

AQS

Advanced Quality Solutions

Web Publishing Frameworks

Febrero 2001

© Copyright Advanced Quality Solutions 2001

www.aqs.es

Índice

| | |
|----------------------------------|----------|
| WEB PUBLISHING FRAMEWORKS | 2 |
|----------------------------------|----------|

| | |
|----------------------------------|----------|
| WEB PUBLISHING FRAMEWORKS | 3 |
|----------------------------------|----------|

| | |
|----------------------|----------|
| COCOON Y XSPS | 5 |
|----------------------|----------|

| | |
|--|-----------|
| ESTRUCTURA DE COCOON | 6 |
| FUNCIONAMIENTO DE COCOON | 7 |
| EXTENSIBLE SERVER PAGES (XSP) | 7 |
| BREVE GUÍA DE PROGRAMACIÓN DE XSP | 8 |
| EJEMPLO: | 15 |
| USO DE TAGLIBS EN XSP. | 23 |
| Ventajas del uso de taglibs: | 23 |
| Cómo funcionan las taglibs | 24 |
| Cómo usar <i>taglibs</i> ya instaladas | 24 |
| Cómo definir una nueva etiqueta | 25 |
| WORKFLOW CON XSP | 27 |
| REFERENCIAS | 28 |

Antes de continuar...

Se asumen conocimientos de: XML, XSL

Recomendado: HTML, Java Servlets, JSPs

WEB PUBLISHING FRAMEWORKS

En nuestros días, cada vez está tomando mayor relevancia la presencia en la WEB. Cada día se demandan más servicios a través de Internet y aumenta el número de competidores. Todo este dinamismo requiere sistemas potentes en cuanto a servicios y contenidos, y a la vez con una gran capacidad de transformación para adaptarse continuamente a las nuevas necesidades impuestas por las estrategias de mercado.

Otra necesidad actual que complica aún más el problema es la de presentar información en diversos dispositivos y formatos. Seguramente a una empresa le gustaría que los datos de su producto estrella en su catálogo pudiesen ser visualizados desde un navegador web, desde un móvil con tecnología WAP o que se pudiesen imprimir con alta calidad para un catálogo en papel.

Imaginemos una web formada por decenas o cientos de páginas HTML que necesite ser renovada para reflejar la nueva imagen de la compañía. ¡Puede ser una auténtica pesadilla!. Técnicas como el uso de *Hojas de Estilo en Cascada* (CSS) ayudan pero no deja de ser todo un desafío que puede durar demasiado tiempo. Si además se pretende hacer versiones para distintos formatos se debería multiplicar el número de páginas.

Si el contenido se genera dinámicamente, por ejemplo mediante la consolidada tecnología de los *Java Servlets*¹, el problema pasa del diseñador web al desarrollador. Este debe retocar el código fuente de sus programas para cambiar la estética del resultado, con el riesgo de alterar todo el funcionamiento del sistema y esto podría llevar aún más tiempo hasta conseguir el cambio deseado. Para conseguir distintos formatos de presentación se debería aumentar el número de servlets o complicar la lógica interna de cada uno para que, en función de algún parámetro, se pueda cambiar la presentación.

La aparición de las *Java Server Pages* (JSP)² de la mano de Sun Microsystems supuso una mejora muy importante con respecto a los servlets en este sentido. Dos de las principales características de las JSP van orientadas a convertirlas prácticamente en vistas para datos que se generan con código fuera de la página:

- El uso de JSP Beans: Un buen diseño de JSP no debería contener apenas código, sólo el necesario para invocar otras clases que contienen toda la lógica y para formatear dentro de la página los datos obtenidos desde estas clases.
- El uso de librerías de etiquetas o *taglibs*: (A partir de la especificación JSP 1.1.) Permiten definir etiquetas XML que se corresponden con un fragmento de código. De esta forma se pueden construir páginas sin código que contienen esta clase de etiquetas.

A pesar de que la situación mejora, siguen existiendo algunos inconvenientes:

- Existe el riesgo de que un diseñador o un desarrollador de HTML pueda alterar el código embebido en la página, con lo que el mantenimiento se puede complicar.

¹ <http://java.sun.com/products/servlet>

² <http://java.sun.com/products/jsp>

- Para obtener distintos formatos hay modificar el código Java que formatea los datos, con lo que se requiere un programador para hacer el trabajo típico de un desarrollador de HTML / diseñador. Este es uno de los problemas que plantea el solapamiento de perfiles profesionales que conlleva el desarrollo de JSPs.
- Resulta muy tentador incluir demasiado código en la página sin utilizar en su lugar beans o taglibs.

Desde el principio se está hablando de la misma cosa, y no es otra que la separación entre contenido y presentación. Por un lado estaría bien tener productores de información que generen la información adecuada en un formato estándar y por otro lado poder definir distintas vistas de esos datos. Así un cambio de imagen sólo afectaría a ciertas vistas pero la lógica para generar la información no se ve alterada. Del mismo modo, la necesidad de tener disponibles los datos en un nuevo formato sólo implica la creación de una nueva vista.

Con esta idea surgen principalmente dos soluciones:

- Motores de plantillas: La idea es que si las JSP solo se usan para crear vistas de los datos generados por los JSPBeans, ¿para qué usar una sintaxis tan compleja?. Los motores de plantillas proponen diseñar vistas a modo de plantillas con unas sintaxis muy sencillas que permitan especificar cómo se obtienen los datos y cómo se colocan para obtener el resultado final. El objetivo es que el diseño de estas plantillas pueda ser llevado a cabo por diseñadores o desarrolladores de HTML con escasos conocimientos de programación de forma sencilla. Esta solución también tiene problemas:
 - No hay ninguna especificación ni ningún estándar, cada producto tiene su propia sintaxis.
 - A pesar de que las sintaxis para el control de datos son sencillas no deja de ser programación, y es difícil convencer a un diseñador para que trabaje con ella.
 - Los productos de este tipo no suelen ser demasiado eficientes comparados con otras soluciones.

Los productos de este tipo más populares a la fecha de este documento son WebMacro³ de Semiotek, Apache Velocity⁴ y FreeMarker⁵.

- XML/XSL Publishing Frameworks: En la actualidad se están convirtiendo en la mejor alternativa.
- Si se busca un formato con el que enviar los datos a las vistas, qué mejor que usar un estándar con las ventajas de XML⁶. Entre estas ventajas podemos destacar, en este contexto, sus capacidades de transformación mediante XSL⁷, sobre todo XSL-T. Así, podemos hablar de un *publishing framework* basado en XML/XSL como un sistema que:
 - Admite peticiones de documentos en diversos formatos.
 - Obtiene la información adecuada de diversas posibles fuentes de datos XML.
 - Obtiene la información sobre las transformaciones XSL necesarias y se las aplica a los datos.
 - Formatea el resultado final y lo sirve como respuesta a la petición inicial.

³ <http://www.webmacro.org/>

⁴ <http://jakarta.apache.org/velocity>

⁵ <http://freemarker.sourceforge.net/>

⁶ W3 Consortium Extensible Markup Language (XML) (<http://www.w3.org/XML/>)

⁷ Extensible Stylesheet Language (XSL) (<http://www.w3.org/TR/xsl/>)

Las ventajas de un sistema de publicación basado en XML/XSL son, entre otras:

- Separación limpia entre contenido y presentación.
- Permite separar claramente los papeles del programador y el diseñador.
- Proporciona una mejora muy notable del mantenimiento: Se puede realizar un cambio radical de imagen de todo un site web con tan solo modificar las hojas XSL y sin tocar ni una sola línea de código.
- A partir de un solo documento XML con el contenido, se pueden obtener páginas HTML para su presentación web, páginas WML para dispositivos WAP, documentos PDF para imprimir...
- Aunque no hay ningún estándar que regule como debe ser un sistema de publicación si que está basado en estándares con mucha fuerza en el mercado, por lo que es más sencillo pasar de usar uno a usar otro.
- Es compatible con el resto de tecnologías web como servlets, JSPs...

La desventaja principal es la poca madurez de la mayoría de los proyectos de este tipo y que aún no están del todo asentados.

Actualmente ya hay un amplio abanico de soluciones, que pasan desde complementos para Apache escritos en Perl como AxKit⁸ a JSPs productoras de XML manejadas por servlets que aplican a su resultado diversas transformaciones. Por esta última línea han surgido varias alternativas, casi siempre basadas en taglibs para manejar xml, pero no dejan de ser 'poco naturales'. Al respecto hay una propuesta muy interesante que forma parte del proyecto Apache Cocoon, que son las XSP (parecidas a las JSP pero con total integración con el manejo de XML).

Este campo de aplicación se está moviendo deprisa, y continuamente aparecen nuevas propuestas. Una lista de ellas se puede encontrar en <http://www.xmlsoftware.com/publishing/>.

El proyecto *open source* Apache Cocoon⁹ es la solución más madura y más adecuada de las que se estudiaron al inicio del proyecto y a la que vamos a dedicar especial atención en el resto del capítulo.

Cocoon y XSPs

Cocoon es un sistema de publicación electrónico basado en XML/XSL orientado a documentos.

Es 100% Java y está basado en estándares. Además es probablemente el framework de este tipo más maduro y reconocido. Como *publishing framework* aporta las ventajas descritas en el apartado anterior y además tiene características de valor añadido como:

- Open source.
- Altamente configurable y personalizable.

⁸ <http://axkit.org/>

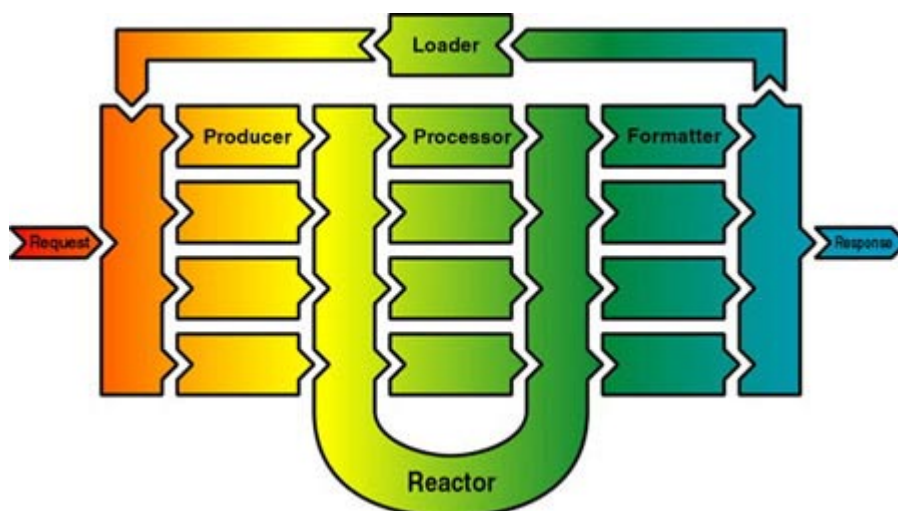
⁹ <http://xml.apache.org/cocoon/index.html>

- Incorpora la característica de poder escribir páginas de servidor aptas para el trabajo con XML (XSPs).
- Permite diferenciar el procesado del documento en función del dispositivo o tipo de software que realiza la petición.
- Incorpora un sistema de caché que permite un rendimiento muy elevado.

Cocoon puede trabajar como un programa en línea de comandos pero su uso normal es como Servlet para la publicación a través de la Web.

ESTRUCTURA DE COCOON

Desde el punto de vista estructural Cocoon se compone principalmente de los siguientes elementos:



10

- **Productores:** Son las fuentes del XML. Pueden ser desde ficheros XML estáticos hasta objetos Java que generen XML dinámicamente. Dentro de los productores cabe destacar las XSP (eXtensible Server Pages). Son parecidas a las JSP sólo que en lugar de compilar a un servlet, compilan a un Producer, que es la interfaz que debe implementar cada productor de XML que se integre con Cocoon.
- **Procesadores:** Son los encargados de tratar el XML de los productores y someterlo a diversos procesos consecutivos. Por ejemplo las XSP son transformadas a clases Java por un procesador, otro procesador permite hacer peticiones a bases de datos en función de instrucciones en XML y otro por ejemplo puede aplicar transformaciones XSL al XML.
- **El Reactor:** Es la pieza central encargada de extraer del xml generado por los productores las instrucciones de proceso que determinan que procesadores actuarán. Como estas instrucciones pueden ser cambiadas por los procesadores se puede alterar dinámicamente el flujo de proceso del xml.
- **Formateadores:** Son los encargados de recoger la representación interna del xml resultante de los procesadores y prepararlo para enviarlo como respuesta al cliente en el formato adecuado (texto, html, xml, wml...).

¹⁰ Imagen obtenida de la web oficial de Cocoon.

Cabe destacar que Cocoon es un producto muy abierto. Podemos crearnos nuestros propios productores, procesadores y formateadores e integrarlos perfectamente. Además podemos cambiar el parser XML y el motor de transformación XSL por el que más nos convenga.

Para aprender más acerca de cómo crear Producers y Formatters se puede visitar <http://xml.apache.org/cocoon/dynamic.html>

FUNCIONAMIENTO DE COCOON

A nivel funcional, el trabajo de Cocoon desde una petición del usuario a la devolución del documento final, a grandes rasgos, pasa por las siguientes fases :

- 1.- El usuario pide un documento.
- 2.- Se analiza la petición para saber a que productor de XML corresponde.
- 3.- El productor genera un documento XML.
- 4.- El reactor extrae las instrucciones de proceso del documento y se lo pasa al procesador adecuado. Y así sucesivamente hasta que no queden más instrucciones a procesar. Hay que tener en cuenta que los procesadores pueden añadir más instrucciones de proceso.
- 5.- El resultado le llega al formateador. Si es un documento final le aplica el formato solicitado según el tipo de documento y se le envía al cliente. Si lo que llega es código ejecutable (XSP compilada), el Loader lo recoge y lo coloca como productor empezando otra vez desde el paso 3.

Este sería un modelo muy simplificado, donde se ha omitido, por ejemplo la caché de Cocoon. Para aprender más sobre la caché de Cocoon ver la sección de [referencias](#) al final de este documento.

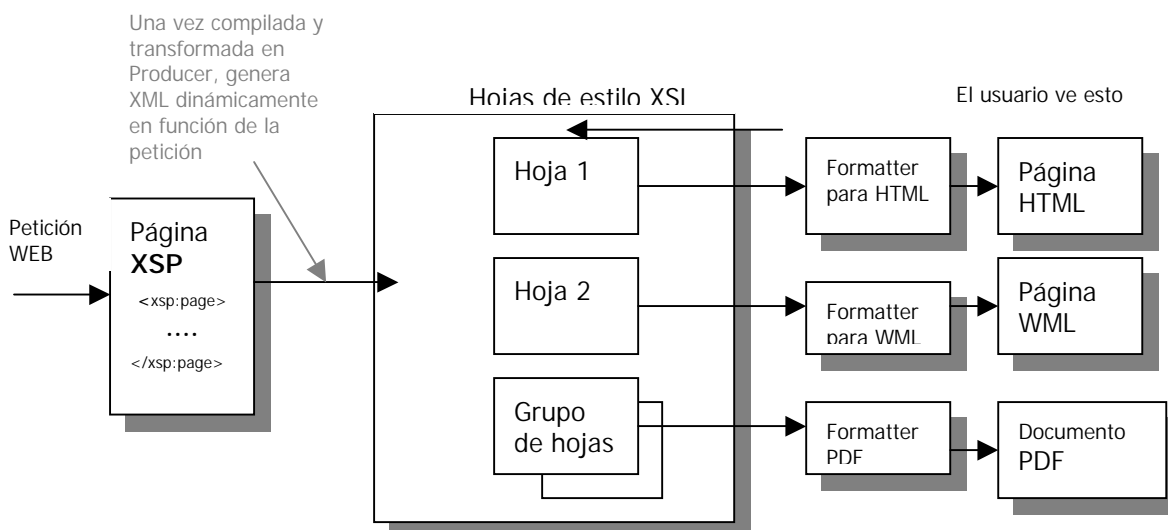
EXTENSIBLE SERVER PAGES (XSP)

Las páginas XSP vienen a solventar las dificultades de las JSP en el trabajo con XML.

Para leer una comparativa XSP vs JSP puede acudirse a: <http://www.oreilynet.com/lpt/a/620>

Una XSP no es más que un documento XML donde podemos incluir contenido estático y lógica para generar XML dinámicamente. Dentro del modelo de Cocoon serían productores.

Aquí vemos cómo a partir de un contenido se obtienen diversas vistas para distintos formatos.



Breve guía de programación de XSP

Debido a que la documentación oficial sobre XSP es algo escasa se incluirá en esta sección una breve guía de aprendizaje para comenzar a programar XSPs.

Una XSP es un **documento XML**¹¹ con ciertas características:

- ✓ Como documento XML que es, su primera línea debe ser:

```
<?xml version="1.0"?>
```

Donde además conviene especificar la codificación de caracteres:

```
<?xml version="1.0" encoding="UTF-8"?>
```

- ✓ Para que el XSPProcessor de Cocoon sepa que el documento procesado es un XSP es necesario añadir al principio una instrucción de proceso.

```
<?cocoon-process type="xsp"?>
```

- ✓ El elemento raíz del documento debe ser xsp:page. Tiene un atributo en el que se especifica el lenguaje de programación usado para la lógica embebida. De momento Cocoon sólo soporta Java¹². Además en el elemento raíz se declaran los espacios de nombres usados y como mínimo hay que declarar el espacio de nombres xsp. Con todo esto, el elemento raíz típico de una XSP tiene este aspecto:

```
<xsp:page language="java" xmlns:xsp="http://www.apache.org/1999/XSP/Core" >
```

- ✓ El primer elemento que aparezca dentro de xsp:page y no esté en el espacio de nombres será el elemento raíz del documento generado.

Y con estas primeras nociones se podría construir una XSP, absolutamente trivial, pero válida:

```
<?xml version="1.0" encoding="UTF-8"?>
<?cocoon-process type="xsp"?>
<xsp:page language="java" xmlns:xsp="http://www.apache.org/1999/XSP/Core">
  <root>
    Hello World XSP Page
  </root>
</xsp:page>
```

Esta página, tras ser compilada a una clase produciría siempre el mismo XML resultante:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  Hello World XSP Page
</root>
```

¹¹ Por ser un documento XML debe estar bien formado y demás. El hecho de que las XSP sean documentos XML proporciona ciertas ventajas, sobre todo en cuanto a sus posibilidades de transformación. (Vease apartado referente a XSP taglibs).

Dentro de `xsp:page` se pueden usar etiquetas para generar un contenido algo más dinámico. Algunas de ellas son propias de XSP (las que pertenecen al espacio de nombres `xsp`) y otras son definidas por el usuario. A continuación se detallan las etiquetas nativas:

- **<xsp:structure>**: Engloba directivas globales a nivel de programa. Debe colgar directamente de `xsp:page`.
- **<xsp:include>**: Permiten importar módulos de forma según el lenguaje. En el caso de Java el contenido del `xsp:include` es transformado en imports.

Por ejemplo `<xsp:include>java.util.Date</xsp:include>` se transformaría en:

```
import java.util.Date ;
```

- **<xsp:logic>**: Especifica que su contenido debe ser considerado como código embebido en la página.
- **<xsp:content>**: Usado dentro de un bloque `<xsp:logic>` indica que el contenido pasa a ser considerado como XML para el documento final.
- **<xsp:expr>**: Se usa para evaluar variables o expresiones y colocar su resultado en el documento generado.
 - Si el contenido de `xsp:expr` es de un tipo primitivo, se inserta en un nodo de texto su representación como cadena.
 - Si el resultado de evaluar la expresión es instancia de alguna subclase de `org.w3c.dom.Node` se insertan si pertenecen al mismo documento o se clonan e insertan si pertenecen a un documento distinto (para cambiar el Document propietario).
 - Si la expresión es de tipo `String` se insertará dentro de un nodo de texto. Nótese que si la cadena representa un fragmento de XML éste no será insertado como tal, sino como texto normal de forma que los caracteres ">", "<" y "&" serán escapados.
 - Si la expresión es un `Array`, se tratará como un `org.w3c.dom.DocumentFragment` y se le añadirá cada elemento del array como hijo aplicándole estas mismas reglas.
 - Si el resultado no es de ninguno de estos tipos se insertará como nodo de texto basado en un `toString()` del objeto.
- **<xsp:element>**: Permite insertar un nuevo elemento en el documento cuyo nombre será el especificado en el atributo `name` de esta etiqueta. El contenido de la etiqueta será el contenido del elemento. Es útil si el nombre del elemento se decide dinámicamente.

Por ejemplo:

```
<clock>
  <xsp:element name="time"><xsp:content><xsp:expr>new
Date()</xsp:expr><xsp:content></xsp:element>
</clock>
```

se traducirá en el documento final en:

```
<clock>
  <time>Wed Jul 04 15:12:49 CEST 2001</time>
</clock>
```

- **<xsp:attribute>**: Permite añadir un atributo al nodo actual. El nombre del atributo será el especificado en el atributo `name` de la etiqueta y el valor será su contenido. Por ejemplo:

```
<clock>
  <xsp:attribute name="time"><xsp:expr>new Date()</xsp:expr></xsp:attribute>
</clock>
```

¹² El proyecto AxKit (<http://axkit.org/>) admite XSP con language PERL.

se traducirá en el documento final en:

```
<clock time="Wed Jul 04 15:12:49 CEST 2001" />
```

- **<xsp:pi>**: Permite insertar una instrucción de proceso en el documento generado. El nombre de la instrucción de proceso vendrá determinado por el atributo `target` de esta etiqueta. El contenido será colocado dentro del cuerpo de la instrucción de proceso. Por ejemplo:

```
<xsp:pi target="process">
  href="<xsp:expr>request.getQueryString()</xsp:expr>"
  type="forward"
</xsp:pi>
```

Se traducirá en:

```
<?process href="http://www.aqs.com/xsp/example" type="forward"?>
```

- **<xsp:comment>**: Permite insertar un texto a modo de comentario dinámico en el documento generado. El texto del comentario será el contenido de esta etiqueta. Por ejemplo:

```
<xsp:comment> Bla Bla Bla </xsp:comment>
```

Se traducirá en:

Bla Bla Bla

Aunque pueda parecer más razonable algo como `<!-- Bla Bla Bla -->`, este es el comportamiento actual de Cocoon (v1.82).

Ahora sólo queda por ver cómo pueden colocarse estas etiquetas. Puesto que en una XSP no se sabe a priori que elementos van a aparecer, difícilmente se puede hacer una DTD o un schema rigurosos, por lo que se va a dar una notación alternativa que debe entenderse como orientativa más que como determinante.

| Etiqueta | Contenido válido | Atributos |
|--------------------|--|---|
| xsp:page | xsp:structure (?), xsp:logic (?), <i>[user root]</i> | language indent-result: yes no xmlns:xsp xml:space: default preserve <i>[user name_space definitions]</i> |
| xsp:structure | xsp:dtd (?), xsp:include (*) | |
| xsp:dtd | <i>[texto]</i> | |
| xsp:include | <i>[texto]</i> | |
| <i>[user root]</i> | (<i>[texto]</i> xsp:attribute(*) xsp:logic (*) xsp:element (*) xsp:expr (*) xsp:pi (*) xsp:comment <i>[user tag]</i> (*))* | <i>[user root attributes]</i> |
| xsp:attribute | (<i>[texto]</i> xsp:expr (*))* | name xml:space: default preserve |
| xsp:logic | (<i>[texto (código embebido)]</i> xsp:content (*) xsp:pi (*) xsp:comment)* | xml:space: default preserve |
| xsp:content | (| |

| | | |
|-------------------|--|------------------------------|
| | <i>[texto (XML)]</i> xsp:logic (*) xsp:expr(*) xsp:element (*) xsp:pi (*) xsp:comment (*) <i>[user tag]</i> (*))* | |
| xsp:pi | (<i>[texto]</i> xsp:expr)* | target |
| xsp:comment | (<i>[texto]</i> xsp:expr(*))* | |
| xsp:element | (<i>[texto]</i> xsp:attribute (*) xsp:element (*) xsp:logic (*) xsp:expr (*) <i>[user tag]</i> (*))* | name |
| xsp:expr | <i>[texto]</i> | |
| <i>[user tag]</i> | (<i>[texto]</i> xsp:attribute(*) xsp:logic (*) xsp:element (*) xsp:expr (*) xsp:pi (*) xsp:comment <i>[user tag]</i> (*))* | <i>[user tag attributes]</i> |

(?) → 0 ó 1 | (+) → 1 ó más | (*) → 0 ó más | Si no, 1

El uso típico de estas etiquetas es:

- Una etiqueta `<xsp:page>` después de las instrucciones de proceso. Este será el elemento raíz de la XSP (no del documento generado).
- Una etiqueta `<xsp:structure>` que contiene la lista de importaciones mediante etiquetas `<xsp:include>`.
- A continuación puede venir una etiqueta `<xsp:logic>` donde se escribirá la lógica a nivel de clase (es decir, definiciones de atributos y de métodos).
- Después ha de venir una etiqueta definida por el usuario. Esta será la raíz del documento generado y su contenido pasará a formar parte del método `populateDocument()` de `XSPPage`. Dentro de la etiqueta `xsp:page` sólo puede haber una etiqueta *no xsp*.
- Dentro de la raíz de usuario se puede escribir en dos modos para generar el documento final: En modo XML y en modo código embebido.
 - En el modo XML se pueden escribir directamente etiquetas XML que pasarán a formar parte del documento generado. Este es el modo por defecto dentro del elemento raíz definido por el usuario. Para pasar a modo XML se introduce una etiqueta `<xsp:content>`. Si en este modo se quiere insertar el resultado de evaluar una expresión o una variable definidos en un bloque de lógica se debe usar una etiqueta `<xsp:expr>`. En este modo también se pueden insertar elementos y atributos usando `<xsp:element>` y `<xsp:attribute>`.

- En el modo de código embebido se puede escribir código fuente usando la sintaxis del lenguaje elegido en el atributo `language` del elemento `xsp:page`, normalmente Java. Este código pasará a formar parte directamente del código fuente generado para la `XSPPage`. Para pasar a este modo hay que usar una etiqueta `<xsp:logic>`. No hay que olvidar que una XSP es un documento XML y como tal tiene ciertos caracteres reservados que hay que escapar. Estos son:

- "<" → "<";
- ">" → ">";
- "&" → "&";

Así, por ejemplo, para escribir:

```
if ( ( a > 10 ) && ( a < 20 ) ) {  
    . . .  
}
```

se tendría que escribir en la XSP:

```
if ( ( a &gt; 10 ) &amp;&amp; ( a &lt; 20 ) ) {  
    . . .  
}
```

o, de forma alternativa:

```
<![CDATA[  
if ( ( a > 10 ) && ( a < 20 ) ) {  
    . . .  
}  
]]>
```

Las XSP proveen de ciertas facilidades al desarrollador:

Ya se incluyen por defecto muchos *imports* frecuentes:

- `java.io.*;`
- `java.util.*;`
- `org.w3c.dom.*;`
- `org.xml.sax.*;`
- `javax.servlet.*;`
- `javax.servlet.http.*;`
- `org.apache.cocoon.parser.*;`
- `org.apache.cocoon.producer.*;`
- `org.apache.cocoon.framework.*;`
- `org.apache.cocoon.processor.xsp.*;`

También se instancian objetos corrientes para que estén ya en la página listos para ser usados. Estos son:

- `request`. El objeto `HttpServletRequest` que encapsula la petición. Se trata de un *wrapper* del objeto real que recibe Cocoon, pero con la misma funcionalidad.
- `response`. Un wrapper del objeto `HttpServletResponse` que recibe Cocoon. Ofrece todas las funciones del objeto real salvo el acceso al `ServletOutputStream` para preservar la consistencia de la salida manejada por Cocoon.
- `session`. El objeto `HttpSession`.
- `servletContext`. El objeto `ServletContext` estándar.
- `document`. El `org.w3c.Document` donde se van insertando todos los elementos hasta conformar el resultado final.
- `xspGlobal`. Es un contenedor de información compartida a nivel de aplicación. Su existencia viene justificada porque el objeto `servletContext` de versiones del JSDK anteriores a la 2.2 no admitían su uso para mantener información compartida a nivel de aplicación. En futuras versiones, este objeto será depreciado.
- `xspNodeStack`. Una pila (`java.util.Stack`) para controlar la anidación de los elementos del documento.
- `xspCurrentNode`. El `org.w3c.Node` actual.
- `xspParentNode`. El `org.w3c.Node` padre del nodo actual.
- `xspParser`. Un DOM parser proporcionado por Cocoon.

Una vez conocidos los elementos que puede contener una XSP y cómo pueden combinarse y el entorno de objetos y clases accesibles, se pasará a explicar aspectos interesantes para el desarrollador sobre cómo funciona internamente una XSP.

La primera vez que se invoca una XSP, comienza el proceso de transformaciones del XML hasta llegar al código fuente de la `XSPPage`. Se produce una transformación por cada espacio de nombres definido en el elemento `xsp:page` que se corresponda con una *logicsheet*¹³ definida en el `cocoon.properties`. Esto se verá con mayor detalle en el apartado dedicado a las taglibs. La última transformación convierte el XML a código fuente en el lenguaje indicado (por el momento Java). Este código fuente incluye:

- Un nombre de paquete generado por Cocoon.
- Una lista de *imports* mezcla de los predefinidos y de los definidos por el usuario con `xsp:include`.
- Un nombre de clase generado por Cocoon. Esta clase extiende `XSPPage`.
- Si colgando de `xsp:page` cuelga un bloque `xsp:logic`, esta lógica se introducirá tal cual dentro de la clase. Por lo que se puede aprovechar para definir atributos y métodos.

¹³ Este es el nombre que da Cocoon a las taglibs.

- El contenido del elemento raíz definido por el usuario será transformado en código para generar XML dentro del método `populateDocument()`. Este método declara que puede lanzar `Exception`, por lo que no será necesario introducir este código dentro de un try-catch, ya que estas excepciones las recoge Cocoon para generar una página de error. Aunque no sea necesario es recomendable controlar en cada momento estos errores y no permitir que aparezca la pantalla de error de Cocoon.

Tras la generación del código fuente se procede a la compilación de la página. Tanto el código fuente como el compilado son guardados en sendos archivos según la estructura de paquetes generada a partir del directorio `repository`¹⁴ en el raíz del servidor.

Posteriormente Cocoon invoca el método `populateDocument()` de la `XSPPage` para obtener el XML resultante.

Este XML puede ser reprocesado por Cocoon si incluye más instrucciones de proceso. El proceso más habitual es la transformación con ayuda de XSLT, aunque también se pueden usar instrucciones de formateo y de acceso a bases de datos.

Para asignar una XSL al resultado de la XSP se debe incluir 2 instrucciones de proceso especiales después de la instrucción de proceso de XSP:

```
<?cocoon-process type="xslt"?>
<?xml-stylesheet type="text/xsl" href="<ruta_archivo_xsl>"?>
```

También se puede asignar de forma dinámica desde el código de la página:

```
<xsp:page .....>
  <root>
    <xsp:logic>
      Node rootNode = xspCurrentNode;
      String MY_XSL = "myXSL.xsl";
      . . .//more code and more XML
      document.insertBefore(
        document.createProcessingInstruction(
          "cocoon-process",
          "type=\"xslt\""
        ), rootNode
      );

      document.insertBefore(
        document.createProcessingInstruction(
          "xml-stylesheet",
          "href=\""+MY_XSL+"\" type=\"text/xsl\""
        ), rootNode
      );
      . . . //more code and more XML
    </xsp:logic>
  </root>
</xsp:page>
```

Por último se describe un ejemplo de XSP:

¹⁴ `repository` es el nombre por defecto, pero se puede cambiar en el `cocoon.properties`.

Ejemplo

Código de la XSP (demo.xsp):

```

<?xml version="1.0" encoding="UTF-8"?>
<?cocoon-process type="xsp"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet type="text/xsl" href="demo.xsl"?>
<xsp:page language="java" xmlns:xsp="http://www.apache.org/1999/XSP/Core">

  <xsp:structure>
    <xsp:include>java.text.SimpleDateFormat</xsp:include>
  </xsp:structure>

  <xsp:logic>
    <!--Esta lógica va a nivel de clase-->
    private Date start = new Date();
    private static SimpleDateFormat dateFormat = new SimpleDateFormat();

    private String getStartDate(){
      return dateFormat.format(start);
    }
  </xsp:logic>

  <output><!--Elemento raíz del resultado-->
    <xsp:logic><!--Esta lógica va en el método populateDocument() -->
      String a = request.getParameter( "a");
      String b = request.getParameter( "b");
      Date now = new Date();
      <xsp:content>
        <titulo> Esto es una demostración de XSP</titulo>
        <parrafo>Se han introducido los parámetros a=[<xsp:expr>a</xsp:expr>]
y b=[<xsp:expr>b</xsp:expr>]</parrafo>
        <parrafo>El proceso de la pagina comenzó en el instante:
<xsp:expr>getStartDate()</xsp:expr></parrafo>
      </xsp:content>
    </xsp:logic>
  </output>

</xsp:page>

```

Instrucción que indica que se debe procesar como XSP

Instrucción que indica que después se debe aplicar una transformación XSLT

La lógica no tiene por qué estar en Java

Se debe cargar el espacio de nombres para las etiquetas de control XSP

Clase generada por Cocoon para esta XSP

```

package _D._Allaire._JRun31._servers._web_construplaza._default_app._admintool;

import java.io.*;
import java.net.*;
import java.util.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.servlet.*;
import javax.servlet.http.*;

import org.apache.cocoon.parser.*;
import org.apache.cocoon.producer.*;
import org.apache.cocoon.framework.*;

import org.apache.cocoon.processor.xsp.*;
import org.apache.cocoon.processor.xsp.library.*;

/* User Imports */

import java.text.SimpleDateFormat;

public class _demo extends XSPPage {
    /* User Class Declarations */

private Date start = new Date();
private static SimpleDateFormat dateFormat = new SimpleDateFormat();

private String getStartDate(){
    return dateFormat.format(start);
}

public void populateDocument(
    HttpServletRequest request,
    HttpServletResponse response,
    Document document
    )
    throws Exception
    {
// Node stack logic variables
    Node xspParentNode = null;
    Node xspCurrentNode = document;
    Stack xspNodeStack = new Stack();

```

Paquete generado por Cocoon

Imports por defecto

Imports por xsp:include

Lógica a nivel de clase del xsp:logic que cuelga de xsp:page

Lógica para populateDocument() que comienza después del raíz del usuario

```
// Make session object readily available
    HttpSession session = request.getSession(false);

    document.appendChild(
        document.createProcessingInstruction(
            "cocoon-process",
            "type=\"xslt\""
        )
    );

    document.appendChild(
        document.createProcessingInstruction(
            "xml-stylesheet",
            "type=\"text/xsl\" href=\"demo.xsl\""
        )
    );

xspParentNode = xspCurrentNode;
xspNodeStack.push(xspParentNode);
xspCurrentNode =
    document.createElement("output");
xspParentNode.appendChild(xspCurrentNode);

xspCurrentNode.appendChild(
    document.createTextNode("\n\t\t")
);

    String a = request.getParameter("a");
    String b = request.getParameter("b");
    Date now = new Date();

xspCurrentNode.appendChild(
    document.createTextNode("\n\t\t\t\t")
);

xspParentNode = xspCurrentNode;
xspNodeStack.push(xspParentNode);
xspCurrentNode =
    document.createElement("titulo");
xspParentNode.appendChild(xspCurrentNode);

xspCurrentNode.appendChild(
    document.createTextNode(" Esto es una demostración de XSP")
);
```

```
((Element) xspCurrentNode).normalize();
xspCurrentNode = (Node) xspNodeStack.pop();

xspCurrentNode.appendChild(
    document.createTextNode("\n\t\t\t\t\t")
);

xspParentNode = xspCurrentNode;
xspNodeStack.push(xspParentNode);
xspCurrentNode =
    document.createElement("parrafo");
xspParentNode.appendChild(xspCurrentNode);

xspCurrentNode.appendChild(
    document.createTextNode("Se han introducido los parámetros a=[")
);

    xspCurrentNode.appendChild(
        xspExpr(a, document)
    );

xspCurrentNode.appendChild(
    document.createTextNode("] y b=[")
);

    xspCurrentNode.appendChild(
        xspExpr(b, document)
    );

xspCurrentNode.appendChild(
    document.createTextNode("]")
);

((Element) xspCurrentNode).normalize();
xspCurrentNode = (Node) xspNodeStack.pop();

xspCurrentNode.appendChild(
    document.createTextNode("\n\t\t\t\t\t")
);

xspParentNode = xspCurrentNode;
xspNodeStack.push(xspParentNode);
xspCurrentNode =
    document.createElement("parrafo");
xspParentNode.appendChild(xspCurrentNode);

xspCurrentNode.appendChild(
```

```
document.createTextNode("El proceso de la pagina comenzó en el instante: ")
);

xspCurrentNode.appendChild(
    xspExpr(getStartDate(), document)
);

((Element) xspCurrentNode).normalize();
xspCurrentNode = (Node) xspNodeStack.pop();

xspCurrentNode.appendChild(
    document.createTextNode("\n\t\t\t")
);

xspCurrentNode.appendChild(
    document.createTextNode("\n\t")
);

((Element) xspCurrentNode).normalize();
xspCurrentNode = (Node) xspNodeStack.pop();

}
}
```

XML generado por la XSP compilada para la petición: demo.xsp?a=1111&b=2222

```
<output>
  <titulo> Esto es una demostración de XSP </titulo>
  <parrafo>
    Se han introducido los parámetros a=[1111] y b=[2222]
    El proceso de la pagina comenzó en el instante: 5/07/01 14:31
  </parrafo>
</output>
```

Hoja de estilo (demo.xsl):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<!-- *****-->
  <xsl:template match="output">
    <html>
      <head>
        <title>Demo XSP</title>
      </head>
      <body bgcolor="#FFFFFF">
        <h1><xsl:value-of select="titulo"/></h1>
        <hr/>
        <xsl:apply-templates select="parrafo"/>
      </body>
    </html>
  </xsl:template>
<!-- *****-->
  <xsl:template match="parrafo">
    <h3><xsl:value-of select="."/></h3>
  </xsl:template>
<!-- *****-->
</xsl:stylesheet>
```

Resultado html (demo.xsp?a=1111&b=2222).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN" "http://www.w3.org/TR/REC-
html40/strict.dtd">

<html>
<head>
  <title>Demo XSP</title>
</head>
<body bgcolor="#FFFFFF">
  <h1> Esto es una demostraci&ocute;n de XSP</h1>
  <hr>
  <h3>Se han introducido los par&acutemetros a=[1111] y b=[2222]</h3>
  <h3>El proceso de la pagina comenz&ocute; en el instante: 5/07/01 14:31</h3>
</body>
</html>
```

El formatter para HTML (por defecto) coloca la cabecera adecuada y sustituye caracteres especiales por las entidades adecuadas.

Vista en el navegador.

Esto es una demostración de XSP

Se han introducido los parámetros a=[1111] y b=[2222]

El proceso de la pagina comenzó en el instante: 5/07/01 14:31

Con este ejemplo da una idea de cómo son las XSP y para qué sirven (sobre todo si el lector tiene conocimientos de XML, XSL y JSPs). Para aprender más sobre XSPs véase la sección de [referencias](#) al final del capítulo, aunque la documentación a fecha de hoy es más bien escasa.

Uso de taglibs en XSP.

Las XSPs permiten, al igual que las JSPs asociar código con una etiqueta xml. Por ejemplo, podría asociarse: `<myPrefix:getDate format="yy/MM/dd hh:mm:ss aa"/>` con el código para obtener una fecha, de forma que cuando se genere el fuente de la XSP aparezca éste en el lugar donde estaba la etiqueta.

Estas etiquetas pueden ser agrupadas formando librerías de etiquetas. A estas librerías las llamaremos *taglibs*.

En las XSP las taglibs se definen como hojas de transformación XSL.

VENTAJAS DEL USO DE TAGLIBS:

- Potencian la reutilización de código usado frecuentemente.
- Permiten limpiar de código las XSP sustituyendo la mayoría del código embebido por etiquetas. Esto mejora el mantenimiento y la legibilidad de las páginas.
- Permiten separar más los papeles en el desarrollo de XSP. Véase el apartado [Workflow con XSP](#). Esto permite un mayor paralelismo en el trabajo.
- Cuando ya se disponga de una buena biblioteca de taglibs se acelerará el desarrollo de páginas.
- Debido a que están basadas en XSL permiten, además de simplemente añadir código a la página, transformarla con gran potencia. Un ejemplo de las ventajas de que proporciona el hecho de que sean hojas de transformación XSL es que podemos definir etiquetas para invocar de forma sencilla métodos con firmas complejas. Si preparamos valores por defecto para cada argumento podemos comprobar que atributos se nos pasan en la etiqueta y para los que falten colocar el valor por defecto.

Así, para un método `longMethod(a1, a2, a3, a4, a5, a6, a7, a8)` donde la mayoría de sus argumentos son usado habitualmente con los mismos argumentos podemos preparar una etiqueta que admita todos los argumentos en forma de atributos pero que si falta alguno ponga un valor típico:

En la XSP se escribirá:

```
MyClass myVar = <myPrefix:longMethod a1="a1Value" a6="this" /> ;
```

Con la taglib adecuada, en el código fuente aparecerá:

```
MyClass myVar = longMethod( "alValue",
                             null,
                             System.currentTimeMillis(),
                             "",
                             -1,
                             this,
                             null,
                             null);
```

Si quisiésemos dar una facilidad semejante desde Java tendríamos que definir una signatura por cada combinación de argumentos.

Otro ejemplo de potencia es que una etiqueta se puede sustituir por más etiquetas de otra taglib, o incluso eliminar contenido XML.

CÓMO FUNCIONAN LAS TAGLIBS

En Cocoon existe un mapeo entre espacios de nombres XML y hojas de transformación XSL.

Cuando se invoca una XSP por primera vez esta tiene que ser compilada a un productor de XML (actualmente una clase Java que extiende XSPPage.). Es durante este proceso cuando se usan las taglibs.

La primera fase es generar el código fuente. Para ello se le aplican transformaciones XSL. Si en la XSP aparecen etiquetas con espacios de nombres se comprueba si están mapeados con una hoja de transformación XSL (taglib). Si es así se le aplica esta transformación, si no se considera que es contenido estático de la página y se conserva tal cual.

Cuando todos los espacios de nombres están procesados se realiza la última transformación que traduce el XML al código Java encargado de generarlo (si se ha especificado Java como lenguaje en la cabecera de la XSP).

CÓMO USAR *TAGLIBS* YA INSTALADAS

Para ver las librerías instaladas podemos examinar el fichero de configuración de Cocoon `cocoon.properties`. En su sección referente al `XSP Processor` se pueden ver líneas de este estilo:

```
processor.xsp.logicsheet.util.java =
resource://org/apache/cocoon/processor/xsp/library/java/util.xsl
```

En esta línea se especifica que para el lenguaje Java, la hoja de transformación asociada al espacio de nombres `"util"` es `//org/apache/cocoon/processor/xsp/library/java/util.xsl`.

Si queremos usar alguna etiqueta definida en esta taglib deberemos declarar en el elemento raíz de la xsp el espacio de nombres `util`. Lo podemos copiar de la declaración de espacios de nombres de la taglib. En este caso del archivo `util.xsl`.

```
<xsp:page language="java"
  xmlns:xsp="http://www.apache.org/1999/XSP/Core"
  xmlns:util="http://www.apache.org/1999/XSP/Util" >
```

Una vez hecho esto ya se puede usar cualquiera de las etiquetas definidas en la taglib. Por ejemplo vamos a incluir una fecha en nuestra página:

Esta es la definición de lo que se hace con la etiqueta en la taglib.

```
<xsl:template match="util:time">
  <xsl:variable name="format">
```

```

<xsl:choose>
  <xsl:when test="@format"><xsl:value-of select="@format"/></xsl:when>
  <xsl:when test="util:format">
    <xsl:call-template name="get-nested-content">
      <xsl:with-param name="content" select="util:format"/>
    </xsl:call-template>
  </xsl:when>
</xsl:choose>
</xsl:variable>
<xsp:expr>
  XSPUtil.formatDate(
    new Date(),
    String.valueOf(<xsl:copy-of select="$format"/>).trim()
  )
</xsp:expr>
</xsl:template>

```

En la XSP escribiremos:

```
<util:time format="yy/MM/dd hh:mm:ss aa"/>
```

O en este caso si analizamos el código xsl vemos que también admite el parámetro como elemento en lugar de sólo como atributo:

```

<util:time>
  <util:format>yy/MM/dd hh:mm:ss aa</util:format>
</util:time>

```

CÓMO DEFINIR UNA NUEVA ETIQUETA

Por un lado debemos construir la XSL para tratar la etiqueta. Debe estar diseñada cuidadosamente para que respete el contenido de la página y sólo afecte a las etiquetas adecuadas, o de lo contrario se podría modificar contenido que en principio no tiene relación con la etiqueta. Este es, bajo mi punto de vista, el principal problema que plantean las taglibs definidas como hojas de transformación. A veces tanta potencia puede convertirse en un arma de doble filo.

Para ello podemos fijarnos en la estructura de cualquiera de las XSL que incluye Cocoon.

En esta XSL debemos declarar el espacio de nombres de las etiquetas que vayamos a tratar.

Conviene definir un xsl:template por cada etiqueta para que sea más fácil de leer para otros desarrolladores que la quieran usar. Y documentar al principio para que sirve y que parámetros se le pueden pasar y cómo.

Si nuestra etiqueta admite parámetros debemos pensar en el tipo de información que va a contener el parámetro. Lo más cómodo es recibir los parámetros en un atributo de la etiqueta, pero si por ejemplo va a ser un texto extenso o va a incluir caracteres tales como comillas conviene recibirlo en otra etiqueta interna. En el ejemplo del apartado anterior, la etiqueta `<util:time>` admite el formato de fecha especificado dentro de un atributo `format` o dentro de un elemento interno `<util:format>`.

Una vez hayamos concluido la xsl, tenemos que instalarla en Cocoon. Para ello debemos colocarla en algún lugar del classpath donde se esté ejecutando Cocoon y añadir otra línea al `cocoon.properties` de la siguiente forma:

```

processor.xsp.logicsheet.myNewPrefix.java =
resource:path_to_my_taglib/my_new_taglib.xsl

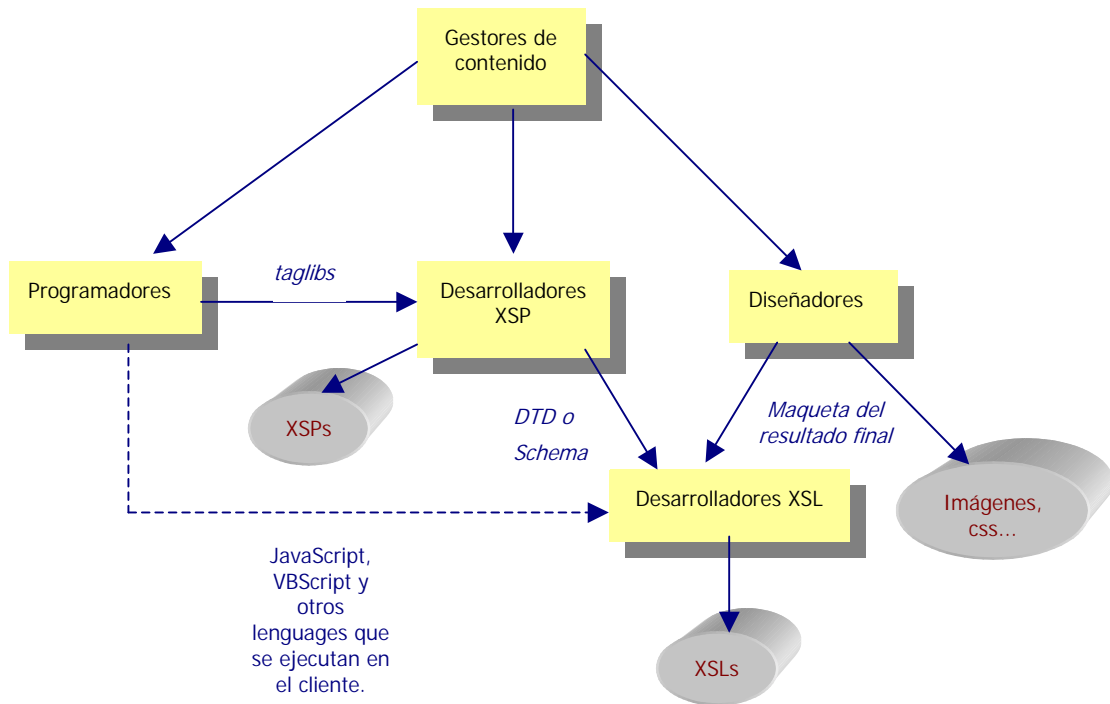
```

Reiniciamos Cocoon y ya está lista para ser usada.

Si realizamos algún cambio futuro en la XSL, posiblemente tengamos que eliminar los archivos .java generados a partir de las XSP que la usen para forzar una recompilación. En las versiones actual (v1.82) y anteriores no se detectan los cambios en las taglibs y debemos hacer este pequeño *"truco"* para usar los nuevos cambios.

WORKFLOW CON XSP

En el workflow ideal para el desarrollo de XSPs intervendrían 5 perfiles de desarrolladores:



- **Gestores de contenido:** Analizan los contenidos necesarios de cada página y se los notifican a los Programadores para que construyan taglibs que permitan la inclusión fácil en las páginas del contenido dinámico. La especificación de contenidos la deben recibir también los desarrolladores de XSP y los diseñadores.
- **Programadores:** Atienden las peticiones de los gestores de contenido sobre el contenido dinámico y les proporcionan taglibs que al ejecutarse generen dicho contenido.
- **Desarrolladores de XML (XSPs):** Estructuran los contenidos en el XML de la página. Introducen el contenido estático y colocan las etiquetas de las taglibs en los lugares donde corresponda para el contenido dinámico. También deberían elaborar algún documento que describa la estructura del XML que va a generar la XSP, ya sea una DTD o un schema.
- **Diseñadores:** Elaboran el boceto de la interfaz donde se presentará el contenido para cada vista, así como el resto de elementos estéticos tales como imágenes, hojas CSS...
- **Desarrolladores de XSL:** Escriben las XSLs que combinan el XML generado por las XSP, basándose en DTDs o XML Schemas, para obtener cada una de las vistas requeridas, basándose en las maquetas hechas por los diseñadores.

Como se puede apreciar el modelo de Cocoon permite un buen paralelismo de actividades mejorando los tiempos de desarrollo.

REFERENCIAS

☞ Sobre *Publishing Frameworks* y Cocoon:

“*Java and XML*”.

Capítulo 9: “Web Publishing Frameworks”

Autor: Brett McLaughlin.

Edición Junio 2000.

Editorial O’ Reilly.

☞ Para aprender más sobre Cocoon:

<http://xml.apache.org/cocoon/>

☞ Para aprender a usar XSPs:

<http://xml.apache.org/cocoon/xsp.html>

<http://xml.apache.org/cocoon/wd-xsp.html>

☞ Para obtener la última versión de Cocoon

<http://xml.apache.org/cocoon/dist/>

☞ Para aprender a instalar Cocoon:

<http://xml.apache.org/cocoon/install.html>

☞ Para aprender más sobre la caché de Cocoon:

<http://xml.apache.org/cocoon/caching.html>