Unión Europea

Fondo Social Europeo

El FSE invierte en tu futuro

Fecha	Versión	Descripción
10/09/2022	1.0.0	Versión inicial.
20/09/2022	1.0.1	Se añaden nuevos ejemplos con JavaBeans y se corrigen errores. Se añaden los videos. Fichero web.xml y se matizan más ejemplos.
01/09/2024	2.0.0	Actualización de los materiales.
13/09/2024	2.0.1	Correción de errores en los objetivos.

Unidad 2 - JSP y Servlets

Objetivos

- Estudiar los componentes Web JSP de la arquitectura JEE
- Comprender la relación estrecha entre JSP y Servlets
- Conocer las posibilidades que nos ofrecen las páginas JSP para la generación de contenido dinámico
- Estudiar los elementos JSP: Directivas, Scripts, Acciones y Variables predefinidas
- Aplicar los nuevos conocimientos adquiridos para completar los formularios HTML vistos en otras asignaturas y darles más sentido.
- Comprender las ventajas de la separación de generación de contenidos y su presentación.
- Formar al alumn@ en el uso de NetBeans para la creación de proyectos Web, formularios, JavaBeans y páginas JSP.
- Conocer las posibilidades que nos ofrecen los Servlet y como se complementan con las páginas JSP.
- Estudiar la estructura de un Servlet.
- Capturar y procesar los datos de usuario.
- Estudiar y aplicar los ámbitos de los objetos, sabiendo cuando se aplica cada uno de ellos.
- Estudiar el ciclo de vida de un Servlet comprendiendo como y cuando se ejecuta cada método.
- Aplicar los nuevos conocimientos adquiridos para completar los conocimientos vistos hasta ahora, comenzando ya a crear pequeñas aplicaciones (sin base de datos, que se verá en otra unidad).

1. Arquitectura JEE: Componente Web JSP.

En este punto nos centraremos en uno de los dos componentes Web de la arquitectura JEE: los componentes JSP, los cuales nos permitirán crear páginas Web dinámicas.

Jakarta Server Pages TM (JSP) permiten la generación dinámica de páginas Web combinando código Java (scriptlets) con un lenguaje de marcas como HTML, para generar el contenido de la página.

Como parte de la familia de la tecnología Java, con JSP podemos desarrollar aplicaciones Web independientes de la plataforma. Una característica importante es que su uso conjunto con Servlets permite separar la interfaz del usuario de la generación del contenido, dando lugar a procesos de desarrollo más rápidos y eficientes.

En este tema introduciremos el uso de JavaBeans, o lo que es lo mismo POJO's con una determinada estructura.

Las JavaBeans son clases Java que encapsulan una determinada funcionalidad, ofreciendo un interfaz de acceso para consultar o modificar determinadas variables. Un JavaBean debe cumplir las siguientes reglas:

- Debe tener un constructor sin argumentos.
- Debe tener un conjunto de variables internas, privadas, ocultas al exterior (por ejemplo, supongamos que una de ellas es apellidos de tipo String).
- Debe ofrecer un interfaz con métodos para consultar (métodos get) y modificar (métodos set) cada una de las anteriores propiedades. Por ejemplo, para la variable msg, debe ofrecer los métodos getApellidos() y setApellidos(String apellidos).

En resumen, las tecnologías JSP y Servlets son de gran importancia para la programación de Web de contenido dinámico y nos permiten:

- Independencia de la plataforma
- Rendimiento optimo
- Separación de la lógica de la aplicación de la presentación de los datos
- Uso de componentes (JavaBeans)

2. Ciclo de vida de una página JSP.

Generaremos archivos con extensión .jsp que incluyen, dentro de la estructura de etiquetas HTML, las sentencias Java a ejecutar en el servidor

La tecnología JSP es una extensión de la tecnología Servlets (la estudiaremos con detenimiento más adelante), los cuales son aplicaciones 100% Java que corren en el servidor: Un Servlet es creado e inicializado, se procesan las peticiones recibidas y por último se destruye. El Servlet es cargado una sola vez y esta residente en memoria mientras se procesan las peticiones recibidas y se generan las respuestas a los usuarios.

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.

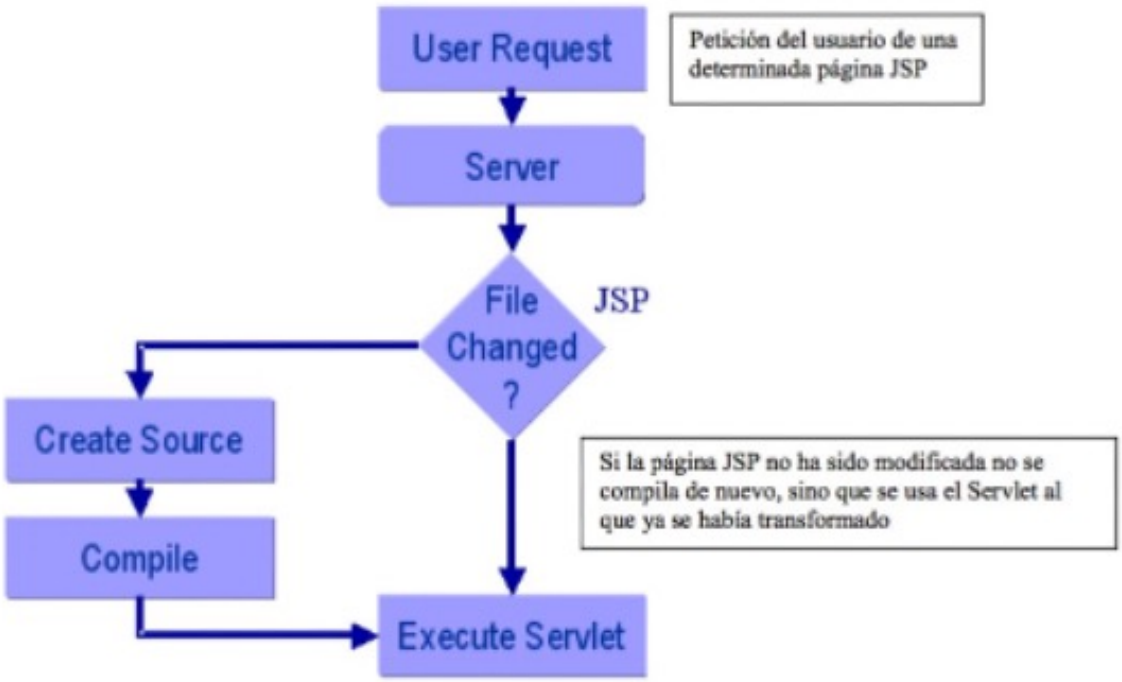
Cada vez que un cliente solicita al servidor Web una página JSP, este pasa la petición al motor de JSP el cual verifica si la página no se ha ejecutado antes ó fue modificada después de la última compilación, tras lo cual la compila, convirtiéndola en Servlet, la ejecuta y devuelve los resultados al cliente en formato HTML.

Por ejemplo, la siguiente página web generaría en el Servlet el método `_jspService` similar al siguiente:

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

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

En la imagen siguiente se puede ver un esquema del proceso de conversión de una página JSP en un Servlet.



3. Contenido dinámico: JSP.

Como se ha indicado anteriormente las páginas JSP generan páginas Web dinámicas combinando una parte estática (HTML, javascript, etc.) con una parte dinámica. Esta parte dinámica se puede generar mediante:

1. Usar código Java directamente a través de implementar los métodos que necesite
2. Usar código Java indirectamente a través de invocar métodos de clases
3. Hacer uso de JavaBeans
4. Desarrollar y usar “customtags“
5. Usar custom tags de terceros o JSTL (JSP Standard Tag Library)
6. Diseñar un framework MVC (Modelo, Vista, Controlador) propio
7. Usar frameworks MVC de terceros (JSF, ZK, PrimeFaces, Thymeleaf, etc)

En esta unidad veremos los apartados 1, 2 y 3. Los apartados 6 y 7 los abordaremos más adelante, cuando tengamos conocimientos de las tecnologías involucradas en el patrón MVC. Los apartados 4 y 5 no los veremos por razones temporales. Respecto al apartado 7 cabe indicar que usaremos Thymeleaf, que nos proporciona una solución adecuada al diseño visual de nuestras aplicaciones, por ello trabajaremos con este framework cuando comencemos a trabajar con Spring Boot.

El código fuente de una página JSP incluye:

- **Directivas JSP**: Dan información global de la página, por ejemplo, importación de elementos, página que maneja los errores, cuando la página forma parte de una sesión, etc.
- **Scripts de JSP**: Es código Java embebido en la página.
- **Variables predefinidas**: Son objetos definidos de forma implícita en toda página JSP

- **Acciones de JSP**: Permite controlar el comportamiento del motor de Servlets Veamos cada una de estas partes y relacionémoslo con el uso de NetBeans

4. Directivas JSP

Una directiva JSP afecta a la estructura general de la clase Servlet que se genera automáticamente a partir de la JSP. Normalmente tienen la siguiente forma:

```
<%@ directive attribute="value" %>
```

Sin embargo, también podemos combinar múltiples selecciones de atributos para una sola directiva, de esta forma:

```
<%@ directive attribute1="value1"
              attribute2="value2"
              ...
              attributeN="valueN" %>
```

Hay dos tipos principales de directivas:

- `page`, que nos permite hacer cosas como importar clases, personalizar la clase de la que extiende el Servlet generado, etc
- `include`, que nos permite insertar un fichero dentro de la clase *Servlet* en el momento que el fichero JSP es traducido a un Servlet

Podéis encontrar un video realizando un ejemplo que trabaja con las directivas en el portal de la asignatura (denominado *Unidad 2 - Video 1 - Ejemplo Directivas JSP*).

4.1. La directiva page

La directiva *page* nos permite definir uno o más de los siguientes atributos (veremos únicamente aquellos que nos interesan):

- `import="package.class"`: Esto nos permite especificar los paquetes que deberían ser importados. Por ejemplo:

```
<%@ page import="java.util.*" %>
```

El atributo `import` es el único que puede aparecer múltiples veces.

- `session="true|false"`: Un valor de `false` indica que no se usarán sesiones, y los intentos de acceder a la variable `session` resultarán en errores en el momento en que la página JSP sea traducida a un Servlet.
- `extends="package.class"`: Esto indica la clase de la que extenderá el Servlet que se va a generar. Debemos usarla con extrema precaución, ya que el servidor Java EE podría no aceptarlo correctamente.
- `info="message"`: Define una descripción que puede usarse para ser recuperado mediante el método `getServletInfo`.
- `errorPage="url"`: Especifica una página JSP que se debería procesar si se lanzará cualquier excepción que no fuera capturada en la página actual.
- `isErrorPage="true|false"`: Indica si la página actual actúa o no como página de error de otra página JSP. El valor por defecto es `false`.

5. Scripts JSP

Los elementos de Script JSP nos permiten insertar código Java dentro del Servlet que se generará desde la página JSP actual. Hay tres formas:

1. Expresiones de la forma `<%= expresión %>` que son evaluadas e insertadas en la salida.
2. Scriptlets de la forma `<% código %>` que se insertan dentro del método `service` del Servlet, que es el método que se ejecuta una vez traducida la JSP a Servlet.
3. Declaraciones de la forma `<%! código %>` que se insertan en el cuerpo de la clase del Servlet, fuera de cualquier método existente.

Cabe indicar que los comentarios en JSP se introducen a través de `<%-- comentario --%>`

5.1. Expresiones JSP

Una expresión JSP se usa para insertar valores Java directamente en la salida, es decir mostrarlos directamente al usuario. Tiene la siguiente forma:

```
<%= expresión Java %>
```

Esta expresión Java es evaluada, convertida a un `String`, e insertada en la página. Esta evaluación se realiza durante la ejecución (cuando se solicita la página) y así tiene total acceso a la información sobre la solicitud. Por ejemplo, esto muestra la fecha y hora en que se solicitó la página:

```
    <body>
        Fecha actual: <%= new Date() %>
    </body>
```

5.2. Scriptlets JSP

Los scriptlets JSP nos permiten insertar código arbitrario dentro del método `service` del Servlet que será construido al generar la página. Tienen la siguiente forma:

```
<% Código Java %>
```

Hay que resaltar que el código dentro de un scriptlet se insertará exactamente como está escrito dentro del método `service` del Servlet, y cualquier HTML estático anterior o posterior al scriptlet se convierte en sentencias `println`. Esto significa que los scriptlets no necesitan completar las sentencias Java, y los bloques abiertos pueden afectar al HTML estático fuera de los scriplets. Por ejemplo, el siguiente fragmento JSP, contiene una mezcla de HTML y scritplets:

```
<% if (Math.random() < 0.5) { %>
Ten un
<B>bonito</B> día!
<% } else { %>
Ten un
<B>buen</B> día!
<% } %>
```

Se convertirá en algo como esto dentro del método `service`:

```
if (Math.random() < 0.5) {
  out.println("Ten un <B>bonito</B> día!");
} else {
  out.println("Ten un <B>buen</B> día!");
}
```

Un ejemplo de scriptlet que usaremos en el video será el siguiente:

```
<% Calendar cal = Calendar.getInstance();
   out.println("La fecha actual es: "
            + cal.get(Calendar.DATE) + " /"
            + (cal.get(Calendar.MONTH) + 1) + " / " + cal.get(Calendar.YEAR));
%>
```

5.3. Declaraciones JSP

Una declaración JSP nos permite definir métodos o campos que serán insertados dentro del cuerpo principal de la clase `Servlet` (fuera del método `service` que procesa la petición). Tienen la siguiente forma:

```
<%! Código java%>
```

Como las declaraciones no generan ninguna salida, normalmente se usan en conjunción con lo anterior. Por ejemplo, aquí tenemos un fragmento de JSP que imprime el nuúmero de veces que se ha solicitado la página actual desde que el servidor se arrancó (o la clase del Servlet se modificó o se recargó):

```
<%!private int accessCount = 0;%><!--define un campo -->
Accesos a la página:
<%=++accessCount%>
```

Como se ve en el ejemplo esa variable pasa a pertenecer al Servlet que es generado. Dado que en el resto de llamadas se llama a esa instancia creada en memoria no se perderá el valor de la variable entre llamada y llamada.

Podéis encontrar un video realizando un ejemplo que trabaja con los scripts en el portal de la asignatura (denominado *Unidad 2 - Video 2 - Ejemplo Scripts JSP*).

6. Variables predefinidas JSP

Para simplificar el código en expresiones y scriplets JSP, tenemos ocho variables definidas automáticamente, también llamadas objetos implícitos. Simplifican el código dado que también se pueden acceder a estos objetos haciendo una serie de casting, llamando a métodos concretos, etc. Las variables disponibles son:

- `request` : Este es el `HttpServletRequest` asociado con la petición, y fundamentalmente nos permite mirar los parámetros de la petición (mediante `getParameter`), además de otra serie de métodos (`HttpServletRequest` 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. Con `getParameterNames()` se obtiene una lista con los nombres de los parámetros enviados por el cliente).
- `response` : Este es el `HttpServletResponse` asociado con la respuesta al cliente (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).
- `out` : Este es el `PrintWriter` usado para enviar la salida al cliente. Observemos que `out` se usa casi exclusivamente en scriptlets ya que las expresiones JSP insertan valores directamente en la salida.

- `session` : Este es el objeto `HttpSession` asociado con la petición, permitiendo almacenar y acceder a la información de la sesión que controla el servlet. Los ámbitos de los objetos los estudiaremos más adelante. Si usáramos el atributo `session` de la directiva `page` para desactivar las sesiones, los intentos de referenciar la variable `session` causarán un error en el momento de traducir la página JSP a un Servlet.
- `application` : Este es el `ServletContext` obtenido mediante `getServletConfig().getContext()`.Este objeto es común para toda la aplicación Web y, entre otras cosas, nos permite almacenar información que será accesible desde todas las páginas de la aplicación Web, independientemente de `session`, de forma que permite compartir información entre los distintos componentes web como Servlets y JSP. Su funcionalidad esta orientada a almacenar claves/valores que sean comunes para toda la aplicación.
- `config` : Este es el objeto `ServletConfig` asociado al Servlet al que se traduce la página. Permite acceder a parámetros de inicialización del Servlet y a su contexto. `ServletConfig` es un objeto que contiene algunos parámetros iniciales o información de configuración creada por `ServletContainer` y pasada al servlet durante la inicialización. `ServletConfig` es para un servlet en particular, eso significa que uno debe almacenar información específica del servlet en `web.xml` y recuperarla usando este objeto (muy importante el fichero `web.xml` que ya veremos más adelante).
- `pageContext` : Es un objeto de la clase `PageContext`. Entre otras cosas, nos permite almacenar información en diferentes ámbitos (`Page`, `Request`, `Session`, `Application`) como veremos más adelante.
- `page` : Esto es sólo un sinónimo de `this`.

Podéis encontrar un video realizando un ejemplo que trabaja con variables predefinidas en el portal de la asignatura (denominado *Unidad 2 - Video 3 - Ejemplo Variables Predefinidas JSP*).

7. Acciones JSP

Las acciones JSP son acciones que tienen la forma `<jsp:accion [parámetros]/>` y permiten controlar el comportamiento del motor de Servlets. Permiten insertar un fichero dinámicamente, reutilizar componentes JavaBeans, reenviar al usuario a otra página, o generar HTML para el plug-in Java.

7.1. Acción para insertar ficheros

La acción `jsp:include` nos permite insertar ficheros en una página que está siendo generada. La sintaxis se parece a esto:

```
<jsp:include page="url_relativa" />
```

Al contrario que la directiva `include`, que inserta el fichero en el momento de la conversión de la página JSP a un Servlet, esta acción inserta el fichero en el momento en que la página es solicitada. Esto se paga en eficiencia, pero proporciona una significativa flexibilidad, dado que modificar las páginas incluidas no afectará al rendimiento que siempre será constante.

7.2. Acción para reenviar peticiones

Esta acción nos permite reenviar la petición a otra página. Tiene un sólo atributo, `page`, que debería consistir en una URL relativa. Este podría ser un valor estático, o podría ser calculado en el momento de la petición, como en estos dos ejemplos:

```
<jsp:forward page="/utils/pagina2.jsp" />
<jsp:forward page="<%= someJavaExpression %>" />
```

Podéis encontrar un video realizando un ejemplo que trabaja con acciones JSP de este tipo en el portal de la asignatura (denominado *Unidad 2 - Video 4 - Ejemplo Forward JSP*).

7.3. Acción para insertar plugin

Esta acción, denominada `jsp:plugin`, sirve para incluir, de manera portable e independiente del navegador, *applets* que utilicen alguna librería de Java 2 (Swing, colecciones, Java 2D, ...), ya que las máquinas virtuales Java distribuidas con algunos navegadores relativamente antiguos (Explorer 5.x, Netscape 4.x,...) son de una versión anterior a Java 2. Esta acción no está en desuso y no es recomendable usarla.

7.4. Acción para cargar y usar JavaBean (POJO) en JSP

Esta acción, denominada `jsp:useBean`, nos permite cargar y utilizar un JavaBean en la página JSP. La sintaxis más simple para especificar que se deberá usar un JavaBean es:

```
<jsp:useBean id="name" class="package.class" scope=="page|request|session|application"
type="..." />
```

Esto significa "instancia un objeto de la clase especificada por `class` si no existe uno con el mismo nombre y ámbito, y únelo a una variable con el nombre especificado por `id`". Los atributos `id` y `class` son obligatorios. Los atributos `scope` y `type` son opcionales.

Mediante la especificación del atributo `scope` se puede hacer que ese JavaBean se asocie con más de una sola página. Esto lo veremos más adelante.

Mediante la especificación del atributo `type` se especifica el tipo de la variable a la que se referirá el objeto. Este debe corresponder con el nombre de la clase o una superclase o un interface que implemente la clase, es decir permite realizar un casting del objeto.

Ahora, una vez que tenemos un JavaBean, podemos:

- modificar sus propiedades mediante `jsp:setProperty`, o usando un scriptlet y llamando a un método explícitamente sobre el objeto con el nombre de la variable especificada anteriormente mediante el atributo `id`.
- Del mismo modo para leer sus propiedades usaremos la acción `jsp:getProperty`, o usaremos un scriptlet que llame al método `getXXX` correspondiente.

7.5. Acción para establecer valores a objetos JavaBeans

Usamos `jsp:setProperty` para establecer valores de propiedades de los beans que se han referenciado anteriormente. Lo usaremos de esta forma:

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName" property="someProperty" value="someValue" />
```

Esto significa "usa la instancia referenciada como *myName* y establece su propiedad llamada `someProperty` con el valor de *someValue*". Un caso especial es la orden:

```
<jsp:setProperty name="myName" property="*"/>
```

la cual significa que todos los parámetros del formulario cuyos nombres correspondan con nombres de propiedades del bean serán almacenados.

7.6. Acción para recuperar valores de objetos JavaBeans

Usaremos `jsp:getProperty` recupera el valor de una propiedad del bean, lo convierte a un `String`, e inserta el valor en la salida de la JSP. Se usa del siguiente modo:

```
<jsp:useBean id="myName" ... />
...
<jsp:getProperty name=" myName " property="someProperty" />
```

Esto significa "usa la instancia referenciada como *myName*, recupera su propiedad llamada `someProperty` y muestrala".

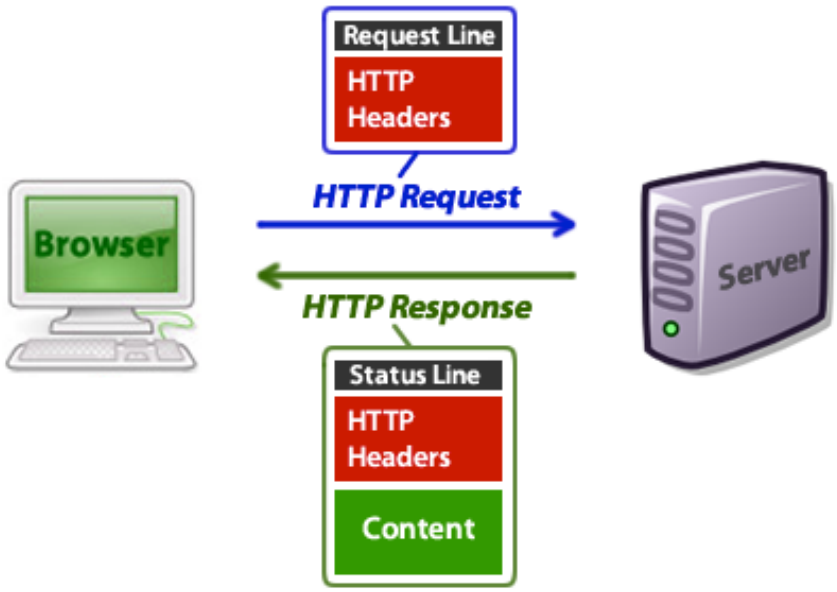
8. Formularios JSP y JavaBeans

8.1. Formularios con JSP

Usando JSP, los datos de un formulario (la información que el usuario introduce en él) se almacenan en un objeto `request` que es enviado desde el navegador Web del cliente hasta el contenedor Web.

La petición es procesada y el resultado se envía a través de un objeto `response` de vuelta al navegador.

En la siguiente imagen podemos ver como funciona el proceso:



Tanto `request` como `response` están disponibles implícitamente como variables predefinidas.

Mediante el método `request.getParameter` se recuperan los datos desde el formulario en variables creadas usando etiquetas de introducción de datos (objetos de tipo `input`, `textarea`, `select`, `radiobutton`). En el caso de que se recuperen varios valores (`checkbox`) se usará el método `request.getParameterValues`.

Podéis encontrar un video realizando un ejemplo que trabaja con formularios con JSP en el portal de la asignatura (denominado *Unidad 2 - Video 5 - Ejemplo Formularios JSP*).

8.2. Formularios con JSP y JavaBeans

Cuanto más código esté implicado en un formulario, más importante es no mezclar la lógica de negocio con la presentación final. De este modo:

- el código de la página JSP se debe limitar a la presentación final: formatea los datos de los JavaBeans para su visualización en el navegador.

- Los JavaBeans contienen los datos que debe mostrar la página. Estos JavaBeans se han completado en la capa de lógica de negocio y llegan hasta la capa de vista.

Aunque lo hemos explicado en el punto 7 volvemos a hacer un pequeño recordatorio. Para hacer accesible un JavaBean a una página JSP se emplea la etiqueta `useBean`. En su forma más simple la sintaxis es:

```
<jsp:useBean id="nombreBean" class="paquete.Clase"/>
```

En caso de que el JavaBean referenciado no existiera previamente, esta etiqueta se puede ver como la creación de una variable en Java de nombre `nombreBean` y de tipo `paquete.Clase`. Así, para crear un *bean* de tipo `usuarioBean` sería equivalente utilizar las siguientes expresiones:

```
<jsp:useBean id="usuario" class="beans.UsuarioBean" />
```

```
<% beans.UsuarioBean usuario = new beans.UsuarioBean() %>
```

El acceso a una propiedad se realiza mediante la etiqueta `jsp:getProperty`. Su sintaxis es:

```
<jsp:getProperty name="nombreBean" property="nombrePropiedad"/>
```

donde `nombreBean` debe corresponderse con el atributo `id` definido mediante alguna etiqueta anterior `jsp:useBean`. El acceso a la propiedad también se podría hacer llamando al método Java correspondiente. De este modo, el acceso a la propiedad `visitas` del *bean* `usuario` se puede hacer mediante las dos formas alternativas:

```
<jsp:getProperty name="usuario" property="visitas"/>
```

```
<%= usuario.getVisitas() %>
```

La asignación de valores se realiza mediante la etiqueta `jsp:setProperty`. Esta etiqueta requiere tres parámetros: `name` (el `id` del JavaBean, definido anteriormente mediante alguna etiqueta `jsp:useBean`), `property` (el nombre de la propiedad) y `value` (el valor que se desea dar a la propiedad). Por ejemplo, para darle a la propiedad `visitas` del JavaBean `usuario` el valor 1 se haría:

```
<jsp:setProperty name="usuario" property="visitas" value="1"/>
```

Una forma alternativa en código Java sería llamar directamente al método, aunque normalmente es preferible el uso de la sintaxis anterior:

```
<% usuario.setVisitas(1) %>
```

Entre los beneficios en la utilización de JavaBeans para mejorar las páginas JSP destacaremos:

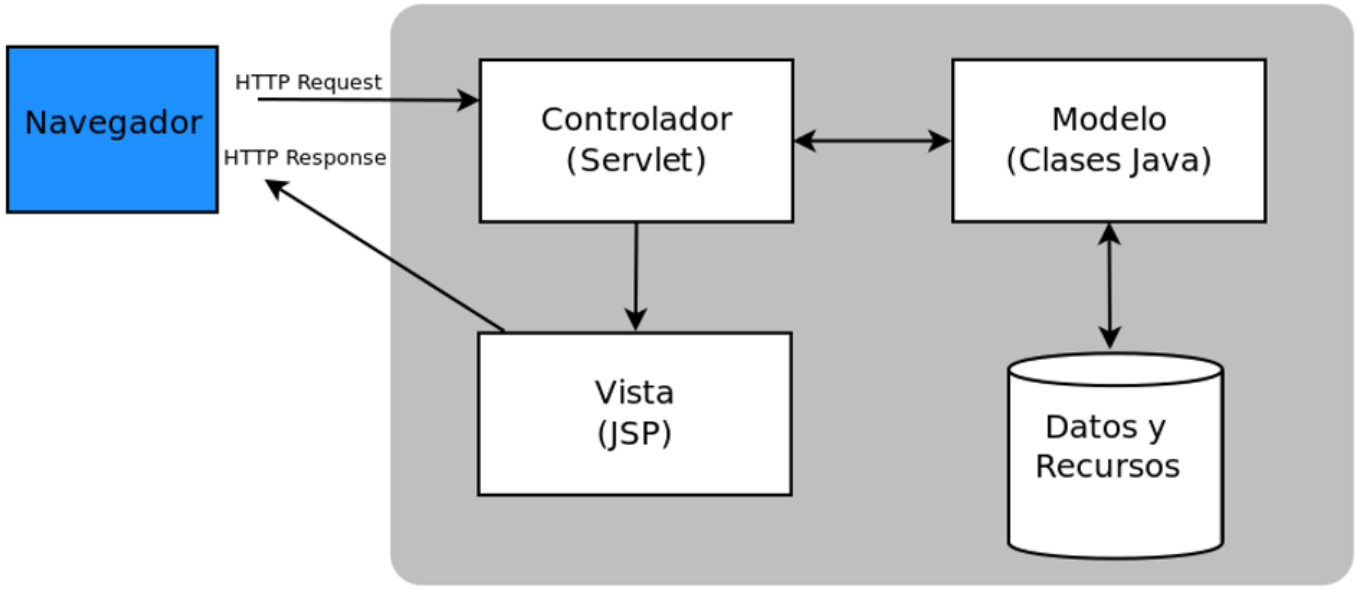
- Componentes Reutilizables: diferentes páginas JSP pueden reutilizar los mismos JavaBeans.
- Separación de la lógica de negocio de la lógica de presentación: podemos modificar la forma de mostrar los datos sin que afecte a la lógica del negocio, y viceversa.

Podéis encontrar un video realizando un ejemplo que trabaja con formularios con JSP y JavaBeans en el portal de la asignatura (denominado *Unidad 2 - Video 6 - Ejemplo Formularios JSP con JavaBeans*).

9. Arquitectura JEE: Componente Web Servlet.

Ahora nos centraremos en uno de los dos componentes Web de la arquitectura JEE: Servlets, los cuales nos permitirán generar contenido dinámico Web, además de independizar totalmente la presentación de los datos de su generación. Estos componentes se ejecutarán en el contenedor Web de un servidor JEE, como ya explicamos en la unidad 1.

Mediante el uso de Servlets las páginas JSP se encargarán únicamente de la presentación de datos, mientras que los Servlets recogen los datos. Llamarán a la lógica de negocio necesaria, encapsularan los datos en un JavaBean y mandaran la respuesta hacia una página JSP para que presente la salida.



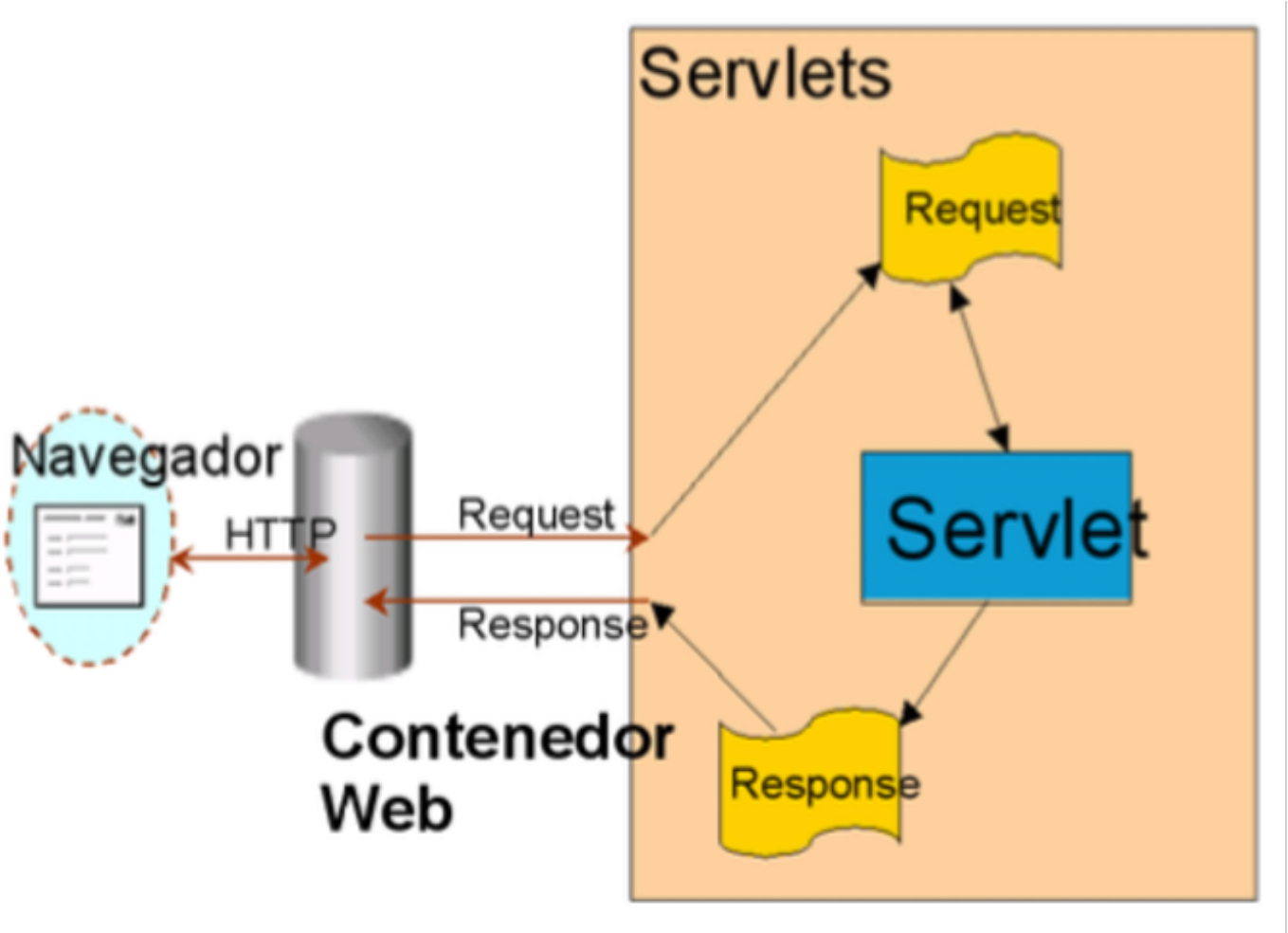
El cliente del Servlet será un navegador Web que realiza una petición a través de un formulario Web. El Servlet procesará la petición y realizará las operaciones pertinentes (lógica de negocio). Normalmente el Servlet devolverá una respuesta que se generará de forma dinámica en formato HTML.

En los Servlets no existe una clara distinción entre el código HTML (contenido estático) y el código Java (lógica del programa y/o negocio). El contenido HTML que se envía al cliente se suele encontrar como cadenas de texto que se incluyen en sentencias del tipo:

```
out.println("<h1>Código HTML</h1>")
```

Esto sería un problema dado que mezclaría lógica de negocio con la presentación. Veremos cómo evitar este problema a través de redirigir la salida a páginas JSP utilizando también componentes JavaBeans para comunicar los datos a mostrar.

De lo explicado anteriormente se puede deducir que un Servlet proporciona clases e interfaces que nos permiten interactuar con las peticiones y respuestas del protocolo HTTP. Así por ejemplo el interfaz `HttpServletRequest` representa la petición que se ha realizado a un Servlet y el interfaz llamado `HttpServletResponse` representa la respuesta que se le va enviar al usuario.



10. Estructura de un Servlet.

10.1. Clases componentes de un Servlet

Todo servlet implementa la interface `Servlet` del paquete `javax.servlet`. Fíjaros que deberia haber cambiado a `jakarta.servlet`, pero por concordancia con los materiales que os vais a encontrar en GitHub lo he dejado con esta versión. Esto lo puede lograr de dos formas diferentes:

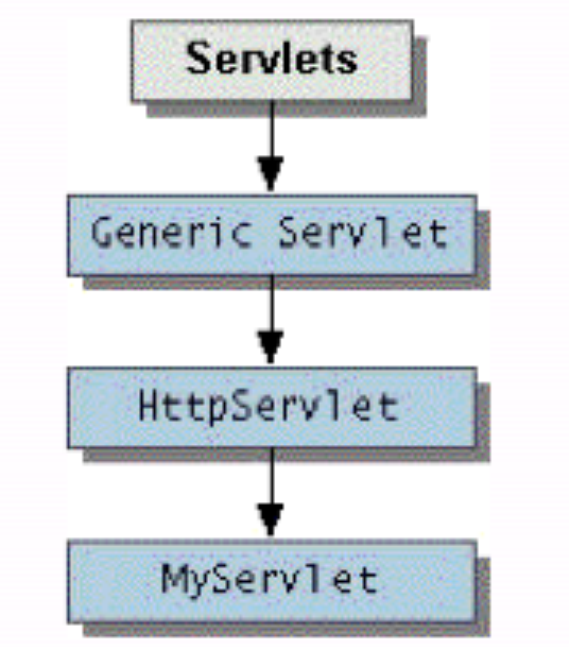
- heredando de `GenericServlet`, que admite peticiones no basadas en ningún protocolo especial.
- o bien, heredando de `HttpServlet` (es subclase de la anterior) que admite peticiones de clientes basadas en el protocolo http. Nosotros usaremos este tipo de Servlets.

La declaración de la clase asociada a un servlet http genérico sería:

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

public class MiServleHttp extends HttpServlet {
...
}
```

En la siguiente figura podemos ver que la interfaz definida en `javax.servlet.Servlet` es implementada por la clase `GenericServlet`, la cual a su vez es extendida por `HttpServlet`. Nuestros Servlets extenderán de `HttpServlet` :



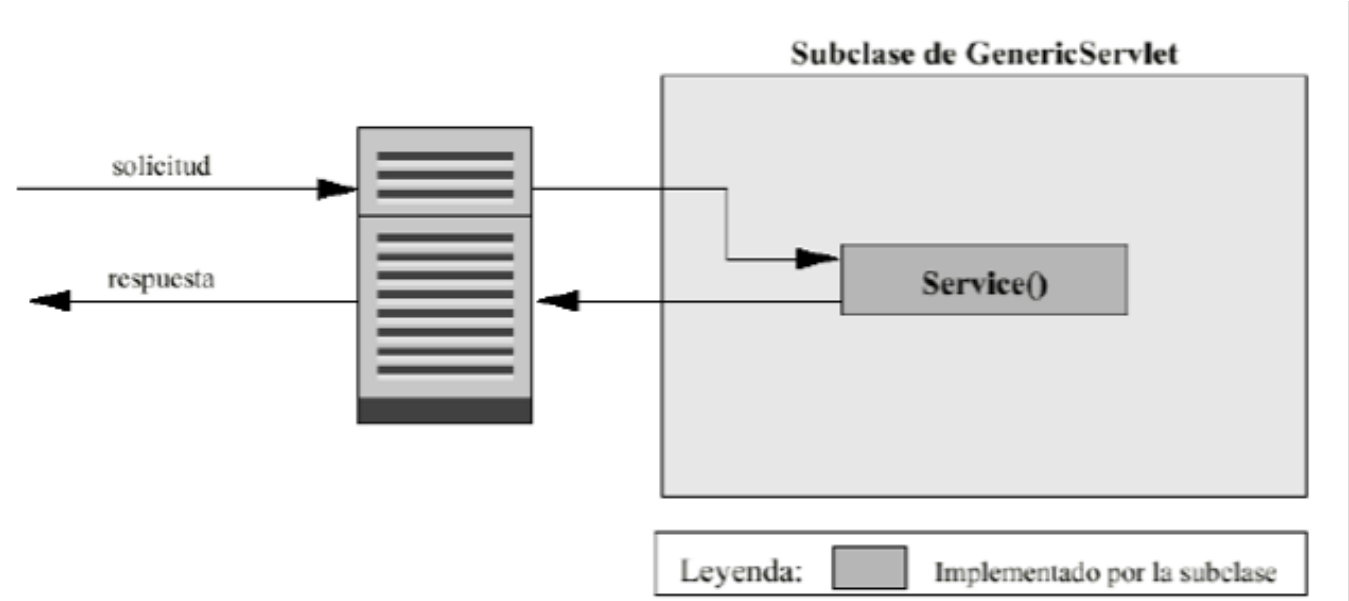
Como nuestros Servlets extienden la clase `HttpServlet` todo servlet va a poder usar directamente los métodos de esta clase, los de su superclase `GenericServlet`, además de los de la propia clase.

La interface `Servlet` declara los métodos del ciclo de vida de un servlet, que son:

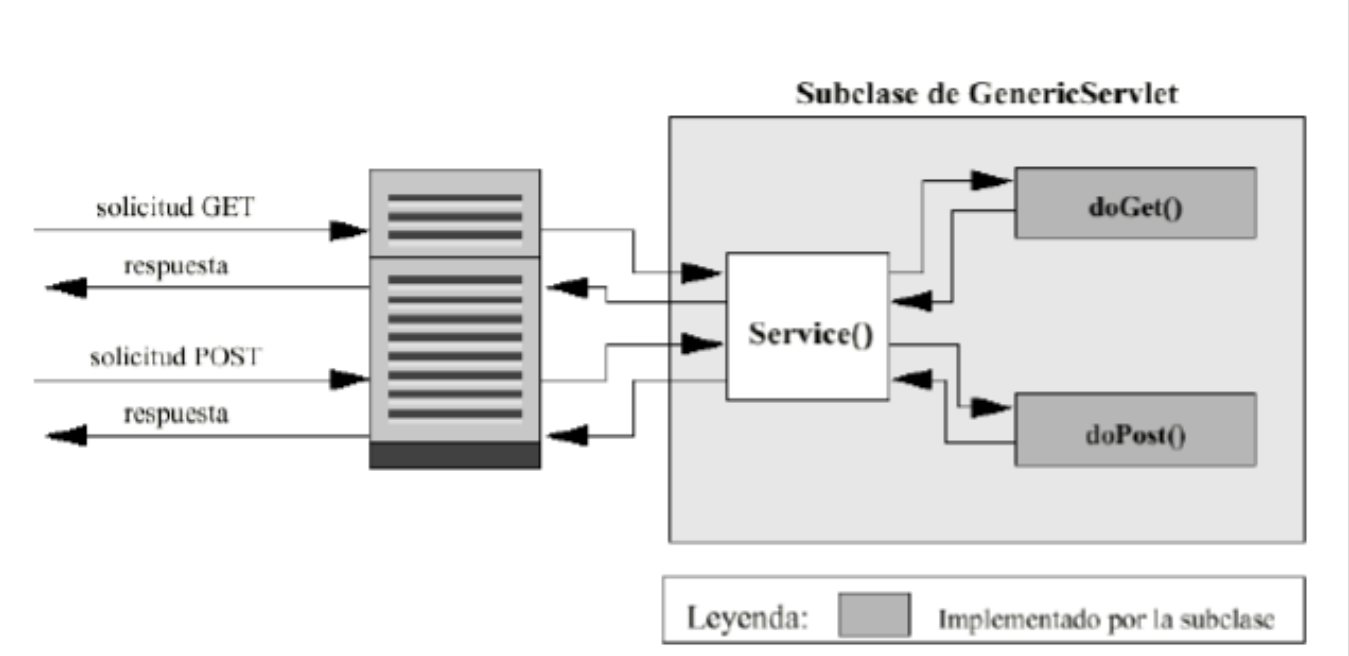
- `void init (ServletConfig config)` : es invocado, una sola vez, por el contenedor del servidor JEE compatible donde se hospeda el servlet y se emplea para inicializarlo. Se ejecuta cuando se realiza la primera petición.
- `void destroy()` : es invocado por el contenedor antes de que el servlet se descargue de memoria y deje de prestar servicio.
- `void service(ServletRequest request, ServletResponse reponse)` : es invocado por el contenedor para procesar las peticiones. Sus argumentos son instancias de las interfaces `javax.servlet.ServletRequest` y `javax.servlet.ServletResponse` que modelan, respectivamente, la petición del cliente y la respuesta del servlet. En un servlet de tipo `HttpServlet`, y este método suele sustituirse por los métodos de `javax.servlet.http.HttpServlet`. 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

- `void doPost(HttpServletRequest request, HttpServletResponse response)` para gestionar peticiones *post*.
- `void doGet(HttpServletRequest request, HttpServletResponse response)` para gestionar peticiones *get*.
- `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()`).

Es decir, en un servlet que extienda de `GenericServlet` ejecutaría el método `service` (deberíamos implementarlo) ante la petición del cliente.



Por otro lado, en los servlets que extiendan de `HttpServlet` tendríamos que implementar el método `doGet` para procesar peticiones GET y `doPost` para procesar peticiones de tipo POST. Recordemos que el atributo `Method` de los formularios permite indicar el tipo de petición (GET o POST).



Podéis encontrar un video realizando un ejemplo donde trabajaremos con nuestro primer Servlet en el portal de la asignatura (denominado *Unidad 2 - Video 7 - Ejemplo Primer Servlet*).

11. Pasos de ejecución de un Servlet

11.1. Ciclo de vida de un Servlet

En este apartado vamos a abordar la forma en la que los servlets son creados y destruidos por el contenedor Web (también llamado en este caso contenedor de servlets). Es el contenedor Web quien controla el ciclo de vida del Servlet.

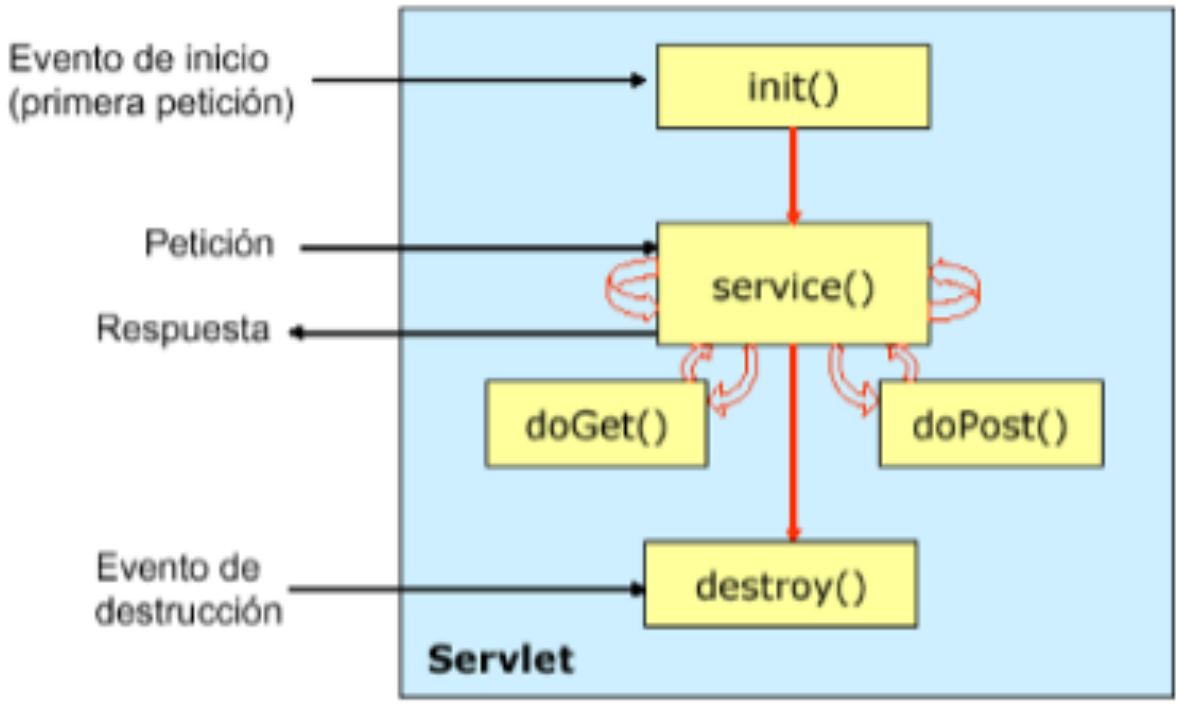
Cuando se crea un servlet, existirá una única instancia de este servlet, que por cada petición que le realicen los usuarios creará un hilo de ejecución (thread) en el que se tratará el método `service` que invocará el método `doGet()` o doPost() según el tipo de petición.

A continuación vamos a tratar todo este proceso en detalle: Cuando se inicia el Servidor Web este lee los descriptores de despliegue de los Servlets que tiene instalados. De este modo el Servidor Web carga la clase asociada al servlet, instanciándola invocando a su constructor sin argumentos por defecto y cargándola en memoria. El objeto `javax.servlet.ServletConfig` es creado por el contenedor durante la carga en memoria y "absorbe" multitud de información sobre el servlet, previa lectura del descriptor de despliegue.

Cuando se realiza la primera petición al servlet este es inicializado. Esta inicialización es única e implica la ejecución, por parte del contenedor, del método `init(ServletConfig config)` del servlet. Por lo tanto este método contendrá el código de inicialización del servlet. Solo es llamado en esta primera petición.

Finalizada la inicialización, el servlet ya está en disposición de procesar las peticiones y generar una respuesta a las mismas. De este modo, cada petición realizada por un usuario sobre el servlet se traduce en un nuevo hilo de ejecución (thread) que realiza una llamada al método `service()`. Múltiples peticiones concurrentes normalmente generan múltiples hilos de ejecución que llaman de forma simultánea al método `service()`.

El método `service()` invocará los método `doGet()` o doPost() dependiendo del tipo de petición HTTP recibida. Los objetos `HttpServletRequest` y `HttpServletResponse` asociados a los argumentos de estos métodos son creados también por el contenedor.



Finalmente, si se tumba el servidor, o se sobrepasa un tiempo límite de inactividad del servlet, o si el servidor está trabajando con insuficiente memoria, el contenedor podría eliminar el objeto servlet de la memoria. Para esto el contenedor ejecutaría el método `destroy` del Servlet. Como se puede ver el ciclo de vida de un servlet sigue el siguiente esquema:

- Ejecución del método `init()` para inicialización del servlet.
- Sucesivas ejecuciones del método `service()` en distintos hilos de ejecución, que resultan en una llamada a un método `doXXX`.
- Ejecución del método `destroy()` para realizar labores de liberación.

Podéis encontrar un video realizando un ejemplo donde trabajaremos el ciclo de vida de un servlet en el portal de la asignatura (denominado *Unidad 2 - Video 8 - Ejemplo Ciclo Vida Servlet*).

11.2. Procesamiento de las 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`. Los métodos de la clase `HttpServlet` que manejan peticiones de cliente (estos métodos son `service`, `doGet`, `doPost`) toman dos argumentos:

- Un objeto `HttpServletRequest`, que encapsula los datos desde el cliente.
- Un objeto `HttpServletResponse`, que encapsula la respuesta hacia el cliente.

11.2.1. HttpServletRequest

Un objeto `HttpServletRequest` proporciona acceso a los datos de cabecera del protocolo HTTP con el que se ha realizado la petición. Además `HttpServletRequest` también permite obtener los argumentos que el cliente envía como parte de la petición (a través de un formulario). Para acceder a los datos del cliente tenemos los siguientes métodos:

- `getParameter(name)` : Devuelve el valor del parámetro con nombre `name`. Si nuestro parámetro pudiera tener más de un valor, deberíamos utilizar `getParameterValues` en su lugar.
- `getParameterValues(name)` : Devuelve un array de valores del parámetro con nombre `name`.
- `getParameterNames(names)` : Proporciona los nombres de los parámetros que vienen en la petición.

Más adelante veremos el uso que se puede hacer de este objeto para almacenar datos y enviarlos a una página JSP para que los procese.

11.2.2. HttpServletResponse

Un objeto `HttpServletResponse` proporciona dos métodos para devolver datos al usuario.

- El método `getWriter` devuelve un objeto de tipo `Writer`.
- El método `getOutputStream` devuelve un objeto de tipo `ServletOutputStream`.

Se utiliza el método `getWriter` para devolver datos en formato texto al usuario y el método `getOutputStream` para devolver datos binarios (ficheros). Debemos seleccionar la cabecera de datos HTTP antes de acceder a `Writer` o a `OutputStream`, para ello usaremos el método `setContentType` de la clase `HttpServletResponse` que permite seleccionar el tipo del contenido. También hay otros métodos útiles, como `sendRedirect()`, que redirige la aplicación a otra página, cuya URL se indica como parámetro.

El problema de usar los métodos anteriores de `HttpServletResponse` es que no se diferencia entre la generación de datos y presentación. Esta es la razón por la que usaremos un método llamado `forward` mediante el cual se pasa la respuesta a la página JSP que se desee, evitando que el Servlet deba escribirla.

Este método `forward` pertenece al objeto `RequestDispatcher` que se obtiene del contexto del Servlet (hablamos ahora de esto). Esto es, estando en un servlet, lo tendremos de la siguiente forma:

```
RequestDispatcher rd =
this.getServletContext().getRequestDispatcher("/ContextRoot/home.jsp");
rd.forward(request, response);
```

donde `/ContextRoot/home.jsp` es la dirección de la página JSP a la que redirigimos la salida, pero siendo una ruta relativa a la raíz del ServletContext.

También podría ser de la siguiente forma:

```
RequestDispatcher rd = request.getRequestDispatcher("/home.jsp");
rd.forward(request, response);
```

donde `/home.jsp` es la dirección de la página JSP a la que redirigimos la salida, pero siendo una ruta relativa a la solicitud actual HTTP.

Pero, ¿que es `RequestDispatcher`? 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. Esto permite que los servlets colaboren entre ellos, pudiendo intercambiar información entre sí.

En el caso de que simplemente quisieramos redirigirnos a otra página también podríamos con un método de response, pero la diferencia entre `request.getRequestDispatcher` y `response.sendRedirect` es:

- `getRequestDispatcher` es el salto interno del servidor, la información de la barra de direcciones permanece sin cambios y solo puede saltar a la página web en la aplicación web.

- `sendRedirect` es una redirección de página, la información de la barra de direcciones cambia, puede saltar a cualquier página web.

Veremos un ejemplo del uso de `forward` más adelante cuando tratemos la relación entre Servlets y formularios, pero ahora podéis encontrar un video realizando un ejemplo donde probaremos la captura de datos enviados por los clientes a través del uso de los métodos del objeto `HttpServletRequest` en el portal de la asignatura (denominado *Unidad 2 - Video 9 - Ejemplo captura de datos desde un Servlet*).

Podemos observar en el código anterior que tenemos una llamada al método `getServletContext()` que como hemos indicado obtiene el contexto del Servlet, pero, ¿qué es esto del contexto? El `ServletContext` como su nombre indica permite acceder a un Servlet a la información de su Contexto o dicho de otra manera a la información asociada con la propia Aplicación y que es común a todos los Servlets que despleguemos dentro de esa aplicación. Este contexto puede registrar eventos, obtener referencias de URL para ubicar recursos y establecer y almacenar atributos a los que otros servlets pueden acceder. Cada aplicación web implementada en el contenedor está asociada con un objeto de instancia de la interfaz `ServletContext`.

12. Ámbitos de los objetos

Los objetos definidos en un ámbito (también llamado `scope`) se usan para mantener en memoria esos objetos (en nuestro caso serán Javabeans o POJO's) disponibles según el ámbito. Evidentemente una vez son descargados de memoria ya no serán accesibles.

Hay cuatro ámbitos de objetos diferentes:

- `Application` : cuyo ámbito es la aplicación Web. Establece objetos en memoria que serán accesibles siempre por cualquier elemento de la aplicación JEE. Se accede a través de:

  ```
  ServletContext scopeApplication = this.getServletContext();
  scopeApplication.setAttribute(name, value);
  ```

  donde `name` es el nombre con el que se almacenará en la memoria y `value` el objeto que almacenamos.

- `Session` : establece objetos en memoria que se mantendrán durante toda la sesión (navegación del cliente) en que fueron creadas. Se accede a través de:

  ```
  HttpSession scopeSession = request.getSession();
  scopeSession.setAttribute(name, value);
  ```

  donde `name` es el nombre con el que se almacenará en la memoria y `value` el objeto que almacenamos.

- `Request` : establece objetos en memoria que se mantendrán desde el inicio de la petición de una página hasta su procesado completo. Se accede a través de:

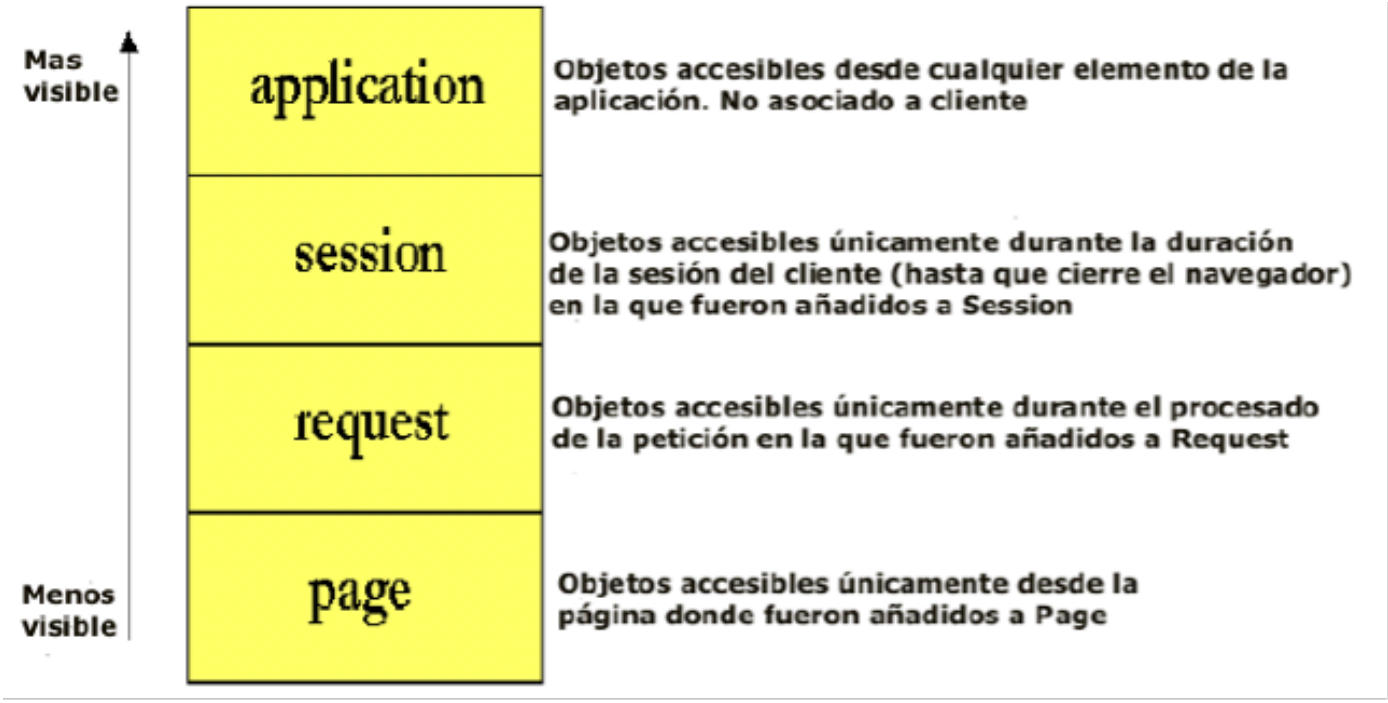
  ```
  request.setAttribute(name, value);
  ```

donde `name` es el nombre con el que se almacenará en la memoria y `value` el objeto que almacenamos.

- `Page` : establece objetos que se mantendrán únicamente en la página que fueron creados. Este ámbito solo tiene vigencia en las páginas JSP. Se accede a través de

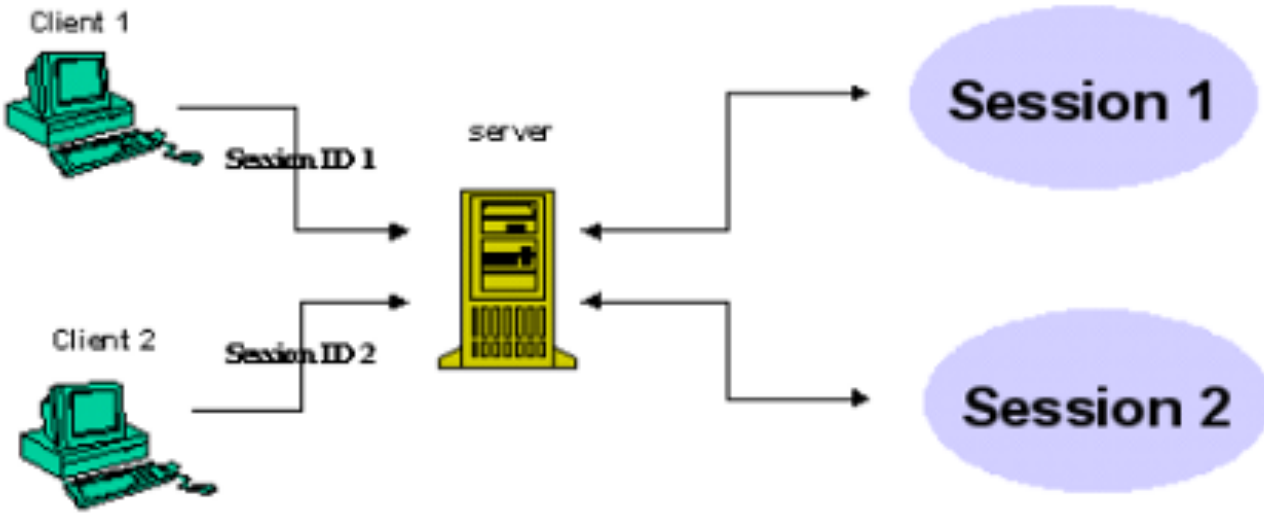
```
<jsp:useBean id="name" class="package.class" scope="page"/>
```

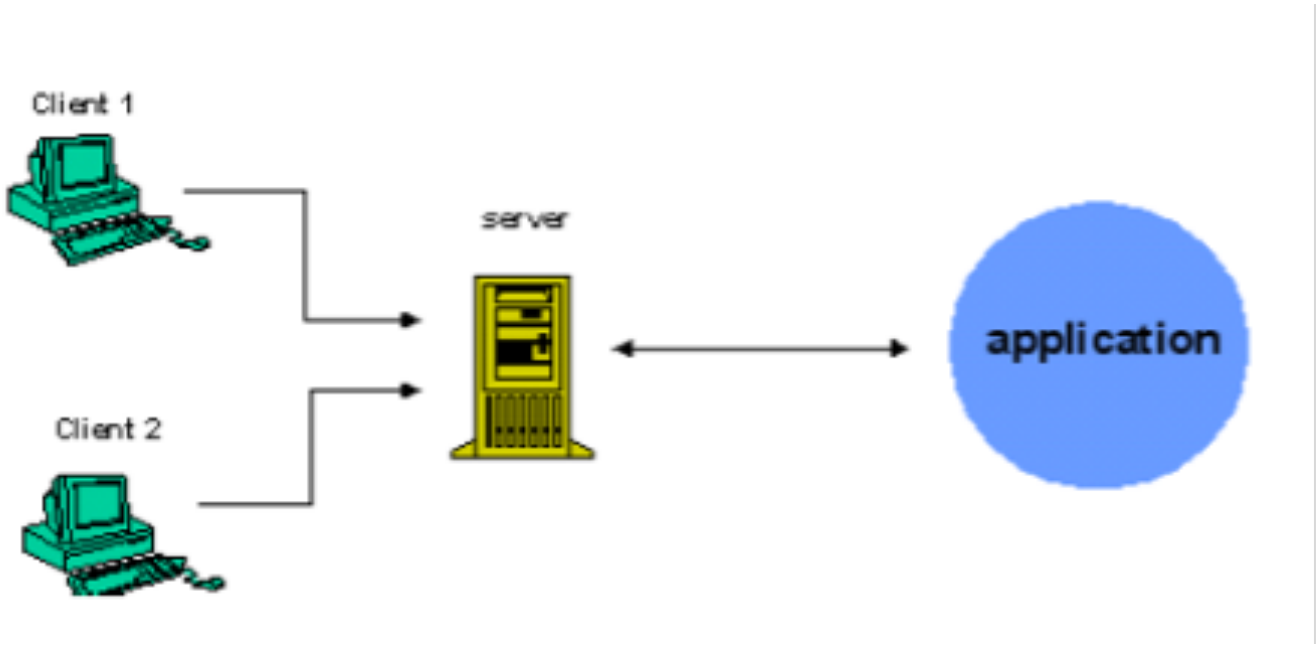
Los ámbitos `Application`, `Session` y `Request` permiten compartir objetos JavaBean entre Servlets y páginas JSP. De este modo, una vez el Servlet tenga un JavaBean con los datos preparados para presentar lo almacenará en uno de esos tres ámbitos mediante el método `setAttribute` correspondiente y pedirá mediante `forward` (lo veremos más adelante) que una página JSP presente esos datos de forma adecuada (recuperará los datos mediante el `getAttribute` correspondiente). Como podemos observar hemos logrado separar la obtención de datos (JavaBean) por los Servlet de su presentación en las páginas JSP (esto es conocido como patrón MVC, que estudiaremos en unidades posteriores).



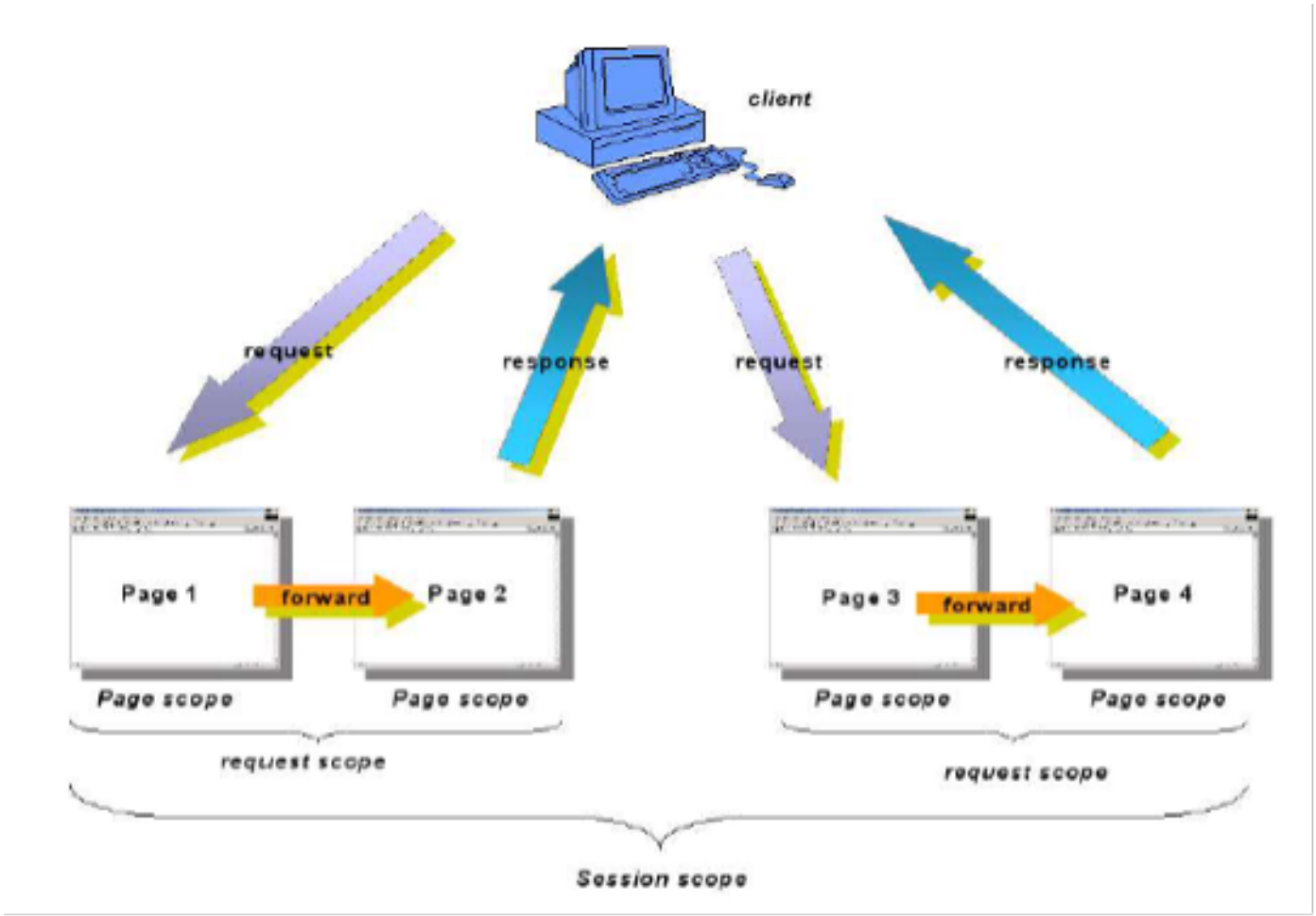
La figura muestra la relación entre los ámbitos `session`, `request` y `page`. Podemos observar que existe un ámbito `page` por cada página JSP. Por otro lado el ámbito de `request` comprende todas las páginas, Servlets y demás elementos involucrados en el proceso de ir de una página a otra, y el ámbito de `session` incluye todas las peticiones y respuestas que un cliente en particular realiza.

La siguiente figura muestra la diferencia entre ámbitos de sesión y de aplicación. Un ámbito de sesión corresponde a un cliente en particular mientras que un ámbito de aplicación es compartido por todos los clientes que acceden a una misma aplicación.





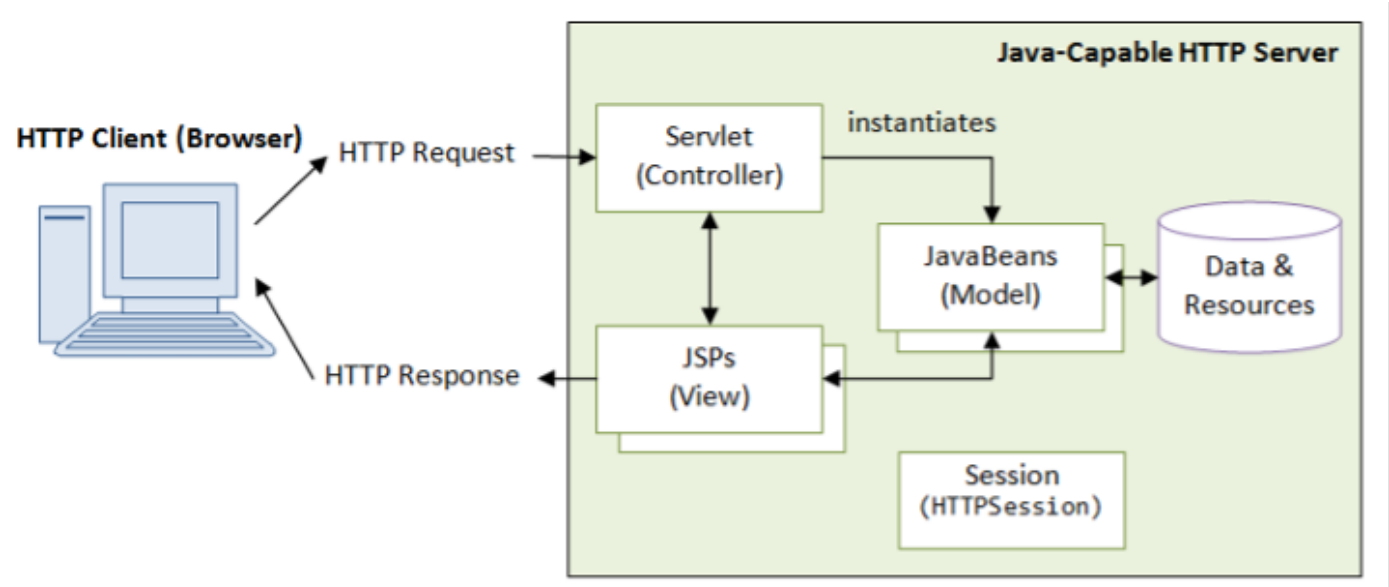
Como se ve en la siguiente figura, el ámbito `session` permite mantener el estado de un cliente a lo largo de su navegación. Otra posibilidad sería por medio de cookies, las cuales son un mecanismo que el servlet utiliza para mantener en el cliente una pequeña cantidad de información asociada con el usuario.



Podéis encontrar un video realizando un ejemplo donde trabajaremos sobre los ámbitos de los objetos en el portal de la asignatura (denominado *Unidad 2 - Video 10 - Ejemplo ámbitos de los objetos*).

13. Relación entre Servlets, JSPs y JavaBeans.

En este apartado vamos a explicar la utilidad final de todo lo explicado hasta ahora: documentos HTML, páginas JSP, JavaBean y Servlets. La relación existente la podemos ver rápidamente en la siguiente figura:



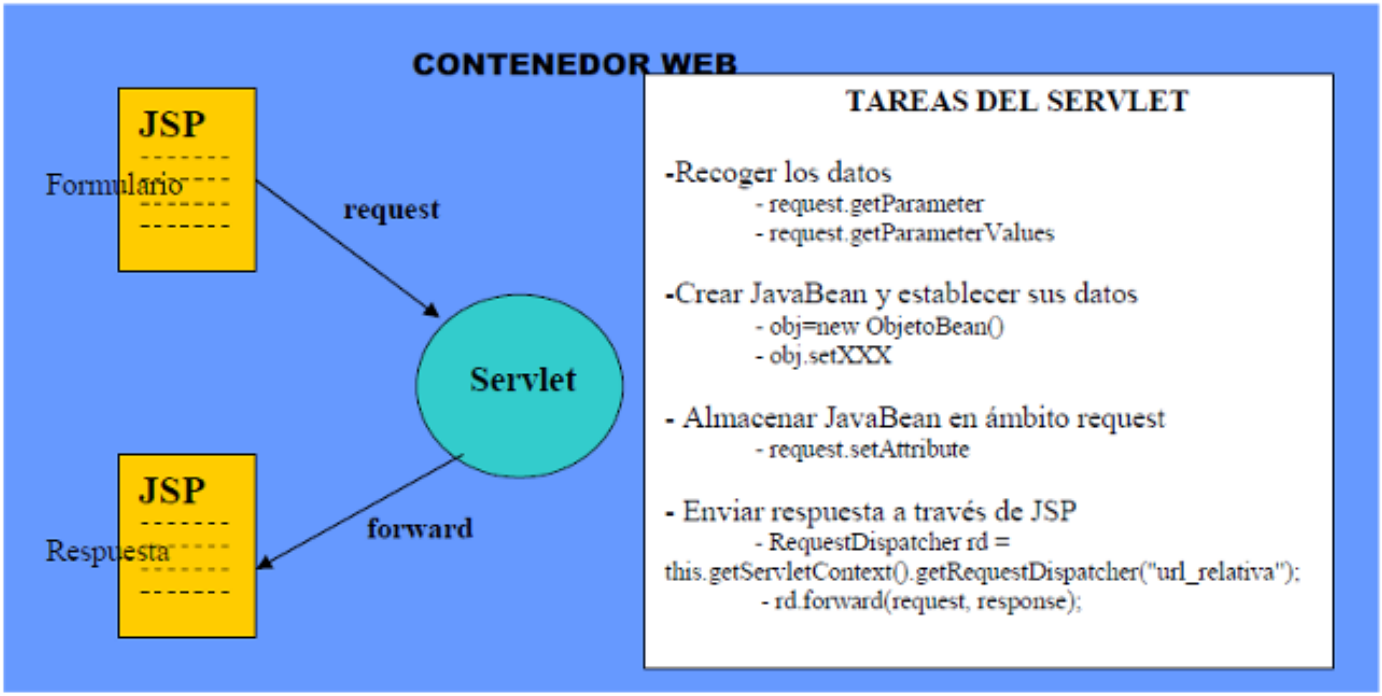
Con todo lo visto hasta ahora hemos trabajado en tecnología necesaria para poder implementar una primera aproximación del modelo vista controlador, donde el Servlet hace de controlador, el JavaBean de modelo y el JSP de vista.

Con todo ello ahora hemos de ser capaces de:

1. mostrar un formulario en pantalla a través de una página JSP
2. crear un objeto JavaBean
3. recoger en un Servlet los datos enviados y pasarlos a una instancia del JavaBean
4. almacenarlo en un ámbito de Request
5. mostrar la respuesta al cliente a través de otra página JSP

Para ello necesitaremos dos páginas JSP, una con el formulario y otra con la respuesta, y un Servlet capaz de recoger y procesar los datos, así como un JavaBean.

Veámoslo a través de la siguiente figura:



Podéis encontrar un video realizando un ejemplo donde trabajaremos dicha relación entre Servlet, JSP y JavaBean en el portal de la asignatura (denominado *Unidad 2 - Video 11 - Ejemplo Servlet, JSP y JavaBeans*).

14. Fichero web.xml

Las aplicaciones web de Java usan un archivo descriptor de implementación para determinar cómo se asignan las URL en los servlets, qué URL requieren autenticación y más información. Este archivo se llama `web.xml` y se encuentra en el WAR de la app en el directorio `WEB-INF/`. `web.xml`.

Cuando el servidor web recibe una solicitud para una aplicación específica, usa el descriptor de implementación (el fichero `web.xml`) a fin de asignar la URL de la solicitud al código que debe manejarla.

El archivo `web.xml` es el archivo de configuración más importante en toda la aplicación web y no solo nos permite esta asignación de URL's, sino que también permite:

- Establecer un recurso web como la página de inicio del sitio web.
- Asigne el programa de servlet a una dirección URL.
- Configurar oyentes para aplicaciones web.
- Configurar filtros para aplicaciones web,
- Configurar parámetros de contexto de aplicación web, configurar parámetros de sesión.
- Configurar spring, springMVC y otros marcos.

Un ejemplo de este fichero podría ser el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
    <servlet>
        <servlet-name>ServletScope1</servlet-name>
        <servlet-class>com.mycompany.ejemploambitosobjetos.ServletScope1</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>ServletScope2</servlet-name>
        <servlet-class>com.mycompany.ejemploambitosobjetos.ServletScope2</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>ServletScope3</servlet-name>
        <servlet-class>com.mycompany.ejemploambitosobjetos.ServletScope3</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServletScope1</servlet-name>
        <url-pattern>/ServletScope1</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>ServletScope2</servlet-name>
        <url-pattern>/ServletScope2</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>ServletScope3</servlet-name>
        <url-pattern>/ServletScope3</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```