correspondiente con la inicialización del Filtro que se ejecuta con su carga y sólo una vez; `void destroy()`, que se ejecuta al descargar al filtro, y que cierra los recursos inicializados por el Filtro; y `void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException`, donde se hace todo el trabajo del Filtro.

La petición y la respuesta, en el caso de que sean instancias de unas HTTP, deberán convertirse a ese tipo de objetos para tener acceso a todos los métodos disponibles. También se incluye el objeto `FilterConfig` (usado para configurar el Filtro en la inicialización) y el objeto `FilterChain` (que representa la actual cadena de Filtros que se aplica a una petición y respuesta dadas).

Una vez compilado el código, antes de poder usarlo, el Filtro debe ser desplegado en la Aplicación Web. Su despliegue es similar al de un Servlet, pero usando el elemento `filter` para declarar el filtro, y con los subelementos `filter-name` para el nombre del filtro, y `filter-class` para indicar la clase Java del Filtro. Y el mapeo se realiza mediante el elemento `filter-mapping`, y con los subelementos `filter-name` se referencia un Filtro ya declarado, y con `url-pattern` se especifican las URL [20] de las peticiones que el Filtro "interferirá". Así se configura cada Filtro para que actúe siempre que la URL requerida sea la definida, pudiéndose utilizar caracteres comodines para que actúe con más de un destino, pudiendo quedar una configuración como se ve en la Figura 3.8 de tres Filtros (**Fx**) con tres destinos (**Dx**).

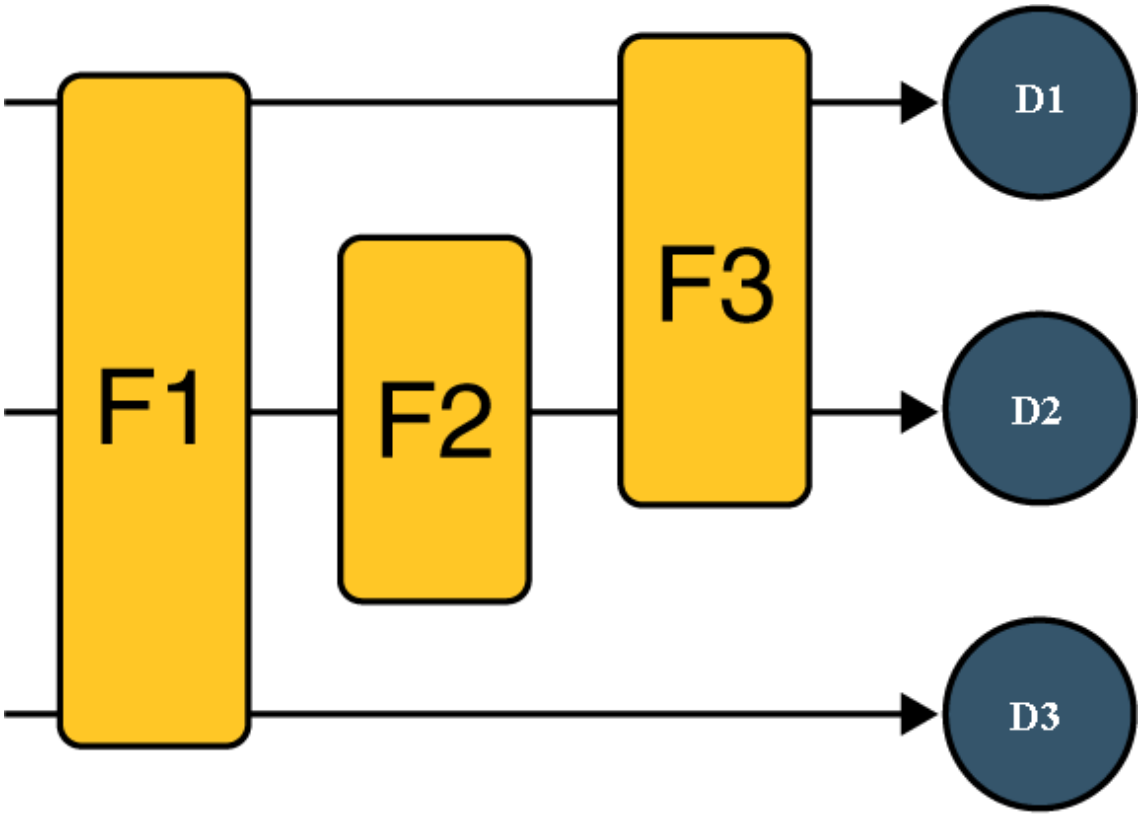

Figura 3.8. Mapeo de Filtros a Destinos

La cadena de Filtros es lo que hace a los Filtros diferentes de los Servlets. El objeto `FilterChain` representa la pila de Filtros que se ejecutan para una determinada petición y respuesta, y permite una separación en capas de las tareas que se le deben aplicar. Si se deben aplicar varios Filtros, el orden de aplicación será el orden ascendente de los elementos `filter-mapping` definidos en `web.xml`. Cada Filtro puede decidir continuar la ejecución de los siguientes de la cadena o no.

9.4 Envoltorios

Los Filtros tienen la capacidad de envolver una petición y/o una respuesta, consiguiendo así encapsular una petición o una respuesta en otra (personalizada). Personalizando un envoltorio se consigue que la petición o la respuesta no se comporte de forma normal. Los envoltorios no son específicos de los Filtros, pero junto con ellos se consigue un gran beneficio.

Colocando un envoltorio a una petición o una respuesta se puede conseguir un preprocesado o postprocesado de los datos: por ejemplo, un sistema para llevar un registro del contenido de las peticiones y las respuestas, un mecanismo de compresión de la información a enviar al cliente o de caché de páginas para no tener que volver a recalcular páginas iguales a enviar al cliente.

9.4.1 Codificación de un Envoltorio

Un envoltorio es solo una implementación del objeto que está siendo envuelto. Por ejemplo, la API Servlet proporciona las clases `ServletRequestWrapper` y

`HttpServletRequestWrapper` para hacer subclases que serán los envoltorios con la nueva funcionalidad que se quiera que realice. Para crear un envoltorio personalizado nuevo se extiende la clase apropiada y personaliza el código de los métodos cuyo comportamiento se desee modificar.

10 Estado de la Sesión en la Aplicación Web

El protocolo HTTP [52] no mantiene una conexión continua con el servidor, es decir, no mantiene información de sesión con el servidor. Sin embargo, muchas Aplicaciones Web deben seguir el rastro del usuario. Esto se consigue intercambiando cliente y servidor un identificador único, creado por el servidor, que lo identifique entre los demás. Este intercambio se puede producir de tres formas: en la cabecera HTTP (mediante cookies), datos extra añadidos a la URL [20] (reescritura de URL), o añadiendo al formulario de la página algún campo oculto con este valor. Los dos primeros los soporta la especificación Servlet, el tercero no (sería a nivel de aplicación).

Mediante la clase `javax.servlet.http.HttpSession`, y algunas APIs, la aplicación controla el estado del cliente. Esta clase mantiene el identificador del cliente y los datos asociados con él, pudiéndose acceder a cualquiera de ellos en cualquier momento. Los datos se guardan con un nombre, que será un `String`, y un valor, que es un objeto Java. Estos datos son únicos y privados para el cliente, y perdurarán hasta que se destruya la sesión del cliente.

Para mantener el estado mediante las cookies, la API Servlet abstrae los detalles técnicos de forma que no se tenga que preocupar el programador para un uso normal. Aún así, contiene métodos para acceder a ellas y modificarlas según convenga.

Cuando las cookies fallan, se utiliza la reescritura de URL que permite llevar el seguimiento de la sesión con HTTP, reescribiendo la URL original para que tenga también el identificador de la sesión. Se puede hacer de distintas formas, siempre que el servidor entienda la nueva URL, por ejemplo, añadiendo nuevos parámetros. Lo malo es que se deben reescribir siempre todas las URLs y diferentes para cada usuario.

Mediante escuchadores, la API Servlet permite inicializar recursos que van a hacer falta durante la duración de la sesión, y terminarlos al finalizar ésta, como puede ser una conexión con una Base de Datos, al crearse y destruirse el objeto `HttpSession`.

11. Paquete Commons FileUpload

El paquete `org.apache.commons.fileupload` [41], de Jakarta [22], es una forma fácil de añadir una robusta capacidad de subida de archivos al servidor y de alto rendimiento en el caso de uso de Servlets y Aplicaciones Web.

Sus métodos procesan una petición HTTP [52] que se ajuste al RFC 1867, “Form-based File Upload in HTML” [21]. Es decir, si se envía una petición HTTP usando el método POST, y con un contenido de tipo “multipart/form-data”, entonces los métodos de *FileUpload* pueden procesar la petición y dejar disponible los resultados de forma que sean fáciles de utilizar por la aplicación que llamó a esos métodos.

La versión utilizada en el proyecto es la 1.1.1, empaquetada en el archivo `commons-fileupload-1.1.1.jar`, ya incluido en el directorio de bibliotecas externas de la Aplicación Web, `WEB-INF/lib`. A su vez, este paquete necesita para su correcto funcionamiento del paquete Commons IO, también de Jakarta, que también se encuentra empaquetado en el archivo `commons-io-1.2.jar`, incluido en el mismo directorio de bibliotecas externas de la Aplicación Web.

11.1 Cómo funciona

Una petición de subida de archivo contiene una lista ordenada de objetos que se codifican de acuerdo a la RFC 1867, "Form-based File Upload in HTML". *FileUpload* procesa la petición y proporciona a la aplicación una lista de objetos subidos individuales. Cada uno de estos objetos implementa la interfaz `FileItem`, a pesar de su implementación subyacente.

Cada objeto de archivo tiene un número de propiedades (como el nombre o el tipo de contenido) que pueden ser utilizadas en la aplicación. Así mismo, la interfaz `FileItem` proporciona métodos para diferenciar simples campos del formulario de un archivo subido y acceder a los datos de la manera más apropiada.

Los objetos de archivo se crean usando un `FileItemFactory`. Esta fábrica tiene todo el control sobre cómo se crean los objetos. Así, se puede configurar que `FileUpload` almacene los datos del objeto en memoria o en disco, dependiendo del tamaño del objeto.

En primer lugar, se necesita procesar la propia petición, comprobando que sea de subida de archivo, mediante un método estático que lo comprueba:

```
boolean isMultipart = ServletFileUpload.isMultipartContent(request);
```

Se puede configurar el comportamiento del manejador de subidas o de la fábrica de objetos archivo o de ambos. A continuación se muestra el código necesario para realizarlo de forma básica:

```
// Crea una fábrica para guardar objetos archivo en disco
DiskFileItemFactory factory = new DiskFileItemFactory();

// Establece las constantes de la fábrica
factory.setSizeThreshold(máximoTamañoEnMemoriaPersonalizado);
factory.setRepository(directorioTemporalPersonalizado);

// Crea un Nuevo manejador de subida de archivo
ServletFileUpload upload = new ServletFileUpload(factory);

// Establece el tamaño máximo de la petición
upload.setSizeMax(tamañoMáximoDeLaPeticionPersonalizado);

// Procesa la petición
List /* FileItem */ items = upload.parseRequest(request);
```

Siempre se puede pasar sin configurar alguno de esos aspectos, que se quedarán a sus valores por defecto:

- Límite de tamaño en memoria: 10KB.
- El directorio temporal es el del sistema por defecto, tal y como devuelve la llamada `System.getProperty("java.io.tmpdir")`.
- El tamaño máximo de la petición se establece en -1, sin límite.

Por razones de rendimiento puede ser necesario establecer algún límite en el tamaño máximo de los archivos a subir.

Una vez que ha concluido el proceso de lectura de la petición, se tendrá un objeto `List` de objetos archivo para procesar, normalmente de forma diferente que los campos normales de formulario:

```
// Procesa los objetos subidos
Iterator iter = items.iterator();
while (iter.hasNext()) {
    FileItem item = (FileItem) iter.next();

    if (item.isFormField()) {
    // Si es un campo de formulario
        String name = item.getFieldName();
        String value = item.getString();
        ...
    } else {
    // Si es un archivo subido
        String fieldName = item.getFieldName();
        String fileName = item.getName();
        String contentType = item.getContentType();
        boolean isInMemory = item.isInMemory();
        long sizeInBytes = item.getSize();
        ...
        if (writeToFile) {
            File uploadedFile = new File(...);
            item.write(uploadedFile);
        } else {
            InputStream uploadedStream = item.getInputStream();
            ...
            uploadedStream.close();
        }
    }
}
```

11.2 Paquete Commons IO

El paquete `org.apache.commons.io` [40] proporciona una biblioteca de utilidades para asistir en el desarrollo de funcionalidades de Entrada/Salida. Es necesaria esta biblioteca para que funcione correctamente la biblioteca “Commons FileUpload”. La versión utilizada en el proyecto es la 1.2, empaquetada en el archivo `commons-io-1.2.jar`, ya incluido el directorio de bibliotecas externas de la Aplicación Web, `WEB-INF/lib`. Se han utilizado directamente sólo algunas de sus funcionalidades en el proyecto, aprovechando que estaban disponibles.

Las utilidades del paquete incluyen tres áreas:

- Clases de utilidad: con métodos estáticos para realizar tareas comunes
- Filtros: varias implementaciones de filtros de archivos
- Flujos: flujos de utilidad, implementaciones de “reader” y “writer”.

11.3 Conclusiones sobre el paquete Commons FileUpload

Ya que los servidores no implementan ningún método para manejar subidas de archivos, dejando todo el proceso a la aplicación del usuario, el paquete para manejo de subida de archivos de Jakarta, Commons FileUpload, proporciona una serie de clases que logran que sea muy fácil tratar con los archivos subidos al servidor. Además, se pueden configurar muchos aspectos para logar mejorar el rendimiento limitando el tamaño de los archivos a subir, por ejemplo.

En definitiva, gracias al paquete Commons FileUpload se puede tener acceso a los archivos subidos al servidor en una petición con un bajo impacto en el programa y de forma sencilla y de alto rendimiento, con varios aspectos configurables en caso de necesidad.

Además, el paquete Commons IO incluye algunos métodos y clases con utilidades que han ayudado a realizar algunas tareas relacionadas con los nombres de los archivos, y con listar determinados archivos de un directorio.

12. Conclusiones

La tecnología Servlet de Java permite añadir contenido dinámico a un servidor usando la plataforma Java. Generalmente, el contenido generado es HTML [49], pero puede ser otro tipo de datos como XML. Los Servlets son los homólogos Java de las tecnologías de contenido Web dinámico no Java, como CGI y ASP .NET.

La utilización de los Servlets permite gozar de las ventajas de utilizar tecnología Java, como son la portabilidad de la Aplicación Web generada, gozar de un recolector de basura, seguridad…

También, las JSPs permiten escribir Servlets productores de texto, sobre todo productores de páginas Web HTML, de forma fácil y sencilla, igual que se haría con una página HTML normal, pero incluyendo cualquier código Java que se quiera en ella.

A parte de los Servlets propiamente dichos y las JSPs, la tecnología incluye una serie de características que expanden su funcionalidad. Por ejemplo, mediante los filtros se puede realizar modificaciones sobre las peticiones, independientemente del Servlet destino, modificando el comportamiento de la petición o la respuesta aplicándoles envoltorios; también se definen escuchadores de eventos, que permiten crear clases que respondan a determinados “eventos” ante los que se quiera reaccionar.

También incluye el control de excepciones de Java, añadiendo además la capacidad de capturar las excepciones lanzadas a nivel global o a nivel particular de cada JSP o Servlet, mostrando páginas de error personalizadas también para los códigos de error HTTP [52].

La tecnología Servlet incluye seguimiento de la sesión de usuario en las conexiones HTTP de forma transparente al programador, permitiendo almacenar todo tipo de información de usuario en un ámbito de sesión, además de los demás ámbitos que proporciona, como son en ámbito de petición, de toda la Aplicación Web, o de página, para una JSP.

Las JSPs con código Java directamente en ellas se pueden volver difíciles de mantener y depurar, pero, gracias a la utilización de la biblioteca estándar de etiquetas, las etiquetas personalizadas y el lenguaje de expresión de JSP, se puede conseguir que una JSP que se genere dinámicamente tenga la apariencia de casi una página HTML normal, o, al menos, un documento XML. Así, se tendrá una página mucho más fácil de mantener, que incluso podrá ser creada por un desarrollador de páginas Web estáticas con muy poco entrenamiento, dejando la creación de la lógica y las etiquetas a desarrolladores Java, o JSP. De esta manera, se puede desarrollar una Aplicación Web que siga el patrón de diseño "Modelo Vista Controlador", permitiendo una separación entre la lógica de negocio (implementada mediante Servlets, etiquetas personalizadas, o JavaBeans [13], por ejemplo) y la presentación (implementada mediante JSPs).