Aplicaciones de Base de Datos con Cocoon 2

Introducción

Cocoon es un Java server framework que permite publicar contenido dinámico XML utilizando XSLT (XML Stylesheets Language Transformation). El contenido se describe en formato XML y XSLT se usa para transformar el contenido en múltiples formatos. Cocoon provee una plataforma para construir aplicaciones con una separación clara entre contenido, lógica y presentación.

Cocoon utiliza el concepto de una tubería para describir el proceso de publicación de contenido Web. Incluye una gran variedad de componentes reusables que pueden ser configurados para producir tecnicas complejas con un mínimo de desarrollo Java. Por ejemplo usando solo XML y XSLT, podemos usar Cocoon para:

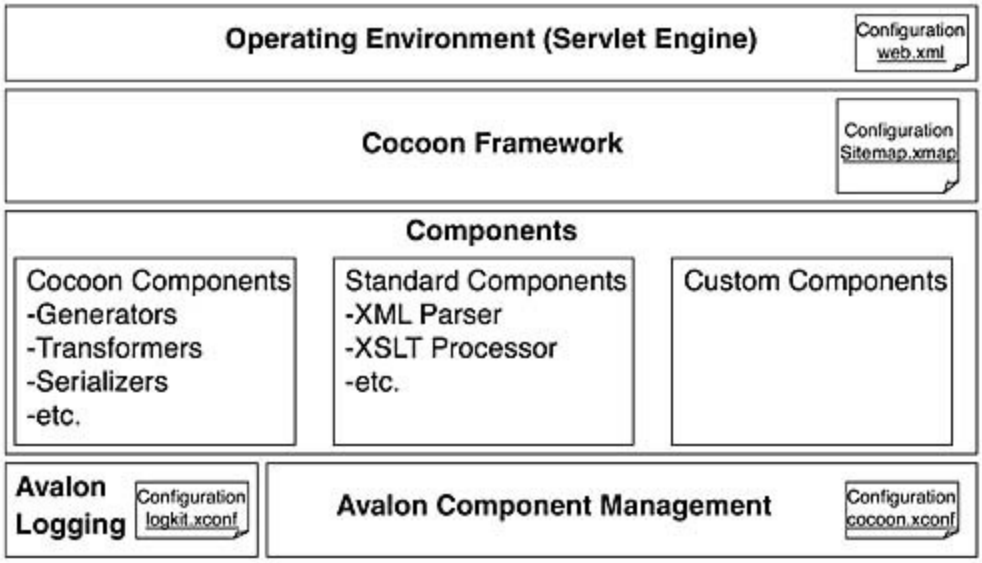


Ilustración 1Arquitectura de Cocoon 2

- Servir archivos estáticos como también repuestas generadas dinámicamente.
- Mapear solicitudes de los usuarios transparentemente en recursos físicos con una cantidad de procesamiento arbitrario.
- Ejecutar transformaciones XSLT simples y de múltiples etapas.
- Pasar parámetros dinámicamente a las transformaciones XSLT
- Generar una gran cantidad de formatos de salida: XML, HTML, PNG, JPEG, SVG y PDF.

Requerimientos

1. Java(TM) 2 SDK, Standard Edition Version 1.4.2_01 - http://java.sun.com/
2. Tomcat 4.1.27-LE-jdk14 - http://jakarta.apache.org/
3. Apache Cocoon 2.1 – http://cocoon.apache.org/

Instalación de Tomcat

1. Bajar la última versión en formato binario de Jakarta Tomcat 4.1.27-LE-jdk14
2. Descomprimir el archivo en un directorio adecuado
3. Crear la variable de ambiente CATALINA_HOME que apunte al directorio donde descomprimimos el archivo. Por ejemplo: CATALINA_HOME=/usr/local/jakarta-tomcat-4.1.9
4. Tomcat puede ser iniciado y parado por scripts ubicados en el directorio $CATALINA_HOME/bin. Por defecto Tomcat viene configurado para utilizar el puerto 8080. Si este puerto si ya está utilizando este puerto, entonces edite el archivo $CATALINA_HOME/conf/server.xml. Busque la cadena "*8080*" y cambiela por el valor que desee.
5. Inicie Tomcat con el script $CATALINA_HOME/bin/startup.sh
6. Compruebe la instalación apuntando el navegador hacia http://localhos:8080/. Una página de bienvenida se presenta si la instalación es correcta.

Instalación Rápida de Cocoon

1. Bajar la última versión binaria de Cocoon 2.1
7. Descomprimir el archivo en un directorio adecuado
8. Si Tomcat está corriendo, párelo ejecutando el script $CATALINA_HOME/bin/shutdown.sh
9. Copie el archivo cocoon.war en el directorio $CATALINA_HOME/webapps
10. Reinicie Tomcat ejecutando el comando $CATALINA_HOME/bin/startup.sh
11. Compruebe la instalación apuntando el navegador hacia http://localhost:8080/cocoon. Habrá una pausa mientras el servidor compila los archivos de configuración. Una página de bienvenida se presenta si la instalación es correcta.

Configurando Cocoon 2.1

Para la operación básica no hay mucho que configurar en Cocoon 2.1. Aparte del Sitemap solo hay dos archivos de configuración. Ambos se encuentran en $CATALINA_HOME/webapps/cocoon/WEB-INF:

- **log.xconf**: Configura la bitácora de Cocoon. Cocoon utiliza internamente Apache log4j para registrar entradas en la bitácora. Por defecto, Cocoon escribe los archivo log en $CATALINA_HOME/webapps/cocoon/WEB-INF/logs
- **cocoon.xconf**: Configura el cache, fuentes de datos y otras opciones avanzadas.

Para fines de desarrollo hay un cambio importante que hacer en cocoon.xconf. Buscar la siguiente entrada:

```
<sitemap file="sitemap.xmap" reload-method="asynchron" check-reload="yes" logger="sitemap"/>
```

Ahora cambie el valor del atributo reload-method a synchron. Esto altera la forma en que Cocoon responde a los cambios en el sitemap, el lugar central desde el que se configura las aplicaciones Cocoon. El valor

synchron significa que Cocoon reaccionará a los cambios inmediatamente, antes de servir cualquier solicitud. El valor implícito asynchron significa que leerá los cambios en background, así que los cambios no serán inmediatamente aplicados. Esto puede ser frustrante en el desarrollo cuando deseamos que los cambios se activen inmediatamente. Sin embargo no es lo mejor para una aplicación en producción que debe continuar sirviendo a los usuarios tan rápido como sea posible.

Si cambia el archivo *cocoon.xconf*, los cambios no se aplican automáticamente. Para aplicar los cambios se debe reiniciar Cocoon. Una forma de hacer esto es reiniciando Tomcat. La otra forma (recomendada) es solicitar directamente la raiz de cocoon con el parámetro cocoon-reload=true:
http://localhost:8080/cocoon?cocoon-reload=true

Esta opción se recomienda que se apague en el ambiente de producción. Para apagarlo se debe colocar en no. Se encuentra ubicado en el archivo web.xml:

```
<init-param>
        <param-name>allow-reload</param-name>
        <param-value>yes</param-value>
</init-param>
```

Configuración de LogKit

Cocoon utiliza las funciones de Avalon para la bitácora. Estas funciones son muy flexibles y poderosas. Es posible configurar los detalles sobre que se va a registrar en bitácora y que se debe hacer con los mensajes de log. Se tienen 5 niveles de log:

- DEBUG
- INFO
- WARNING
- ERROR
- FATAL_ERROR

Cada componente envía mensajes de log en uno de estos 5 niveles. LogKit decide que se hará con estos mensajes.

Usando el archivo de configuración, es posible elejir que solo ciertos niveles serán registrados a un archivo. En producción se recomienda solo registrar los mensajes con niveles ERROR o FATAL_ERROR. Cuando se está desarrollando se recomienda ver todos los niveles. Gracias al orden de los niveles, el nivel que se elija de la lista contiene a todos los niveles inferiores. Asi que si se elije DEBUG, se registrarán todos los mensajes. Si se elije WARNING, se registrarán los niveles: WARNING, ERROR y FATAL_ERROR.

Lo primero que se debe configurar es donde Cocoon encontrará el archivo de configuración de LogKit. Esto se hace mediante un parámetro del archivo web.xml:

```
<init-param>
<param-name>logkit-config</param-name>
<param-value>/WEB-INF/logkit.xconf</param-value>
</init-param>
```

La ubicación implícita de este archivo es /WEB-INF/logkit.xconf dentro del directorio de contexto de Cocoon. Este archivo es un documento XML que describe la configuración de LogKit:

```
<logkit>
	<factories>
		<factory type="priority-filter"
			class="org.apache.avalon.excalibur.logger.factory.PriorityFilterTargetFactory"/>
		<factory type="servlet"
			class="org.apache.avalon.excalibur.logger.factory.ServletTargetFactory"/>
		<factory type="cocoon"
			class="org.apache.cocoon.util.log.CocoonTargetFactory"/>
	</factories>

	<targets>
		<cocoon id="cocoon">
			<filename>${context-root}/WEB-INF/logs/cocoon.log</filename>
			<format type="cocoon">
				%7.7{priority} %{time} [%{category}] (%{uri})
				%{thread}/%{class:short}: %{message}\n%{throwable}
			</format>
			<append>true</append>
			<rotation type="revolving" init="1" max="4">
				<or>
					<size>100m</size>
					<time>01:00:00</time>
				</or>
			</rotation>
		</cocoon>
		<priority-filter id="filter" log-level="ERROR">
			<servlet>
				<format type="extended">
					%7.7{priority} %{time} %{message}\n%{throwable}
				</format>
			</servlet>
		</priority-filter>
```

```
</targets>

<categories>
        <category name="cocoon" log-level="DEBUG">
                <log-target id-ref="cocoon"/>
                <log-target id-ref="error"/>
        </category>
</categories>
```

Arquitectura de Cocoon 2

El modelo de Tubería

El concepto simple de una tubería es clave en la arquitectura de Cocoon 2. Una tubería consiste en datos de entrada seguido de un número de pasos de procesamiento que actúan sobre estos. Cada paso del procesamiento acepta como entrada la salida del paso anterior, hasta que se llega al final de la tubería y se produce la salida final.

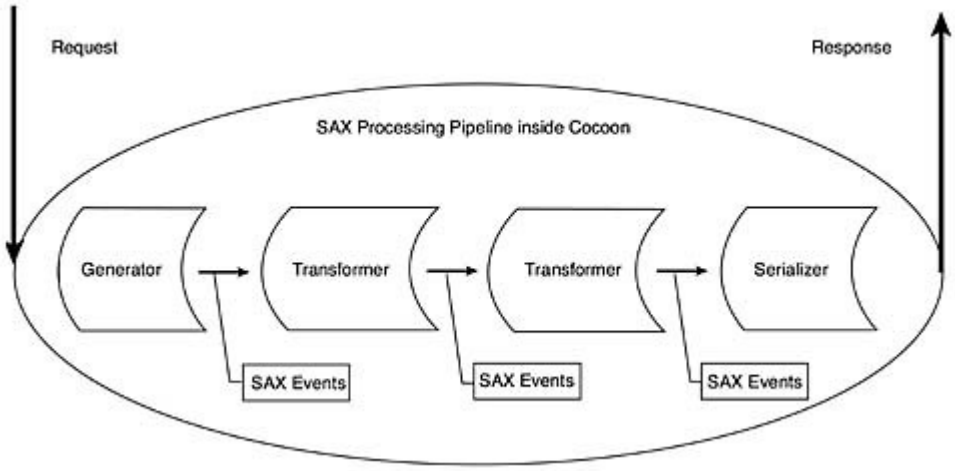


Ilustración 2Modelo de Tubería de Cocon 2

Las tuberías son simples en concepto. Es fácil descomponer cualquier tarea compleja en una pequeña serie de pasos que pueden ser organizados en una tubería. Este concepto es común en el mundo Linux donde los usuarios usan diferentes aplicaciones generales (por ejemplo: *find*, *grep* y *sort*) para que juntas cumplan una tarea especifica.

Esto además ilustra otro de los beneficios potenciales de las tuberías. Porque cada paso del proceso cumple una función especifica y esta amarrado con entradas y salidas fijas, es posible crear componentes de tuberías generales y reutilizables. La reutilización permite construir aplicaciones con poco esfuerzo de programación.

Componentes de una Tubería

Cocoon incluye un número de componentes de tubería generales que pueden ser conectados entre ellos en

formas útiles. Estos componentes pueden ser agrupados según su tipo, que depende del rol que juegan en una tubería.

- Entradas de la tubería generadores y lectores
- Pasos de Procesamiento transformadores y acciones
- Salidas de la Tubería Serializadores
- Procesamiento Condicional matcher y selectores

Una tubería generalmente consiste de por lo menos un generador y un serializador, pero también puede constar de cualquier número de Pasos de Procesamiento. Los datos son pasados dentro de una tubería como eventos SAX.

Entradas: generadores y lectores

Los **generadores** son responsables de leer una fuente de datos (por ejemplo un archivo) y pasar estos datos dentro de la tubería como una serie de eventos SAX. El generador mas simple es un analizador SAX. Generalmente cualquier fuente de datos que pueda ser mapeado en una serie de eventos SAX se puede convertir en la base de un generador.

Hay un número de generadores disponibles en Cocoon. Los más usados son:

- FileGenerator: Lee archivos XML del sistema de archivos o el Web
- HTMLGenerator: Lee archivos HTML del sistema de archivos o el Web
- DirectoryGenerator: Lee el sistema de archivos para proveer listados de directorios.

Los Lectores son un caso especial en el modelo de tubería de Cocoon porque no son componentes basados en XML. Los Lectores simplemente acceden a una fuente externa de datos y la copian directamente a la salida. Se utilizan para servir archivos estáticos tales como imágenes u hojas de estilo CSS. Los Lectores se pueden ver como una mini tubería, porque el mismo genera los datos de entrada y los serializa para la respuesta.

Procesamiento: transformadores y acciones

Tranformadores: son los pasos principales de procesamiento en una tubería Cocoon. Aceptan eventos SAX como entrada, ejecutan algún procesamiento útil y luego pasan los resultados al siguiente componente de la tubería como eventos SAX. Una forma útil de ver al transformador es como un componente que modifica un flujo de eventos SAX cuando este pasa a través de él. Desde este punto de vista son similares a los filtros SAX.

El transformador más usado es el XSLT. Este alimenta su entrada a un procesador XSLT que ejecuta una

transformación XSLT. Los resultados de la transformación son retroalimentados dentro de la tubería como eventos SAX.

Acciones: sirven para conectar funcionalidad dinámica adicional dentro de una tubería y con frecuencia se construyen personalizados para aplicaciones particulares. A pesar de esto algunas acciones genéricas forman parte de Cocoon. Por ejemplo: para manejar la interacción con bases de datos, validación de formas, envío de correo electrónico, etc. La terminación exitosa de una acción puede también determinar si se ejecutarán los siguientes pasos del procesamiento o no.

Salidas: Serializadores

Serializadores son los puntos de salida de las tuberías de Cocoon. Son responsables de tomar un flujo de eventos SAX producido directamente desde un generador (este es el caso de la tubería mas corta posible) o de un paso de procesamiento anterior (por ejemplo un transformador) y presentarlo en el formato requerido para la respuesta. El formato especifico depende del serializador exacto que usemos.

El serializador mas simple es el serizalizador XML, el cual tan solo convierte los eventos SAX de nuevo en documentos XML. Otros serializadores pueden producir HTML, WML, texto plano TXT, PDF o imágenes JPEG, PNG. Todos los serializadores esperan un flujo de eventos SAX que cumpla con un vocabulario XML especifico:

- HTML Serializer: Cambia XHTML a HTML
- SVG Serializer: Cambia SVG a imágenes JPEG o PNG
- PDF Serializer: Cambia XSL-FO a PDF
- HSSF Serializer: Cambia GMR a XLS

La habilidad de tomar contenido XML, procesarlo y servirlo en múltiples formatos es el verdadero poder de Cocoon.

Condicionales: Matchers y Selectores

Cualquier tubería no trivial contendrá algunas secciones condicionales. Por ejemplo, los pasos exactos de procesamiento pueden depender de factores como los parámetros de la solicitud, el navegador del usuario u otros.

Matchers: es el mas sencillo de los dos componentes condicionales y es el equivalente a la expresión if. Si una condición es verdadera, entonces una tubería particular o una sección de una tubería es evaluada.

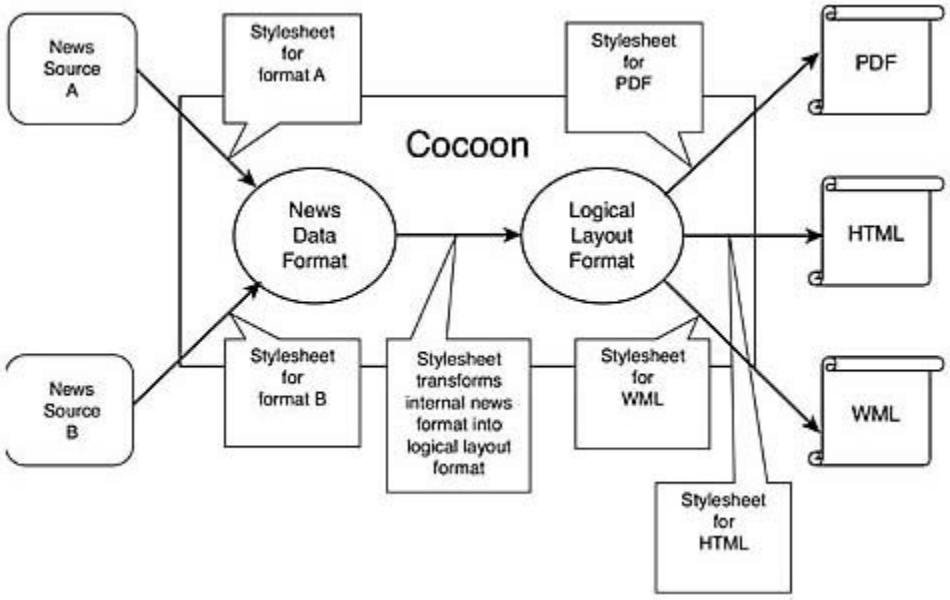
El segundo tipo es el **selector** el cual es similar a una expresión if-then-else. Los selectores se usan cuando una de las diferentes opciones está disponible. Se usan normalmente para crear secciones condicionales

dentro de una tubería. Mientras que los matchers son usados para determinar si se entra a una tubería en particular.

Hay diferentes implementaciones para cada uno de estos componentes. Ellos siguen un comportamiento común de comprobar algunos aspectos de la solicitud (como el Hostname, navegador, parámetros o URL) o la sesión de usuario. Los matcher pueden crearse usando comodines o expresiones regulares. Los selectores normalmente enumeran todos los valores posibles.

Haciendo trabajar las Tuberías

Después de ver los componentes normalmente usados para construir tuberías Cocoon 2, es importante colocar las tuberías en el contexto para ver como ellas se ponen a trabajar. A continuación presentamos una descripción del ciclo lógico para recibir solicitudes y servir respuestas:



1. Aceptar la solicitud del usuario
2. Determinar la tubería que se debe usar para interpretar esta solicitud y producir una respuesta (usando un matcher).
3. Construir la tubería de los componentes disponibles y preconfigurados.
4. Instruir a la tubería que sirva la solicitud.
5. Retornar la respuesta generada al usuario, posiblemente cache el resultado para uso posterior.

Este es el ciclo básico de solicitud-respuesta que usa Cocoon 2 para publicar en el Web datos XML. Para administrar este ciclo Cocoon 2 proporciona un archivo de configuración llamado sitemap.

Herramientas de desarrollo

1. Pollo – Editor XML para Sitemap - http://sourceforge.net/projects/pollo
2. SQuirreL SQL Client – http://sourceforge.net/projects/squirrel-sql/

3. jEdit – Editor XML - http://www.jedit.org/

Desarrollo

Directorios de la Aplicación

1. docs – Documentos XML de la aplicación
2. stylesheets – Hojas de estilo XSL
3. dtd – Plantillas DTD
4. resources – recursos de la aplicación
5. sitemap.xmap – Documento que contiene las reglas del espacio URL

Sitemap

Es el centro principal de configuración de Cocoon2. El Sitemap es un documento XML y por este motivo tiene una estructura bien definida. Su propósito es:

1. Declarar los componentes antes de ser usados en tuberías
2. Definir las tuberías usando los componentes declarados (definir la forma en que las páginas Web serán generadas).

Un sitemap debe contener el namespace sitemap http://apache.org/cocoon/sitemap/1.0 que sirve para identificar los elementos del sitemap. También debemos tomar en cuenta que el sitemap está dividido en dos secciones principales map:components y map:pipelines que reflejan las dos principales responsabilidades del sitemap:

```
<map:sitemap
        xmlns:map="http://apache.org/cocoon/sitemap/1.0">
        <map:components>
                <!-- component declarations -->
                <map:generators/>
                <map:readers/>
                <map:transformers/>
                <map:actions/>
                <map:serializers/>
                <map:actions/>
                <map:matchers/>
```

```
        <map:selectors/>
    </map:components>
    <map:pipelines>
        <!-- pipeline definitions -->
    </map:pipelines>
</map:sitemap>
```

Las declaraciones de cada tipo de componente se agrupan juntas dentro del elemento especifico. Por ejemplo todas las declaraciones de los generadores se encuentran dentro del elemento map:generators.

Declarando componentes en el Sitemap

Para declarar componentes debe notar los siguientes cosas:

- *component-type*: Es el nombre del tipo especifico de componente. Por ejemplo: los elementos Generadores, contienen declaraciones de generadores.
- Cada componente tiene un atributo único *name*. El nombre se usa para referirse al componente en cualquier otro lado del sitemap.
- Cada componente debe identificar su *implementation*. Es posible tener mas de una instancia del componente compartiendo la misma implementación, pero declarada con nombres diferentes.
- Un componente *default* puede ser identificado. Se usa cuando una instancia de componente no se nombra de forma especifica.
- Los componentes pueden recibir parámetros del sitemap. Es posible tener instancias del mismo componente, pero con diferentes parámetros. Los elementos parámetros son específicos a cada componente.

```
<map:component-types
        default="component-name">

  <map:component-type
        name="component-name"
        src="implementation">

        <!-- component-type
        specific parameters -->

        </map:component-type>
</map:component-types>
```

Para ejemplo concretos de la declaración de componentes, lea la primera sección de sitemap implícito que se encuentra en $CATALINA_HOME/webapps/cocoon/sitemap.xmap.

Extensibilidad

Cocoon carga los componentes según las declaraciones en el Sitemap utilizando la capacidad de carga dinámica de clases. Para ser cargados y conectados en una tubería, cada componente debe implementar un interfaz Java específico dependiendo de su tipo. Por ejemplo, todos los generadores deben implementar el interfaz org.apache.cocoon.generation.Generator

Cocoon se apoya en el interfaz para describir cada tipo de componente, esto significa que las capacidades de este pueden ser fácilmente extendidas tan solo escribiendo nuevas implementaciones de estos interfaces y agregando la declaración apropiada en el sitemap.

Por ejemplo, supongamos que tenemos un montón de datos en formato CVS que deben estar disponibles en formatos HTML y CVS. Una forma de resolver esto es escribiendo una aplicación que lee CVS y lo escribe en XML. Esto agrega trabajo adicional en la administración ya que necesitaremos procesar los nuevos datos a medida que vayan llegando y almacenarlos en formato XML y CVS.

Ambas integraciones pueden ser logradas en Cocoon escribiendo un generador personalizado como com.mycompany.CSVGenerator que es capaz de parsear archivos CVS para generar directamente eventos SAX. Esta clase puede ser conectada a Cocoon declarandola en el sitemap:

```
<map:generator name="csv" src="com.mycompany.CSVGenerator"/>
```

Este componente ahora se puede utilizar para alimentar datos CVS a una tubería de Cocoon, donde puede ser transformada, manipulada y serializada en múltiples formatos. El archivo CVS original siempre estará disponible usando un componente Lector. Porque solo habrá un solo archivo origen y todas las transformaciones dinámicas las hace Cocoon hay menos trabajo administrativo. Obviamente este ejemplo puede ser aplicado a cualquier otro tipo de formato y extenderlo para para incluir la entrega del documento desde el sistema de administración de contenido, etc.

Configuración del Sitemap

Aquí se presentan algunos ejemplos de configuraciones de sitemaps que muestran como los componentes pueden ser combinados para ejecutar trabajo útil. Se centran en la configuración del Sitemap en lugar de detallar las transformaciones individuales y los formatos XML.

Este ejemplo asume que la siguiente estructura de directorios esta creada en $CATALINA_HOME/webapps/cocoon (a partir de ahora lo llamaremos $COCOON_HOME):

```
/static      Documentos Estaticos HTML
/content      Contenido XML
/styles       Hojas de estilo CSS
```

/transforms Hojas de Estilo XSLT

Truco: Se recomienda experimentar con Cocoon, en un ambiente limpio. Creamos un nuevo directorio bajo $CATALINA_HOME/webapps/, por ejemplo: cocoon-dev. Luego copie $COCOON_HOME/cocoon.xconf y el directorio $COCOON_HOME/WEB-INF que es el que contiene todas las clases de Cocoon. Esto creará una nueva aplicación Cocoon que puede ser encontrada en http://localhost:8080/cocoon-des/.

Luego creamos un nuevo sitemap.xmap en este nuevo directorio. Así los experimentos se harán sin dañar los ejemplos o la documentación original.

Declaración de Componentes

Los ejemplos asumen que hemos hecho las siguientes declaraciones en el sitemap:

```
<map:generators default="file">
        <map:generator name="file" src="org.apache.cocoon.generation.FileGenerator"/>
</map:generators>

<map:transformers default="xslt">
        <map:transformer  name="xslt" src="org.apache.cocoon.transformation.TraxTransformer"/>
</map:transformers>

<map:readers default="resource">
        <map:reader name="resource" src="org.apache.cocoon.reading.ResourceReader"/>
</map:readers>

<map:serializers default="html">
        <map:serializer name="xml" mime-type="text/xml" src="org.apache.cocoon.serialization.XMLSerializer"/>
        <map:serializer name="html" mime-type="text/html" src="org.apache.cocoon.serialization.HTMLSerializer"/>
        <map:serializer name="svg2png" src="org.apache.cocoon.serialization.SVGSerializer" mime-type="image/png"/>
        <map:serializer name="fo2pdf" src="org.apache.cocoon.serialization.FOPSerializer" mime-type="application/pdf"/>
</map:serializers>

<map:matchers default="wildcard">
        <map:matcher name="wildcard" src="org.apache.cocoon.matching.WildcardURIMatcher"/>
</map:matchers>
```

Sirviendo Documentos Estáticos

Este primer ejemplo muestra como se pueden servir documentos estáticos con Cocoon. La tubería a contnuación muestra como se puede hacer esto utilizando un Lector. Obviamente es preferible dejar que un Web Server haga este trabajo, pero es un buen ejemplo para mostrar el trabajo con las tuberías:

```
<map:pipelines>
```

```
        <map:pipeline>
                <map:match pattern="index.html">
                        <map:read src="static/index.html" mime-type="text/html"/>
                </map:match>
        </map:pipeline>
<map:pipelines>
```

En primer lugar se define la tubería con un hijo <map:pipeline> del elemento <map:pipelines>. Este último elemento debe contener todas las definiciones de tuberías.

El componente matcher se usa para para asociar esta tubería con una solicitud usando *match pattern*. En esta instancia una solicitud del documento "*index.html*" (por ejemplo: http://locahost:8080/cocoon/index,html) disparará esta tubería.

El proceso que debe suceder cuando se dispare esta tubería se define después. En este caso se le instruye a un Lector enviar al usuario el archivo $COCOON_HOME/static/index.html con el tipo mime text/html. La ubicación del archivo es completamente independiente del URL que solicitó en usuario.

Es obvio que asociar explícitamente cada archivo individual a una tubería requiere de un gran trabajo. Por suerte el componente matcher puede trabajar de otra forma para evitar esto.

Usar comodines

El ejemplo anterior lo cambiaremos para demostrar el uso de comodines para match fragmentos de la URL solicitada. Se ha ampliado un poco la tubería para permitir la entrega de hojas de estilo CSS y documentos HTML. Para hacer esto solo se debe agregar un nuevo matcher a la tubería actual. Esto tiene el mismo efecto que crear una tubería nueva, porque Cocoon verificará los dos patterns.

```
 <map:pipeline>
        <map:match pattern="*.css">
                <map:read src="styles/{1}.css" mime-type="text/css"/>
        </map:match>
        <map:match pattern="**.html">
                <map:read src="static/{1}.html" mime-type="text/html"/>
        </map:match>
</map:pipeline>
```

El matcher de comodines permite dos tipos de comodines. Ambos se muestran en el ejemplo. El primero con un solo asterisco match cualquier número de caracteres excepto la barra (/). Y el doble asterisco match cualquier número de caracteres incluyendo la barra (/). El texto matched con estos patterns está disponible a los otros componentes del sitemap y pueden ser referidos como {1}, {2}, {3}, etc.

Dependiendo de cuantos comodines se usen.

En el ejemplo de arriba una solicitud http://localhost:8080/cocoon/misitio.css match con el pattern CSS y el valor "*misitio*" se asignará a {1}. El lector enviará de vuelta $COCOON_HOME/styles/misitio.css con el tipo mime correcto. Una solicitud http://localhost:8080/style/misitio.css no match no el pattern porque este sólo tiene un asterisco.

El nuevo match pattern HTML usa doble asterisco como comodín. Así que una solictud a http://localhost:8080/help/help.html será exitosa y {1} será igual a *help/help*. El lector enviará de vuelta al usuario el archivo $COCOON_HOME/static/help/help.html.

Es importante saber que Cocoon procesa los match patterns secuencialmente en el orden que se definen en el sitemap. Cocoon procesa la solicitud acorde a la primera plantilla que exitosamente match. Por eso el orden es importante y se debe procurar colocar primero los match más específico. Por ejemplo unmatch para *index.html* debe ser declarado antes de **.html*. O sino nunca será alcanzado.

Ejecutando una transformación

Al menos se necesitan tres componentes para ejecutar una transacción: un generador para leer el documento XML, un transformador para realizar la transformación y un serializador para entregar los resultados. Aquí se explica como se colocan las piezas juntas para efectuar una transformación.

Primero se declara una tubería y definimos una plantilla match para dispararla:

```
<map:pipe>
        <map:match pattern="content/*.html">
```

Luego, agregamos un generador para leer el documento XML desde el directorio "*content*":

```
                <map:generate  src="content/{1}.xml"/>
```

Luego se agrega un transformador para transformar el documento XML usando una hoja de estilo específica:

```
                <map:transform  src="transforms/content2html.xsl"/>
```

Al final, se usa un serializador para cambiar los resultados de la transformación en un documento HTML que se devuelve al usuario:

```
                <map:serialize type="html"/>
        </map:match>
```

```
</map:pipe>
```

Solicitando el URL http://localhost:8080/content/document.html se dispara esta tubería y causa que Cocoon primero analize document.xml, luego lo transforme utilizando $COCOON_HOME/transforms/content2html.xsl antes de enviar los resultados de regreso al navegador.

Se pueden crear transformaciones mas complejas agregando más transformadores a la tubería. Una vez mas, se usan comodines para evitar tener que definir las entradas actuales a la tubería mediante el uso del proceso de match.

Generando otros formatos

Ahora que se ha visto como generar dinámicamente HTML a partir de contenido XML, es importante aprender como entregar otros formatos de documentos. Cocoon permite esto mediante el uso de serializadores personalizados.

Cuando entregamos XML en lugar de HTML como resultado de una transformación, se debe utilizar el serializador XML. Si está disponible una transformación para generar archivos RSS, entonces se debe escribir una tubería que incluye el fragmento siguiente:

```
<map:transform src="transforms/content2rss.xsl"/>
<map:serialize type="xml"/>
```

Note que un serializador especifico ha sido elejido con el atributo *type* en el componente serializador. El valor del atributo match el nombre de uno de los serializadores del panel de declaraciones de componentes descrito anteriormente.

SVG (Scalable Vector Graphics) es un formato XML para describir diagramas vectoriales. Normalmente se necesita de una extensión al navegador para ver documentos SVG. Sin embargo, Cocoon incluye un serializador capaz de crear imágenes JPEG o PNG directamente a partir de un documento SVG. Este serializador puede ser llamado de la siguiente forma:

```
<map:transform src="transforms/content2svg.xsl"/>
<map:serialize type="svg2png"/>
```

Cocoon también permite la creación de archivos PDF directamente a partir de documentos XSL-FO. De nuevo, solo se debe asegurar que la hoja de estilo produzca el formato correcto de entrada en el serializador especial:

```
<map:transform src="transforms/content2fo.xsl"/>
<map:serialize type="fo2pdf"/>
```

Pasando parámetros

Con frecuencia es muy útil pasar parámetros a las transformaciones XSLT. Cocoon permite pasar parámetros desde dentro del sitemap. La primera forma de lograr esto es mediante el elemento <map:parameter>. Por ejemplo:

```
<map:transform src="transforms/content2html.xsl">
        <map:parameter name="myFixedParam" value="fixed-value"/>
        <map:parameter name="myDynamicParam" value="{1}"/>
</map:transform>
```

Tanto el nombre como el valor del parámetro se definen como atributos del elemento <map:parameter>. Es posible pasar parámetros fijos y dinámicos a las hojas de estilo con este método. El elemento de parámetro tendrá su valor igual según el primer comodin del match pattern. La hoja de estilo debe incluir un elemento <map:param> para recibir correctamente el parámetro en la transformación.

La forma alternativa de enviar parámetros a una hoja de estilo permite pasar todos los parámetros del URL solicitado. Por ejemplo si una solicitud http://localhost:8080/contenido.html?param1=valor1&param2=valor2 causa que se dispare esta tubería, entonces los dos parámetros param1 y param2 serán pasados a la hoja de estilo.

```
<map:transform src="transforms/content2html.xsl">
  <map:parameter name="use-request-parameters" value="true"/>
</map:transform>
```

Este método es útil cuando hay un número variable de parámetros que deben ser pasado en la solicitud. Sin embargo este método tiene un costo adicional en el rendimiento, porque Cocoon es menos hábil de cachear los resultados de transformaciones que usan este método que aquellos que usan parámetros fijos. Si cualquiera de los parámetros del URL se cambia, incluso si no directamente es usado por la hoja de estilo, los resultados cacheados no se usarán.

Sitemap Principal

No se recomienda modificarlo mucho. Está ubicado en $CATALINA_HOME/webapps/cocoon/sitemap.xmap.

Subsitemap

Sirve para definir la configuración exclusiva que se aplica a la aplicación que estamos creando. Debemos crear nuestro propio sitemap.xmap que debe contener únicamente la configuración que se aplica a nuestra aplicación. Luego debemos “montar” nuestro subsitemap al “Sitemap Principal”. Se logra agregando un elemento de tubería que contenga una etiqueta <map:mount>. Las tuberías normalmente contienen información acerca de como se procesan los URL para crear los documentos esperados (por ejemplo:

HTML). La tubería que debemos colocar en el Main sitemap debe ser similar a esta:

```
<map:pipeline>
        <map:match pattern="xbank">
                <map:redirect-to uri="xbank/login"/>
        </map:match>
        <map:match pattern="xbank/**">
                <map:mount uri-prefix="xbank/" src="projects/xbank/" check-reload="yes"/> </map:match>
</map:pipeline>
```

El 1° elemento <map:match> es la regla para resolver el URL: *http:///{servidor}/cocoon/xbank*. Su trabajo es reenviar el procesamiento hacia *http:///{servidor}/cocoon/xbank/login* en este misma tubería.

El 2° elemento <map:match> dice que todos los URL que inicien con *xbank* son procesados por reglas en otro sitemap – **subsitemap**.

El atributo **src** define el directorio donde se encuentra el archivo **sitemap.xmap**. Este archivo define las reglas de nuestro subsitemap. El atributo **src** siempre debe apuntar a la raíz de nuestra aplicación. En nuestro caso *{webappdir}/proyectos/agbar*.

Subsitemap de la aplicación

Las instrucciones de cómo se debe procesar */login* se encuentran en el subsitemap de nuestra aplicación. Por ejemplo:

```
<map:pipeline>
        <map:match pattern="login">
                <map:generate type="file" src="docs/login.xml"/>
                <map:transform type="xslt" src="stylesheets/simple-page2html.xsl"/>
                <map:serialize type="html"/>
        </map:match>
<map:pipeline>
```

En este caso el documento */login* se **genera** a partir del archivo *docs/login.xml*, luego el árbol XML resultante es **transformado** por xslt a otro árbol XML utilizando la hoja de estilo *stylesheets/simple-page2html.xsl* (a partir de este momento contiene etiquetas HTML válidas, pero todavía es un XML). Al final el XML es **serializado** como un documento HTML normal.

Normalmente no se debe especificar el atributo **type** para los componentes implícitos (generadores, transformadores y serializadores, etc.) En el ejemplo anterior podríamos obviar el atributo **type** en todos los componentes porque "**file**" es el generador implícito, "**xslt**" es el transfomador implícito y "**html**" es el serializador implícito.

Componentes Implícitos

Es posible cambiar los componentes implícitos solo para nuestro subsitemap definiendo algo como esto:

```
<map:components>
        <map:generators default="file"/>
        <map:transformers default="xslt"/>
        <map:readers default="resource"/>
        <map:serializers default="html"/>
        <map:selectors default="browser"/>
        <map:matchers default="wildcard"/>
</map:components>
```

El significado de esto es que el conjunto de generadores (definido en Main Sitemap) no cambia, y el nuevo generador implícito es de tipo "**file**". Para ver la lista completa de todos los componentes en la distribución Cocoon2, ver el elemento <map:components> en el Main Sitemap.

Conexión a Bases de Datos

Crear la conexión a la base de datos cuando se necesita en una aplicación es ineficiente. Esto se debe a los altos costos de configuración de la misma y el cierre posterior de la conexión. Al igual que otros recursos de procesamiento costosos, Las conexiones deben ser administradas usando un pool. Las conexiones pueden ser tomadas y regresadas al pool por los componentes individuales dejando que la aplicación maneje el pool. Los pool son creados típicamente cuando la aplicación se inicia y pueden aumentar o reducir su tamaño dependiendo de su uso.

Cocoon2 proporciona un **pool de conexiones** para bases de datos. Es un repositorio central para mantener las conexiones JDBC para todas las aplicaciones servidas por Cocoon. Después de configurar los parámetros en el archivo de configuración de Cocoon (*cocoon.xconf*) es posible acceder a los datos tan solo especificando el nombre del pool de conexiones. Después es posible usar etiquetas ESQL (definidas en la hoja lógica ESQL) en nuestras páginas XSP para poder encapsular todo el código Java en etiquetas XML.

Antes de poder utilizar el pool de conexiones se deben configurar algunos parámetros en Tomcat y Cocoon 2:

Carga del Controlador JDBC

En primer lugar es necesario asegurarse que el controlador JDBC este en el CLASSPATH de Java. A diferencia del controlador JDBC jdbc:odbc: los demás controladores JDBC no son parte de la distribución J2SDK. Por ejemplo postgresql.org distribuye el controlador JDBC para postgreSQL como *pgjdbc2.jar*. Este paquete contiene el controlador JDBC (org.postgresql.Driver) que se necesita cargar explícitamente en el ambiente Java.

Es posible configurar el CLASSPATH de diferentes maneras. En general basta con editar el script de inicio del servidor de aplicaciones para que al inicio la variable de ambiente CLASSPATH incluya en archivo requisito. Como una alternativa se puede copiar el archivo al directorio $COCOON_HOME/WEB-INF/lib. Cocoon automáticamente agrega cualquier zip o jar que encuentre en este directorio a su CLASSPATH.

El controlador debe ser cargado directamente a memoria antes que JDBC lo pueda usar. Este trabajo lo debe realizar Tomcat.

Configuración de Tomcat

Tomcat utiliza el descriptor de instalación web.xml ubicado en el directorio WEB-INF de la aplicación Cocoon2. Todo lo que se debe hacer es modificar el parámetro de inicialización *load-class* <init-param>

```
<init-param>
        <param-name>load-class</param-name>
        <param-value>
                <!-- For IBM WebSphere: -->
                        com.ibm.servlet.classloader.Handler
                <!-- For JDBC-ODBC Bridge: -->
                        sun.jdbc.odbc.JdbcOdbcDriver
                <!-- For Interbase DBMS: -->
                        interbase.interclient.Driver
                <!-- For PostgrSQL -->
                        org.postgresql.Driver
         </param-value>
</init-param>
```

Ahora el controlador JDBC de la base de datos será cargado cada vez que se inicie Tomcat. No se debe olvidar de configurar el CLASSPATH para que incluya el camino a todos los controladores JDBC que la aplicación usa.

Configuración en Cocoon2

Cocoon2 usa el archivo XML *cocoon.xconf* para guardar parámetros en tiempo de ejecución. Este archivo esta ubicado en $COCOON_HOME*/WEB-INF*. Este archivo contiene parámetros de los componentes internos tales como cache, hojas de estilo, sitemap y fuentes de datos.

Para configurar una fuente de datos y usarla en un pool de conexiones es necesario agregar los parámetros de su base de datos al elemento <datasources> en *cocoon.xconf*. Por ejemplo:

```
 <jdbc name="hb_pool">
        <pool-controller min="5" max="10"/>
        <dburl>jdbc:interbase://hbser/d:\ibbase\hbsbase.gdb</dburl>
        <user>webapp</user>
        <password>heslo</password>
</jdbc>
```

En este ejemplo tenemos un pool de conexiones llamado hb_pool con conexión JDBC a un servidor Interbase en la computadora hbser conectándose a la base de datos d:\ibbase\hbsbase.gdb. El nombre de usuario y contraseña normalmente deben ser incluidos. Si es necesario asegurarse que se ha incluido la cadena de conexión JDBC sin errores, debe utilizar una herramienta como SQuirreL SQL Client.

Es posible crear pools adicionales el único requisito es que el nombre debe ser único.
El elemento pool-controler controla las opciones del pool. Cuando Cocoon 2 inicia, automáticamente crea la cantidad de conexiones definida en el atributo min. El atributo max, define la cantidad máxima de conexiones.
El elemento auto-commit es opcional e indica si las conexiones deben hacer un commit automático. Su valor implícito es true. Si se configura a false, entonces la aplicación deberá hacer sus propios commit. El valor recomendado para esta opción es true a menos que se desee tener el control completo de las transacciones.

Al finalizar la configuración es necesario reiniciar Cocoon 2 para que vuelva a leer el archivo de configuración. A diferencia del sitemap, Cocoon solo lee el archivo de configuración cuando arranca. Solo reinicie Tomcat para que los cambios tengan efecto.

Hoja Lógica ESQL

Las hojas lógicas son un mecanismo para crear etiquetas personalizadas que se usan en las páginas XSP. Se implementan usando transformaciones XSLT. Un generador de código usa las transformaciones XSLT para reemplazar las etiquetas con código Java que implementa la funcionalidad deseada.

La hojas lógicas ESQL es una capa delgada sobre el estándar JDBC API, definiendo un número de etiquetas que corresponden a una particular funcionalidad JDBC. Esta hoja lógica es una forma sencilla de

generar código JDBC para una aplicación.

Elementos base ESQL

```
<xsp:page
        language="java"
        xmlns:xsp="http://apache.org/xsp"
        xmlns:esql="http://apache.org/cocoon/SQL/v2">
<root>
        <esql:connection>
                <esql:execute-query>
                        <!-- información de conexión -->
                        <esql:pool/>

                        <!-- Consulta SQL -->
                        <esql:query/>

                        <!-- elementos resultantes del procesamiento -->
                        <esql:results/>
                        <esql:update-results/>
                        <esql:no-results/>
                        <esql:error-results/>
                </esql:execute-query>
        </esql:connection>
</root>
</xsp:page>
```

Se usan un número común de elementos estructurales cuando se trabaja con la hoja lógica ESQL. Cada uno de estos elementos deriva su funcionalidad a partir de un objeto equivalente en el JDBC API. La presencia de estos elementos es usada para manejar la generación de secciones especificas de código JDBC en su forma compilada de una página XSP.

El esql:connection es el equivalente del Connection de JDBC. Al igual que todas las operaciones JDBC se derivan de una conexión de base de datos en particular, cada uno de los elementos ESQL debe estar correctamente anidado dentro de un elemento esql:connection. Es posible tener múltiples elementos de conexión dentro de una sola página XSP, lo que le permite a una sola página interactuar con diferentes fuentes de datos. El elemento de conexión debe también contener otros elementos que definen como se creará la conexión a la base de datos (por ejemplo: esqp:pool).

El elemento esql:execute-query es el equivalente de objeto JDBC PreparedStatement. Define como ejecutar cada una de las consultas dentro de una conexión a base de datos dada como tambien como los resultados de esas consultas se procesarán. Es aceptable anidar elementos individuales esql:execute-query para crear consultas anidadas de bases de datos.

El elemento esql:query contiene la consulta SQL (SELECT, INSERT, UPDATE, DELETE) que se ejecutará usando esta conexión.
El procesamiento de los resultados de la consulta depende del tipo de consulta que se lleva a cabo. El procesamiento de un SELECT es definido dentro del elemento esql:results, mientras que el procesamiento de expresiones INSERT, UPDATE y DELETE son procesadas dentro de un elemento esql:update-results. Sea cuidadoso de usar el elemento correcto; Cocoon no generará un error si se usa el elemento incorrecto.

Los elementos esql:results y esql:update-results usualmente contienen otras etiquetas ESQL que proveen un acceso mas fino a los resultados de la consulta. Estas etiquetas tienen su equivalente en los objetos ResultSet y ResultSetMetaData.

Los elementos esql:no-results y esql:error-results contienen etiquetas que describen el procesamiento que se debe ejecutar en caso que la consulta no tenga resultados o halla generado un SQLException, respectivamente.

En cualquier caso, se pueden mezclar etiquetas ESQL con etiquetas definidas por el usuario que son la salida directa de una página XSP. También se pueden usar etiquetas XSP y otras de otras hojas lógicas, para definir procesamiento adicional como se requiera. La única restricción real es asegurarse que la página XSP permanezca bien formada.

Definiendo la conexión

Es posible definir la conexión del elemento esql:connection de dos formas.

La primera es refiriéndose a un pool de conexiones. Esto se hace utilizando el elemento esql:pool:

```
<esql:connection>
        <esql:execute-query>
                <esql:pool>myPoolName</esql:pool>
                  ...
        </esql:execute-query>
</esql:connection>
```

El contenido del elemento debe corresponder al nombre del origen de datos JDBC definido en cocoon.xconf.

La segunda forma es definiendo los parámetros de conexión directamente en el elemento esql:connection. Esto se hace usando los siguientes elementos:

- esql:dburl –- la cadena JDBC de conexión

- esql:usename -– el nombre de usuario para conectarse a la base de datos
- esql:password -– la contraseña del nombre de usuario
- esql:driver -– el controlador JDBC
- esql:autocommit -– indica si la conexión JDBC debe hacer commit automáticamente

El beneficio de esta última forma es que los detalles de la conexión puede ser generada dinámicamente – por ejemplo solicitando el nombre del usuario y la contraseña de la sesión actual. Mientras que esto ofrece un gran nivel de flexibilidad, elimina los beneficios de rendimiento de dejar que Cocoon administre la conexión usando un pool de base de datos.
A menos que sea necesario hacer conexiones dinámicamente, se recomienda que las conexiones se hagan usando el elemento esql:pool.

Procesamiento de consultas sencillas

Tomando la estructura básica descrita anteriormente. Es posible crear una página XSP que consultará la tabla Categorías de la base de datos de ejemplo para generar una lista sencilla y ordenada de las categorías de música. Las consultas SQL se definen con el elemento esql:query:

```
<esql:query>
        select cat_id, name from category
                order by cat_id
</esql:query>
```

La definición de cómo procesar cada línea del resultado de esta consulta se logra usando el elemento esql:row-results, que es un hijo de esql:results. El contenido de este elemento se traslada al código que es ejecutado para cada línea del JDBC ResultSet que es generado por la consulta. Por ejemplo, este elemento describe procesamiento a nivel de líneas:

```
<esql:results>
        <esql:row-results>
                <category>
                        <esql:get-columns/>
                </category>
        </esql:row-results>
</esql:results>
```

Este ejemplo declara que el elemento category es generado para cada línea en los resultados e indica que la hoja lógica debe automáticamente mapear los datos de la línea en elementos XML. Cada columna se convierte en un elemento XML diferente cuyo contenido es el valor de esa columna de la línea actual. Así al ejecutar esta página XSP (por ejemplo: http://localhost:8080/bdc2/categories.xml) se produce:

```
<categories>
        <category>
                <CAT_ID>1</CAT_ID><NAME>Blues</NAME>
        </category>
        ...masa resultados...
</categories>
```

Note que los nombres de los elementos son una versión en mayúsculas de los nombres de las columnas definidos en la consulta. Aunque esto proporciona una forma rápida de producir XML desde una consulta, los nombres resultantes pueden ser no deseados, especialmente si la convención de nombres de la base de datos hace difícil de leer el XML generado. Es posible crear SQL alias para cambiar los nombres de los elementos individuales (por ejemplo: SELECT cat_id as id, name FROM ....). Si prefiere una versión en minúsculas, agregue el atributo tag-case a esql:get-columns. Este atributo puede tener el valor de lower o upper, indicando el caso preferido para la generación automática de los nombres de etiquetas.

Otras alternativas más flexibles toman ventaja de algunos elementos ESQL adicionales que permiten el acceso directo a los valores de las columnas individuales en los resultados.

Procesando líneas individuales

Donde se necesite mayor control de los resultados de procesamiento, la hoja lógica ESQL provee un número de elementos *getter* que proporcionan acceso especifico del tipo a las columnas individuales en los resultados de una consulta. Estos métodos *get* son equivalentes directos de los numerosos métodos *get* del objeto java.sql.ResultSet.

El siguiente código muestra como insertar datos de los resultados en elementos específicamente nombrados:

```
<esql:results>
        <esql:row-results>
                <category>
                        <id><esql:get-int column="cat_id"/></id>
                        <name><esql:get-string column="name"/></name>
                </category>
        </esql:row-results>
</esql:results>
```

Note que la columna numérica cat_id es referenciada usando el elemento esql:get-int, mientras que el nombre de la se accede usando esql:get-string. Hay variaciones de este elemento para acceder a boolean, date, long y otros tipos.

Es posible mezclar estas etiquetas ESQL con otras etiquetas XSP, permitiendo libertad para mapear los

resultados a XML de muchas formas diferentes. A continuación se muestra el uso del elemento xsp:attribute para crear atributos en lugar de elementos en los resultados de la consulta:

```
<esql:results>
	<esql:row-results>
		<category>
			<xsp:attribute name="id">
				<esql:get-string column="cat_id"/>
			</xsp:attribute>
			<xsp:attribute name="name">
				<esql:get-string column="name"/>
			</xsp:attribute>
		</category>
	</esql:row-results>
</esql:results>
```

El resultado XML es:

```
<categories>
	<category id="1" name="Blues"/>
	<category id="2" name="Classical"/>
	 ...más resultados...
</categories>
```

Aunque en general es seguro usar el elemento get-string sin preocuparnos del tipo del dato en la base de datos, especificar el tipo asegura que se halla colocado en el tipo o objeto Java apropiado. Esto es útil cuando los resultados de la consulta serán procesados de otra forma, como por ejemplo en código personalizado. Un ejemplo trivial de esto utiliza código Java para asignar un atributo adicional a los resultados, dependiendo de que si el identificador de la categoría es par o impar:

```
<esql:results>
	<esql:row-results>
		<category>
			...xsp:attribute elements as before...
			<xsp:logic>
				if (<esql:get-int column="cat_id"/> % 2 == 0) {
					<xsp:attribute name="rowtype">even</xsp:attribute>
				} else {
					<xsp:attribute name="rowtype">odd</xsp:attribute>
				}
			</xsp:logic>
		</category>
	</esql:row-results>
</esql:results>
```

Esta ayuda muestra que los elementos ESQL solo proporcionan un atajo para llamar los métodos del objeto ResultSet y que pueden ser fácilmente mezclados con código Java dentro de elementos xsp:logic.

Otros elementos para manipular datos

La hoja lógica ESQL incluye otros elementos de tipo result- y row-level que proporcionan mayor soporte al procesamiento de resultados. De nuevo, la mayoría de estos elementos tiene un equivalente en los objetos ResultSet y ResultSetMetaData.

Estos elementos son:

- esql:get-column-count – Retorna la cantidad de columnas en los resultados. Use este elemento para configurar ciclos donde se necesita procesar las columnas en secuencia.
- esql:get-metadata – Retorna una referencia a el ResulSetMetaData.
- esql:get-resultset – Retorna una referencia a ResultSet para manipulación directa
- esql:get-row-position – Toma la posición de la línea actual en los resultados. Use este elemento para asignar números de línea en tablas o similares.
- esql:get-column-name – Toma el nombre de la columna actual, que debe ser referenciada por su posición. Use este elemento cuando secuencialmente procese las columnas en un resultset.
- esql:is-null -- Verifica si una columna dada tiene valor NULL.
- esql:get-xml – Agrega a la funcionalidad proveida por un ResulSet JDBC. El valor de la columna nombrada se toma y se analiza como XML. El valor de la columna puede ser opcionalmente envuelto en otro elemento (especificado por un atributo root) antes de analizar. Use este elemento para ocasiones cuando datos planos XML están almacenados en la base de datos.

Manejando resultados vacíos y errores

El ejemplo anterior siempre asume que la consulta siempre se ejecuta con éxito y retorna resultados. Esto no es siempre el caso: Algunas consultas no retornan datos y pueden ocurrir errores en la base de datos (por ejemplo debido a fallos en la conexión). La hoja lógica de ESQL provee etiquetas que permiten manejar estas situaciones.

Si una consulta no retorna resultados, es posible tomar acción usando el elemento esql:no-results. Ejemplo:

```
<esql:results>
        <results/>
</esql:results>
<esql:no-results>
        <no-results/>
</esql:no-results>
```

El elemento esql:error-results provee un gancho para procesar SQLException generada durante el procesamiento de los resultados de la consulta. Para acceder al trazado de pila y el mensaje asociado con la excepción:

```
<esql:error-results>
        <error>
                <message><esql:get-message/></message>
                <trace><esql:get-stacktrace/></trace>
                <string><esql:to-string/></string>
        </error>
</esql:error-results>
```

Es importante diferenciar las condiciones de excepción que pueden ocurrir cuando se están procesando los resultados de aquellas que ocurren cuando una conexión a la base de datos está siendo creada o de consultas que contienen errores de sintaxis. Estos errores más serios no se pasan a la página XSP y son manejados directamente por Cocoon.

Pasando parámetros a las consultas

Pasar parámetros desde una solicitud HTML a la consulta de la base de datos es muy útil. La realizarla se usa una combinación de las hojas lógicas ESQL y XSP-Request.

Por ejemplo, usando la base de datos de ejemplo, es útil sacar la lista de todos los álbumes de una categoría. A continuación un SELECT de todas las columnas de la tabla album que se debe filtrar según cat_id:

```
<esql:query>
        select * from album
                where cat_id = <xsp-request:get-parameter name="id"/>
</esql:query>
```

Se llama usando http://localhost:8080/dbc2/albums-in-category.xml?id=1 para retornar la lista de todos los álbumes en la categoría cuyo id es uno. Si no hay ningún álbum, entonces retorna una lista vacía; sin embargo una categoría no existente causa que se genere un elemento no-such-category.

Anidando y agrupando resultados de consultas

El mapeo de datos relacionales al modelo jerárquico de un documento XML muchas veces requiere de la habilidad de poder ejecutar consultas anidadas de los datos relacionales. Esto permite que los registros de una tabla (*la principal*) sea mapeada dentro de elementos conteniendo otros elementos hijos que han sido generados de los datos relacionados en otras tablas (*el detalle*).

Una de las posibles formas de mapear la base de datos de ejemplo en un documento XML es:

```
<categories>
	<category id="1">
		<name>...</name>
		<albums>
			<album>
				<id>1</id>
				<title>...</title>
				<artist>...</artist>
				<tracks>...</tracks>
			</album>
			<album>
				<id>2</id>
				<title>...</title>
				<artist>...</artist>
				<tracks>...</tracks>
			</album>
		</albums>
	</category>
	<category id="2">
		<name>...</name>
		<albums/>
	</category>
</categories>
```

Note que cada elemento categoría puede tener cualquier número de elementos anidados album. Hay dos formas de lograr esto en ESQL. La primera es mediante el uso de consultas anidadas y la segunda forma es una función ESQL que permite automáticamente agrupar los datos generados de una consulta.

Consultas anidadas

Un elemento esql:connection puede tener múltiples elementos anidados esql:execute-query, permitiendo que los resultados de las consultas sean anidados dentro de otras:

```
<esql:connection>
	<esql:execute-query>
		<esql:query/>
		<esql:results>
			<!-- resultados de la primera consulta -->
			<esql:execute-query>
				<esql:query/>
				<esql:results>
					<!-- resultados de la segunda consulta -->
				</esql:results>
```

```
                    </esql:execute-query>
                </esql:results>
            </esql:execute-query>
</esql:connection>
```

Estas consultas anidadas no tienen la obligación de interactuar una con la otra, o ellas pueden tomar sus parámetros de la solicitud HTTP, así que no hay sintaxis adicional requerida. Sin embargo, en algunas circunstancias la consulta interna es dependiente de la consulta externa. En el ejemplo anterior, la consulta externa solo debe de generar la lista de categorías mientras que la consulta interna debe retornar solo los álbumes de una categoría en particular.

La consulta externa es clara y no la explicaremos aquí. La consulta interna introduce una nueva pieza de sintaxis ESQL:

```
<esql:query>
        select alb_id as id, title, artist, num_tracks as tracks
                from album
                where cat_id = <esql:get-int column="cat_id" ancestor="1"/>
</esql:query>
```

Note que en la clausula WHERE de la consulta interna, se usa el elemento esql:get-int para tomar el valor de la categoría actual en la consulta externa. El nuevo elemento *ancestor* indica desde cual consulta externa se tomará el valor –- tal como el resultado asociado con el primer ancestro esql:execute-query.

Esta es una técnica poderosa, porque se pueden anidar cualquier cantidad de consultas dentro de otra y extraer datos desde cualquier consulta anidada de la estructura como sea necesario. Esta técnica es similar a la usada en las herramientas para la creación de informes tradicionales, donde una sola consulta principal proporciona el contexto para una serie de otras consultas que son usadas para construir el informe resultante.

Agrupando datos

Como un método alternativo para lograr el mismo resultado, es posible tomar todos los datos requeridos en una sola consulta y luego usar el elemento de agrupamiento ESQL para identificar como se desea que se desglosen los datos en secciones anidadas.

Un simple join de las tablas de categoría y álbum puede generar los siguientes resultados:

cat_id	***name***	***alb_id***	***title***
1	Blues	1	The Healer
1	Blues	2	Mr Lucky

cat_id	*name*	*alb_id*	*title*
2	Classical	3	The Four Seasons

Note que las primeras dos líneas tienen álbumes que están en la misma categoría y por eso tienen los mismos resultados en la columna cat_id. Estas plantillas en el resultado de join de las tablas puede ser explotado por los elementos esql:group y esql_member para unir los registros relacionados. En este ejemplo, los álbumes que comparten el mismo cat_id se listan juntos:

```
<esql:row-results>
        <esql:group group-on="cat_id">
                <category>
                <!-- procesamiento de los datos de categoría -->
                        <albums>
                                <esql:member>
                                        <album>
                                                <!-- procesamiento de los datos del álbum -->
                                                <id><esql:get-string column="alb_id"/></id>
                                        </album>
                                </esql:member>
                        </albums>
                </category>
        </esql:group>
</esql:row-results>
```

El elemento <esql:group group> tiene un atributo group-on. Este atributo identifica la columna en los resultados que puede ser usada para distinguir los elementos externos (osea las categorías). Los contenidos del elemento define el procesamiento que será aplicado para cada categoría distinta, como lo opuesto a cada línea del resultado.

El elemento esql:member define el procesamiento para cada línea que que comparte la columna compartida definida por el atributo group-on. Aquí el elemento contiene el procesamiento requerido para la estructura del XML para cada álbum, con la estructura final XML siendo idéntica a la generada por la versión con consultas anidadas.

Entendiendo como generar datos XML de una consulta de base de datos es solo un aspecto de la funcionalidad de ESQL. También es posible usar hojas lógicas para permitir que los datos sean insertados, actualizados o borrados.

Insertar, actualizar y borrar

Estas tres operaciones en los datos de la bases de datos comparten una sola funcionalidad: ellos actualizan el estado actual de la base de datos porque agregan, eliminan o alteran registros. Esto es diferente del SELECT, el cual solo retorna el estado actual de la base de datos. ESQL soporta el rango completo de estas operaciones, todas pueden ser ingresadas en el elemento esql:query. Sin embargo, como se dijo anteriormente, el resultado de estas operaciones se reporta de forma diferente.

En lugar de usar el elemento esql:results descrito arriba, se debe usar el elemento esql:update-results para procesar los resultados de estas tres operaciones. Además a diferencia del gran rango de elementos disponibles para procesar los resultados de las consultas, las actualizaciones solo retornan la cuenta de la cantidad de elementos afectados por la operación. Este dato puede ser accedido usando el elemento esql:get-update-count.

Ejemplo de insertar:

```
<esql:execute-query>
        <esql:query>
                insert into album values
                (
                        <xsp-request:get-parameter name="id"/>,
                        <xsp-request:get-parameter name="cat"/>,
                        '<xsp-request:get-parameter name="title"/>',
                        '<xsp-request:get-parameter name="artist"/>',
                        sysdate,
                        <xsp-request:get-parameter name="tracks"/>
                )
        </esql:query>
        <esql:results/>
        <esql:no-results/>
        <esql:error-results>
                <error>
                        <message><esql:get-message/></message>
                        <trace><esql:get-stacktrace/></trace>
                        <string><esql:to-string/></string>
                </error>
        </esql:error-results>

        <esql:update-results>
                <inserted><esql:get-update-count/></inserted>
        </esql:update-results>
  </esql:execute-query>
```

Ejemplo de actualizar:

```
<esql:execute-query>
        <esql:query>
                update album set
                        cat_id=<xsp-request:get-parameter name="cat"/>,
                        title='<xsp-request:get-parameter name="title"/>',
                        artist='<xsp-request:get-parameter name="artist"/>',
                        submitdate=sysdate,
                        num_tracks=<xsp-request:get-parameter name="tracks"/>
                        where alb_id=<xsp-request:get-parameter name="id"/>
        </esql:query>
        <esql:results/>
        <esql:no-results/>
        <esql:error-results/>
        <esql:update-results>
                <updated><esql:get-update-count/></updated>
        </esql:update-results>
</esql:execute-query>
```

Ejemplo de eliminar:

```
<esql:execute-query>
        <esql:query>
                delete from album
                        where alb_id = <xsp-request:get-parameter name="id"/>
        </esql:query>
        <esql:results/>
        <esql:no-results/>
        <esql:error-results/>
        <esql:update-results>
                <deleted><esql:get-update-count/></deleted>
        </esql:update-results>
  </esql:execute-query>
```

Como se puede ver en todos los casos el parámetro para la operación de actualización se toma de la solicitud HTTP. El número de registros afectados se reporta en un elemento inserted, deleted o update.

Si solicitamos por ejemplo http://localhost:8080/dbc2/delete-album?id=1 para borrar el álbum cuyo cat_id es uno. El resultado es:

```
<albums>
        <deleted>1</deleted>
</albums>
```

Si se vuelve a solicitar por segunda vez, la cuenta de borrados será cero. Esto permite a los resultados de una operación ser identificados por la tubería de Cocoon. No puede usar los elementos esql:no-results o esql:error-results con este tipo de consultas.

Por un lado, Ud debe tener cuidado cuando diseña este tipo de interfaz. Para demostrarlo, es fácil mostrar la modificación de los datos usando una simple solicitud GET, pero las aplicaciones verdaderas deben usar siempre solicitudes HTTP POST. La especificación de HTTP claramente describe que las solicitudes GET nunca deben tener este tipo de efectos. Es posible identificar fácilmente el tipo de solicitud usando el elemento xsp-request:get-method, como se muestra:

```
<xsp:logic>
        if (<xsp-request:get-method/>.equals("POST"))
        {
                //perform insert or update
        }
        if (<xsp-request:get-method/>.equals("GET"))
        {
                //perform query
        }
        ...etc
</xsp:logic>
```

La otra consideración importante es la de verificar los datos enviados por el usuario para asegurarse que solo datos válidos son insertados.

Validación de formularios

Es el proceso de tomar los datos enviados por el usuario, usualmente ingresados en un formulario HTML y validarlos contra las reglas de negocio de una aplicación. En su forma más simple, la validación envuelve pruebas del tipo y longitud de los campos para asegurarse que estos cumplen con las restricciones de la base de datos. Para generar retroalimentación inmediata al usuario, ejecute estas pruebas tan pronto como el formulario se envía.

Cocoon provee un componente, la acción FormValidator, la cual es capaz de ejecutar unas serie de pruebas de validación. La acción se configura para ejecutar pruebas especificas de los parámetros solicitados mediante la creación de una descripción XML separada de los campos del formulario. Esta descripción se usa como un parámetro por la acción, que puede aplicar las reglas de validación apropiadas y luego avisarle al sitemap si el procesamiento fue exitoso.

Describiendo un Formulario

El ejemplo muestra la sintaxis básica del formato XML soportado por la acción validadora de formularios

para describir campos de formularios. Describe las reglas de validación de los datos enviados por el usuario que ingresó un nuevo álbum a la base de datos de ejemplo:

```
<root>
	<!-- field definitions -->
	<parameter name="id" type="long"
		min="1" max="99999" nullable="no"/>
	<parameter name="cat" type="long"
		min="1" max="999" nullable="no"/>
	<parameter name="title" type="string"
		max-len="100" nullable="no"/>
	<parameter name="artist" type="string"
		max-len="100" nullable="no"/>
	<parameter name="tracks" type="long"
		max="99" nullable="no"/>
	<constraint-set
		name="insert-album">
		<validate name="id"/>
		<validate name="cat"/>
		<validate name="title"/>
		<validate name="artist"/>
		<validate name="tracks"/>
	</constraint-set>
</root>
```

Note lo siguiente: Primero, el elemento root del documento es ignorado así que puede tener cualquier nombre. Segundo, el documento se divide en dos secciones: una serie de definiciones de campos seguida de un *conjunto de restricciones*.

Cada una de las definiciones de campos describe un parámetro enviado por el usuario. Los parámetros deben tener único el atributo name. Los campos deben tener también un tipo, que pueden ser uno de los siguientes: long, double o string. Cada campo puede luego definir una serie de reglas de validación que se especifican como atributos adicionales:

- nullable – identifica si el campo puede ser nulo.
- default – identifica el valor implícito del campo, si no se suple un valor.
- min y max – indica los valores mínimo y máximo
- min-len y max-len – Indica la longitud mínima y la longitud máxima
- matches-regex – define una expresión regular POSIX que debe cumplir el valor del parámetro; esto permite una validación del contenido mucho mas rica, por ejemplo para el formato de la dirección de correo electrónico.

El conjunto de restricciones describe una combinación de los campos definidos que deben ser validados en una sola pasada, como se describirá a continuación. Los elementos validate en un conjunto de restricciones soportan dos atributos adicionales:

- equals-to – define una cadena fija que el parámetro debe match.
- equals-to-param – define el nombre de otro parámetro que debe tener el mismo valor, tales como para comprobar si el usuario ha ingresado correctamente su contraseña dos veces en un formulario que verifica la nueva contraseña.

Validando campos

Ahora que ya se definieron las reglas de validación, el siguiente paso es aplicarlas a una solicitud recibida. Como cualquier otro componente de Cocoon, primero se debe declarar la acción que valida el formulario en el sitemap antes de que esta se pueda usar:

```
<map:components>
        ...
        <map:actions>
                <map:action name="form-validator"
                        src="org.apache.cocoon.acting.FormValidatorAction"/>
                </map:actions>
        ...
</map:components>
```

La acción luego se agrega a una tubería Cocoon de esta forma:

```
<map:match pattern="form/insert-album">
        <map:act type="form-validator">
                <map:parameter name="descriptor"
                        value="context://insert-album.xml"/>
                <map:parameter name="validate-set"
                        value="insert-album"/>
                <!-- if success -->
                ...
        </map:act>
        <!-- if fail -->
        ...
</map:match>
```

La tubería se invoca cuando se envía el formulario “insert-album”. Su primer paso es invocar la acción que valida formularios y envía dos parámetros a este. El primer parámetro, descriptor, define el archivo XML que describe las reglas de validación. El esquema URL context:// se refiere al sistema de archivos bajo la estructura del directorio $COCOON_HOME. El segundo parámetro, validate-set, define un conjunto

de restricciones definido dentro de ese archivo. Todos los parámetros de solicitud HTTP que match un nombre de parámetro en ese conjunto de restricciones son validados usando las reglas configuradas.

Si la validación es exitosa, entonces el otro componente de la tubería dentro del elemento map:act se procesa. Esto es donde la página XSP insert-album.xsp será invocada para insertar los datos que se sabe que son válidos.

Si la validación no es exitosa, solo los componentes después de map:act se ejecutan. De este modo, la función de las acciones es similar a una expresión if: si es exitosa ejecuta ejecuta su procesamiento, osea el bloque de componentes de tubería anidado dentro de este se ejecuta; en caso contrario el bloque no se ejecuta.

Usando la hoja lógica Form Validator, es posible proveer al usuario retroalimentación detallada del porque falló el formulario.

Usando la hoja lógica Form Validator

Solo se pueden usar esta hoja lógica con la acción Form Validator. Se usa normalmente con páginas XSP que se ejecutan si el formulario falla en la validación. Estas acciones almacenan una descripción detallada de sus resultados como parámetros de solicitud HTTP. La hoja lógica provee un API sencillo basado en etiquetas para interpretar estos resultados.

El nombre de espacio de esta hoja lógica es: http://apache.org/xsp/form-validator/2.0.

Cada uno de los siguientes elementos acepta un atributo name, que indica el nombre del campo cuyo estado de validación se está comprobando:

- formval:is-ok – retorna un booleano indicando si el campo fue validado correctamente
- formval:is-null – indica que el campo es nulo cuando este no debe ser null.
- formval:is-toosmall – indica que el campo es mas pequeño que min-len o su valor es menor que el valor min
- formval:istoolarge – indica que el campo es más grande que max-len o su valor es mayor que el valor max
- formval:is-nomatch – indica que el campo fallo el match de la expresión regular configurada.

Es posible usar una prueba simple con un **if** que corresponda a cada uno de estos elementos: formval:is-ok comprueba si el campo nombrado se validó correctamente y si es cierto, su contenido es evaluado. Estas dos variantes permiten una gran flexibilidad, porque el el on-* puede ser usado para pruebas simples, mientras que las versiones if-* pueden ser usadas en código personalizado (en elementos xsp:logic).

En caso que se requiera aplicar varias de estas pruebas al mismo campo, en lugar de usar el atributo name

varias veces, es posible anidar las pruebas dentro de un elemento validate. Este elemento debe tener un atributo name que provee el contexto para cualquier etiqueta validadora de formularios anidada.

Otro Ejemplo de Bases de Datos

Es una excelente sustitución del código Java para consultar bases de datos en las páginas XSP. Utiliza JDBC, por este motivo toda la configuración previa para Cocoon2 es válida si se usa ESQL. La mejor forma de presentar cómo se usa ESQL es mediante un ejemplo. El ejemplo a continuación consulta las líneas 3 a 7 de una institución bancaria con un número de cuenta específico y los resultados se ordenan por fecha. Para simplificar el ejemplo, no hay contenido estático en la página, pero fácilmente se pueden incluir otras etiquetas para incluir información adicional en la página resultante.

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
<xsp:page language="java"
    xmlns:xsp="http://apache.org/xsp"
    xmlns:esql=”http://apache.org/cocoon/SQL/v2>
<page>
<esql:connection>
        <esql:pool>hb_pool</esql:pool>
        <transactions>
                <esql:execute-query>
                        <esql:skip-rows>2</esql:skip-rows>
                        <esql:max-rows>5</esql:max-rows>

                        <esql:query>
                                select * from srv_trans where acc_num = 0180698943 order by trans_date
                        </esql:query>
                        <esql:results>
                                <esql:row-results>
                                        <trans>
                                                <xsp:attribute name="no">
                                                        <xsp:expr><esql:get-row-position/>+1</xsp:expr>
                                                </xsp:attribute>
                                                <id><esql:get-int column="trans_id1"/></id>
                                                <date><esql:get-date column="trans_date" format="dd. MMM.
yyyy"/></date>
                                                <account><esql:get-string column="acc_num"/></account>
                                                <amount><esql:get-double column="amount"/></amount>
                                                <currency><esql:get-string column="curr"/></currency>
                                        </trans>
                                </esql:row-results>
                        </esql:results>
                        <esql:no-results>
                                No records found...
```

```
                    </esql:no-results>
                    <esql:error-results>
                        SQL Exception: <esql:get-message/>
                    </esql:error-results>
                </esql:execute-query>
            </transactions>
</esql:connection>
</page>
</xsp:page>
```

1. Esta página XSP es un documento normal XML por eso empieza con la declaración XML <?xml...>
2. El uso de la librería de etiquetas ESQL se especifica con *namespace xmnls* con <xsp:page>. Esto le dice a Cocoon que debe interpretar las etiquetas ESQL (que debe usar la hoja lógicas ESQL).
3. La conexión a la base de datos se especifica con la etiqueta <esql:connection>. En este caso se usa el pool de conexiones "*hb_pool*". También es posible especificar directamente la configuración de la base de datos (DBURL, Nombre_De_Usuario, Contraseña)
4. El elemento <esql:query> contiene la cadena de consulta SQL SELECT la cual es muy simple en el ejemplo, pero puede ser fácilmente parametrizada.
5. Para ejecutar la consulta los elementos <esql:results> y <esql:row-results> deben ser incluidos en <esql:execute-query>. La etiqueta <esql:results> sirve el multiplicador ResultSets (raro), <esql:row-results> encapsula el ciclo a través de cada una de los registros de un ResultSet.
6. Dentro de <esql:row-results> se construyen elementos *transacción* (<trans>) con elementos hijos *id*, *date*, *account*, *amount* and *currency*. Los valores de los registros del resultado de la consulta se toman usando <esql:get-int>, <esql:get-date>, <esql:get-string> o <esql:get-double> respectivamente.
7. Al final es necesario resolver las excepciones. El elemento <esql:no-result> se usa en caso de que no hallan resultados y el elemento <esql:error-results> se usa en caso de errores JDBC.

El resultado es un XML generado dinámicamente sin código JAXP o JDBC – todo completamente claro y conciso. Lo mejor de todo esto es que cabe la posibilidad de utilizar de mezclar y anidar varias librerías de etiquetas para auxiliar la profundidad y luego fácilmente usar los resultados de una como parámetro para otra.

Dependiendo de los datos, el documento XML resultante del ejemplo anterior puede ser:

```
<?xml version="1.0" encoding="UTF-8"?>
<page xmlns:xspdoc="http://apache.org/cocoon/XSPDoc/v1" xmlns:esql="http://apache.org/cocoon/SQL/v2"
    xmlns:xsp="http://apache.org/xsp">
        <transactions>
            <transaction no="1">
                <id>444</id>
                <date>31. mar. 2001</date>
```

```
                <account>0180698943</account>
                <amount>-10.0</amount>
                <currency>SKK</currency>
            </transaction>
            <transaction no="2">
                <id>344</id>
                <date>30. apr. 2001</date>
                <account>0180698943</account>
                <amount>-10.0</amount>
                <currency>SKK</currency>
            </transaction>
            <transaction no="3">
                <id>141</id>
                <date>31. maj. 2001</date>
                <account>0180698943</account>
                <amount>-10.0</amount>
                <currency>SKK</currency>
            </transaction>
            <transaction no="4">
                <id>849</id>
                <date>30. jun. 2001</date>
                <account>0180698943</account>
                <amount>-10.0</amount>
                <currency>SKK</currency>
            </transaction>
            <transaction no="5">
                <id>4329</id>
                <date>31. jul. 2001</date>
                <account>0180698943</account>
                <amount>-10.0</amount>
                <currency>SKK</currency>
            </transaction>
        </transactions>
</page>
```

Ejemplo de Aplicación Web

Para explicar cómo se crean aplicaciones con Cocoon2, vamos a suponer que deseamos crear una aplicación sencilla de base de datos que administra usuarios y departamentos. Cada elemento tiene un nombre y un identificador. Un departamento puede tener varios empleados, pero cada empleado solo puede pertenecer a un departamento. La aplicación debe permitir crear, cambiar y borrar empleados y departamentos.

Conceptos

El SQL

El SQL que presentamos funciona para PostgreSQL.

```
CREATE TABLE "department" (
    "department_id" INTEGER NOT NULL PRIMARY KEY,
    "department_name" VARCHAR(64) NOT NULL,
);

CREATE TABLE "employee" (
    "employee_id" INTEGER NOT NULL PRIMARY KEY,
    "employee_name" VARCHAR(64) NOT NULL,
    "department_id" INTEGER NOT NULL REFERENCES department(department_id) ON UPDATE CASCADE,
);
```

Funcionalidad

1. Crear Departamento (solo se necesita el nombre)
2. Actualizar Departamento (cambiar nombre, reasignar empleados potenciales al departamento, crear empleados para el departamento)
3. Borrar Departamento
4. Buscar Departamento (por nombre o ID)
5. Crear Empleado (necesita el nombre y crear un departamento si es necesario)
6. Actualizar Empleado (cambiar nombre, reasignar departamento si es necesario)
7. Borrar Empleado
8. Buscar Empleado (por nombre, ID o departamento)

Pantallas

Se deben incluir como un documento separado que presente los interfaces y páginas de resultados a usar.

Desarrollo

Para hacer cualquier cosa en Cocoon2 se necesita un sitemap. No se explicará en detalle, se enseñará como colocar una entrada para poder ver la aplicación. En la mayoría de las situaciones de desarrollo el sitemap es configurado por otra persona. Como se desea empezar desde cero, se copia el sitemap que viene en los ejemplos de Cocoon2 y se borra todo lo que esta bajo la etiqueta <map:pipelines>. Luego en su lugar se se agrega el siguiente código:

```
<map:pipeline>
        <map:match pattern="">
                <map:redirect-to uri="home.html"/>
```

```
        </map:match>

        <map:match pattern="**.xml">
                <map:generate src="docs/{1}.xml"/>
                <map:serialize type="xml"/>
        </map:match>

        <map:match pattern="**.html">
                <map:generate src="docs/{1}.xml"/>
                <map:transform src="stylesheets/apache.xsl"/>
                <map:serialize/>
        </map:match>

        <map:match pattern="images/**.gif">
                <map:read src="resources/images/{1}.gif" mime-type="image/gif"/>
        </map:match>

        <map:match pattern="images/**.jpg">
                <map:read src="resources/images/{1}.jpg" mime-type="image/jpg"/>
        </map:match>

        <map:match pattern="images/**.png">
                <map:read src="resources/images/{1}.png" mime-type="image/png"/>
        </map:match>

        <map:match pattern="resources/**.css">
                <map:read src="resources/styles/{1}.css" mime-type="text/css"/>
        </map:match>

        <map:match pattern="resources/**.js">
                <map:read src="resource/styles/{1}.js" mime-type="application/x-javascript"/>
        </map:match>

        <map:handle-errors>
                <map:transform src="stylesheets/system/error2html.xsl"/>
                <map:serialize status-code="500"/>
        </map:handle-errors>
</map:pipeline>
```

Lo que hace esto es decirle al sitemap que deseamos capturar todos los URL con extensión .xml y buscar su archivo equivalente en el subdirectorio *docs*. No se está realizando ninguna transformación en este momento. El mantenimiento del Sitemap es trabajo del Administrador del Sitio. Hay algunas excepciones a esta regla general, pero se explicarán cuando sea necesario. Se usa el Marcaje de Documento especificado en el StyleBook DTD Format.

Creando las Páginas

En primer lugar se necesita que las páginas cumplan con el estándar XML. Veremos lo que esto nos ayuda para depurar XSP (XML Server Pages). Como ya se ha creado el diseño de pantallas y la base de datos, se creará ahora el markup.

La página de inicio será sencilla: una lista de enlaces a las páginas principales.

```
<document>
	<header>
		<title>Página de Inicio</title>
	</header>
	<body>
		<s1 title="Bienvenidos al Administrador de Personal">
			<p>
				Bienvenidos a nuestro Administrador de Personal.
				Favor ejecutar una de las siguientes funciones:
			 </p>
			<ul>
				<li>
					<link href="search-dept.html">Buscar Departamentos</link>
				</li>
				<li>
					<link href="search-empl.html">Buscar empleados</link>
				</li>
				<li>
					<link href="create-dept.html">Crear Departamento</link>
				</li>
				<li>
					<link href="edit-dept.html">Editar un Departamento</link>
				</li>
				<li>
					<link href="create-empl.html">Crear Empleado</link>
				</li>
				<li>
					<link href="edit-empl.html">Editar un Empleado</link>
				</li>
			</ul>
		</s1>
	</body>
</document>
```

Aunque esto no parezca la gran cosa en estos momentos, se tiene en el sitemap dos entradas “**.xml” y “**.html” para el mismo recurso. Vea “home.html” y vea como se ve ahora. Una gran diferencia. Aún no se

debe de eliminar la etiqueta para ver los documentos en formato XML . Se usará más tarde para depurar las páginas XSP.

El primer formulario

Primero se crearán las plantillas **“crear”**. Es muy importante entender el método correcto de la manipulación de formularios. Aunque es posible crear páginas XSP que ejecutan la lógica automáticamente, este método no es muy fácil de mantener. Además se debe escoger cuando se hará acceso directo a la base de datos o encapsular esta lógica dentro de los objetos.

El acceso directo SQL es más rápido para comenzar, pero es más difícil de mantener al final. Se puede decidir iniciar con acceso directo SQL al inicio del proyecto y construir los objetos después. Con esto en mente, se usará la funcionalidad que Cocoon 2 tiene integrado para hacer esto un poco más fácil. Cocoon 2 tiene un grupo de acciones de Bases de datos que permite mapear campos del formulario a llamadas SQL dinámicamente creadas. También tiene una hoja de estilo que hace la creación de páginas amarradas un poco más fácil.

El primer formulario es el formulario “Crear Departamento”. A la especificación del sitio web le faltan las etiquetas para la construcción de formularios, aquí se dará un ejemplo de esto:

```
<document>
        <header>
                <title>Departamento</title>
        </header>
        <body>
                <s1 title="Crear a Departamento">
                        <form handler="create-dept.html">
                                <p>
                                        Puede crear un departamento escribiendo
                                        su nombre, luego presione el botón “Crear Departamento”
                                </p>
                                <p>
                                        Nombre: <text name="name" size="30" required="true"/>
                                </p>
                                        <submit name="Crear Departamento"/>
                                <note> * Estos campos son requeridos. </note>
                        </form>
                </s1>
        </body>
</document>
```

Es importante notar que la etiqueta submit se convierte en un botón HTML “submit” con el nombre

"*cocoon-action-ACTIONNAME*". El parámetro del formulario "*cocoon-action-ACTIONNAME*" es un valor mágico que usa Cocoon 2 para elegir una acción especifica de un grupo de acciones que solo se ejecutan en ese momento. Se puede ver que esta página se presenta correctamente, pero todavía no hace nada. El manejador es donde va la navegación una vez que se presiona el botón "*Crear Departamento*". Lo que vamos hacer ahora es crear una página de confirmación para todas las páginas de Departamentos y Empleados.

Cocoon 2 tiene acciones validadoras de formularios que se ocuparán de asegurarse que los valores ingresados sean correctos. También tiene las siguientes acciones de Bases de Datos: *DatabaseAddAction*, *DatabaseUpdateAction*, *DatabaseDeleteAction* y *DatabaseAuthenticatorAction*. Necesitaremos sólo las acciones *Add*, *Update* y *Delete* para nuestra aplicación. Para poder preparar estas acciones, creamos un archivo XML de configuración que le dice a las acciones como mapear los parámetros solicitados a las tablas de la base de datos y colocar restricciones en los parámetros. Para el grupo de formularios de Departamento, este será:

```
<root>
        <!-
                Los elementos "parameter" identifican las restricciones de la raiz "root" para la acción
FormValidatorAction.
                Definimos que el parámetro "id" es entero (otros pueden ser "long", "double", "boolean" y "string").
                Definimos que el parámetro "name" es una cadena con al menos 5 caracteres y no mas de 64 caracteres.
         -->
        <parameter name="id" type="long"/>
        <parameter name="name" type="string" min-len="5" max-len="64"/>
        <!-
                Cada grupo de restricciones se usa cuando definimos una nueva forma de validar el formulario. Los grupos
                de restricciones se definen por función. Porque que tenemos la misma forma de manejar el Validador de
                Formularios, tenemos un grupo para actualizar y un grupo para agregar.

                Es posible agregar restricciones adicionales a las restricciones implícitas definidas arriba. Además Ud. no
                "tiene" que reforzar una restricción. Cada elemento "validate" de abajo identifica las restricciones del
                parámetro que estamos reforzando. Para más información vea los JavaDocs de AbstractValidatorAction
         -->
        <constraint-set name="update">
                <validate name="name"/>
                <validate name="id" nullable="no" min="1"/>
        /constraint-set>

        <constraint-set name="add">
                <validate name="name"/>
        </constraint-set>
        <!--
                Aquí es donde definimos los mapeos de tablas con los que las DatabaseActions pueden hacer su mágia.
```

```
                Note que los nombres de los parámetros son los mismos de arriba – como también son los mismos
                nombres de los parámetros del formulario.

                Primero le decimos a las Acciones de Base de Datos que vamos a usar el pool de conexiones “personnel”
                que configuramos en cocoon.xconf. Este archivo lo debe configurar el administrador.

                También le decimos a las Acciones de Bases de Datos la estructura de la tabla con que vamos a llenar.
                “keys” se usa para identificar cuales columnas deben ser tratadas como llaves – estas se manejan diferente
                cuando se crean las expresiones SQL. Note que hay un atributo “mode” en el elemento “key”.
                El mode se refiere a la forma en que se deben generar las llaves nuevas. Hay tres modos:
                1- “automatic” - la llave la genera la base de datos.
                2- “manual” - la llave se genera buscando el valor mas alto + 1.
                3- “form” - la llave se tomará del valor del parámetro en el formulario.

                Ambos elementos “keys” ans “values” sirven para mapear los nombre de los parámetros a las columnas en la
                tabla, convirtiendo su valor en su tipo nativo.

                Si quiere ver la lista de tipos soportados busque AbstractDatabaseAction en JavaDocs
         -->
        <connection>personnel</connection>
        <table name="department">
                <keys>
                        <key param="id" dbcol="department_id" type="int" mode="manual"/>
                </keys>
                <values>
                        <value param="name" dbcol="department_name" type="string"/>
                </values>
        </table>
</root>
```

Después de crear el archivo descriptor, deberá crear algunas entradas en el Sitemap para poder utilizar el descriptor de formularios. En primer lugar el Sitemap debe saber como referenciar las acciones que queremos. Para hacer esto, modifique la sección “*map:actions*” para que liste todas las acciones que necesitamos:

```
 <map:actions>
        <map:action name="dbAdd"
                src="org.apache.cocoon.acting.DatabaseAddAction"/>
        <map:action name="dbDel"
                src="org.apache.cocoon.acting.DatabaseDeleteAction"/>
<map:action name="dbUpd"
                src="org.apache.cocoon.acting.DatabaseUpdateAction"/>
        <map:action name="form"
                src="org.apache.cocoon.acting.FormValidatorAction"/>
</map:actions>
```

Finalmente, debemos crear un conjunto de acciones. Un conjunto de Acciones es un grupo de acciones que deben ser aplicadas de un solo. Si el conjunto de acciones tiene un parámetro "*action*", entonces la acción especifica solo se ejecuta si el ACTIONNAME del parámetro mágico solicitado "*cocoon-action-ACTIONNAME*" es igual al valor del parámetro "*action*". En nuestro caso, el conjunto de acción que debemos definir es este (se debe definir en el sitemap):

```
<map:action-sets>
        <map:action-set name="process">
                <map:act type="form" action="Create Department">
                        <map:parameter name="validate-set" value="add"/>
                        <map:act type="dbAdd"/>
                </map:act>
                <map:act type="form" action="Update Department">
                        <map:parameter name="validate-set" value="update"/>
                        <map:act type="dbUpd"/>
                </map:act>
                <map:act type="dbDel" action="Delete Department"/>
        </map:action-set>
</map:action-sets>
```

Ahora que definimos las acciones que deseamos, con los parámetros que lo controlan en tiempo de ejecución, podemos usarlo en nuestra tubería. Agregue el siguiente código en el archivo sitemap:

```
<map:match pattern="*-dept.html">
        <map:act set="process">
                <map:parameter name="descriptor"
                        value="context://docs/department-form.xml"/>
                <map:parameter name="form-descriptor"
                        value="context://docs/department-form.xml"/>
                <map:generate type="serverpages" src="docs/confirm-dept.xsp"/>
                <map:transform src="stylesheets/apache.xsl"/>
                <map:serialize/>
        </map:act>
        <map:generate type="serverpages" src="docs/{1}-dept.xsp"/>
        <map:transform src="stylesheets/apache.xsl"/>
        <map:serialize/>
</map:match>

<map:match pattern="*-dept.xml">
        <map:act set="process">
                <map:parameter name="descriptor"
                        value="context://docs/department-form.xml"/>
                <map:parameter name="form-descriptor"
```

```
                value="context://docs/department-form.xml"/>
            <map:generate type="serverpages" src="docs/confirm-dept.xsp"/>
            <map:serialize type="xml"/>
        </map:act>
        <map:generate type="serverpages" src="docs/{1}-dept.xsp"/>
        <map:serialize type="xml"/>
</map:match>
```

Puede que no es claro que es lo que pasa ahora. La forma en que las acciones trabajan es que si retornan null, nada dentro de la entrada "*map:act*" se ejecutará y el procesamiento solicitado fluirá dentro de la segunda sección "*map:generate*". Este es un efecto lateral del uso de FormValidatorAction. Si elejimos crear nuestro propios objetos de negocios y nuestro marco validador de formularios, no estaremos restringidos por esta construcción.

Además hemos cambiado el tipo de generador que usamos: lo convertimos en un generador "servepages" o XSP. Hemos hecho el cambio en este momento así que podemos reportar información al usuario acerca de lo que falló. Primero, necesitamos convertir nuestro archivo "*create-dept.xml*" a un página XSP para que podamos ver otra vez la página (en este momento obtendremos un error). Para hacer esto, solo agregue una nueva etiqueta a la base del documento llamada "*xsp:page*" declarando el nombre de espacio XSP. El cambio debe ser:

```
<xsp:page xmlns:xsp="http://apache.org/xsp">
        <!-- aquí debe quedar el documento original -->
</xsp:page>
```

Para completar la transformación, debemos cambiar la extensión a "*.xsp*" para que sepamos con que estamos trabajando. Cree un nuevo archivo llamado "*confirm.xsp*" con el siguiente contenido:

```
<xsp:page xmlns:xsp="http://apache.org/xsp">
        <document>
                <header>
                        <title>Departamento</title>
                </header>
                <body>
                        <s1 title="Departamento Procesado">
                                <p>
                                        Ha procesado el departamento con éxito.
                                </p>
                        </s1>
                </body>
        </document>
</xsp:page>
```

Agregando soporte para reportar errores

Para que podamos reportar errores del procesamiento de página, agregue otra declaración de namespace al elemento "*xsp:page*". Al final la página del formulario debe verse de esta forma:

```
<xsp:page xmlns:xsp="http://apache.org/xsp"
        xmlns:xsp-formval="http://apache.org/xsp/form-validator/2.0">
<document>
        <header>
                <title>Departamento</title>
        </header>
        <body>
                <s1 title="Crear a Departamento">
                        <form handler="create-dept.html">
                                <p>
                                        Puede agregar un departamento escribiendo su nombre y
                                        presionando el botón "Crear Departamento"
                                </p>
                                <p>
                                        Nombre: <text name="name" size="30" required="true"/><br />
                                        <xsp:logic>
                                                if (<xsp-formval:is-toosmall name="name"/>) {
                                                        <xsp:text>"Name" must be at least 5 characters</xsp:text>
                                                } else if (<xsp-formval:is-toolarge name="name"/>) {
                                                        <xsp:text>"Name" was too long</xsp:text>
                                                }
                                        </xsp:logic>
                                </p>
                                <submit name="Crear Departamento"/>
                                <note>
                                        * Estos campos son requeridos.
                                </note>
                        </form>
                </s1>
        </body>
</document>
</xsp:page>
```

Agregando soporte a bases de datos con la hoja de estilo ESQL

La página "*Crear Empleado*" requiere acceso a bases de datos para que sepamos a qué departamento se asignará el empleado nuevo. Esto es bien fácil de hacer con la hoja de estilo ESQL. De nuevo cuando usamos la hoja de estilo ESQL perdemos un poco de la ventaja de separación de contenido

```
<xsp:page xmlns:xsp="http://apache.org/xsp"
```

```
        xmlns:xsp-formval="http://apache.org/xsp/form-validator/2.0"
        xmlns:esql="http://apache.org/cocoon/SQL/v2">
<document>
        <header>
                <title>Empleado</title>
        </header>
        <body>
                <s1 title="Crear un Empleado">
                        <form handler="create-empl.html">
                                <p>
                                        Puede crear un empleado escribiendo el nombre y
                                        presionando el botón “Agregar Empleado”.
                                </p>
                                <p>
                                        Nombre: <text name="name" size="30" required="true"/><br />
                                                <xsp:logic>
                                                        if (<xsp-formval:is-null name="name"/>) {
                                                                <xsp:text>"Nombre" no puede quedar vacio</xsp:text>
                                                        } else if (<xsp-formval:is-toolarge name="name"/>) {
                                                                <xsp:text>"Nombre" muy largo</xsp:text>
                                                        }
                                                </xsp:logic>
                                </p>
                                <p>
                                        Department:
                                        <select name="department">
                                        <esql:connection>
                                                <!-- declare el pool de conexiones que usamos -->
                                                <esql:pool>personnel</esql:pool>

                                                <!-- Se pueden repetir bloques de ejecución de consultas -->
                                                <esql:execute-query>

                                                        <!-- Encuentre todos los departamentos y ordenelos -->
                                                        <esql:query>
                                                                SELECT department_id, department_name
                                                                FROM department ORDER BY department_name
                                                        </esql:query>

                                                        <!-- Que hacer con los resultados -->
                                                        <esql:results>
                                                        <!--
                                                                Una consulta exitosa que retorna resultados ejecuta
                                                                este bloque. Es posible meter mas bloques
                                                                “execute-query” dentro de row-results. De esta forma es
```

```
                                        posible tener consultas que filtren la información basada
                                        en los resultados de otras consultas.
                                -->
                                <esql:row-results>
                                        <option>
                                                <xsp:attribute name="name">
                                                 <esql:get-string column="department_id"/>
                                                </xsp:attribute>
                                                 <esql:get-string column="department_name"/>
                                        </option>
                                </esql:row-results>
                                <!--
                                        Otros tipos de resultado son “no-result” y
                                        “error-results”. Una consulta exitosa que no retorna
                                        resultados (un resultado vacio) usará el XML embebido
                                        en la sección “no-results”. Una consulta que falló y
                                        lanzó una excepción usará el XML embebido en la
                                        sección “error-results”
                                -->
                                </esql:results>
                        </esql:execute-query>
                </esql:connection>
        </select>
</p>
<submit name="Crear Empleado"/>
<note>
        * Estos campos son requeridos.
</note>
</form>
</s1>
</body>
</document>
</xsp:page>
```

Como se puede apreciar ESQL es flexible y poderoso, pero el precio de la flexibilidad es la pérdida de facilidad de lectura. Usar una hoja de estilo para agrupar la información en un objeto de negocios es otra alternativa. Vea como trabaja ESQL:

- Primero definimos la información de conexión que usarán todas las consultas en la estructura ESQL.
- Luego, definimos la primera consulta que vamos a usar. Note que puede anidar consultas y también tener más de una consulta en un elemento “*esql:connection*”.
- Al final, definimos como procesar los resultados. Hay tres tipos de resultados: “*esql:row-results*”, “*esql:no-results*” y “*esql:error-results*”. Estos nos permiten manejar diferentes escenarios fácilmente.

Dentro de cada uno de los elementos de resultado podemos anidar nuevas consultas a procesar.

Una nota sobre las acciones

Las acciones son el pan y la mantequilla del procesamiento lógico en Cocoon. Hay algunas técnicas que podemos usar al desarrollar las acciones. Se pueden crear acciones para cada pieza de la lógica de negocios. Esta técnica es muy difícil de manejar y requiere que una gran parte del tiempo de desarrollo creando acciones.

La técnica recomendada para crear acciones es proporcionar acciones genéricas que puedan manejar un gran rango de acciones especificas. Ejemplos de esta técnica son DatabaseActions y ValidatorActions. Estas leen un archivo de configuración especificado por un parámetro y luego modifican los resultados específicos basados en el archivo de configuración. Para poder usar esta técnica en sus propias acciones, es posible extender el AbstractComplimentaryConfigurationAction. Básicamente lo que hace es encapsular la lógica para leer y cachear la información de configuración para su Acción.

Redirecciones

La mayoría de los desarrolladores Web aceptan que redirigir según el usuario en la entrada es una parte valiosa y necesaria del desarrollo Web. En Cocoon solo hay 2 lugares donde se pueden editar redirecciones, en el Sitemap y en las Acciones. En esencia, Cocoon nos obliga a planificar de tal forma que las redirecciones se usen solo donde sea necesario.

Una técnica que es bueno usar es obligar a todo el tráfico que pase por un URL de acción de control. Esta acción comprobará si el usuario ha ingresado y si no lo enviará a una pagina de ingreso. Otro uso de esta técnica es probar los permisos de usuario y si el no tiene acceso, lo enviamos a otra página.

Escribiendo Acciones

Escribir una acción es tan simple como escribir un Componente que cumpla al interfaz de la Acción. Asegúrese de examinar todas las acciones del paquete org.apache.cocoon.acting – puede encontrar algunas acciones abstractas que puede extender. Las Acciones son componentes de Avalon, para más información ver en http://jakarta.apache.org/avalon/ las páginas blancas de Avalon.

Las acciones retornan un mapa que contiene valores que el administrador de sitemap puede usar en el sitemap. Si la acción retorna NULL, entonces no será ejecutado nada dentro del elemento "*map:act*".

Valores de Retorno

El interfaz de acciones especifica que el retorna un mapa. Este mapa se usa para reemplazar valores en el sitemap y comunicar información a otras acciones. Cuando una acción se especifica en el sitemap, usa la siguiente sintaxis.

```
<map:act type="mi-accion">
        <map:generate src="{source}"/>
        <map:transform src="doc2{theme}"/>
        <map:serialize/>
</map:act>
```

Esta pieza de código asume que tenemos ya especificada una acción de nombre "*mi-accion*". También asume que que hay dos parámetros retornados de la acción en el mapa. El sitemap consulta en el mapa retornado los valores "*source*" y "*theme*" y sustituye sus valores en el lugar entre llaves que lo referencia. En otras palabras, cuando el sitemap ve el elemento "*map:generate*" con un atributo src de "*{source}*" el lo busca en el mapa. En nuestro caso supongamos que el valor guardado es "*index.html*". El Sitemap ejecutará la sustitución y así el atributo src ahora contendrá "*index.html*".

En el caso que la acción de arriba retorne un valor nulo, todo lo que este dentro del elemento "*map:act*" será saltado. Es posible tomar provecho de esto como lo hace el Validador de Acciones. Si todo se valida correctamente el retorna un mapa. Si hay un error, el retorna nulo y coloca la información en los atributos Request.

Selectores

Los selectores tienen un rol similar a los matchers con flexibilidad adicional. Están diseñados para evaluar una expresión booleana simple dependiente de alguna parte del ambiente (por ejemplo: el URI solicitado, cabeceras, o cookies). El resultado de esta evaluación determina cual cuales fragmentos de la tubería se combinarán dentro de la tubería dada. A diferencia de los matchers, los selectores pueden ser componentes activos en la decisión del rumbo a tomar. Por ejemplo, un matcher hace una sencilla decisión de tipo si/no. Si un match es exitoso, se ejecuta la tubería. En caso contrario, se ignora. Los selectores van más allá permitiendo casos de uso más complejos u opciones múltiples. En resumen, se deben considerar los matcher como expresiones “if” simples. Por extensión se deben considerar los selectores como construciones “if-else if-else” o “switch case”. La sintaxis de los selectores es muy similar a las expresiones <xsl:test> utilizadas en XSLT.

Como ejemplo consideremos el escenario típico en el cual una página será presentada de forma diferente según el tipo de navegador del cliente. Dado el gran numero y la diversidad de navegadores disponible es muy difícil y complicado resolver este problema utilizando matchers. El BrowserSelector verifica el parámetro dado con el agente del cliente en el encabezado de la solicitud. Utilizando un solo selector es posible crear una configuración consistente y clara.

```
<map:match pattern="docs/*.html">
        <map:generate src="xdocs/{1}.xml"/>

        <map:select type="browser">
                <map:when test="netscape">
                        <map:transform src="stylesheets/netscape.xsl" />
                </map:when>
                <map:when test="explorer">
                        <map:transform src="stylesheets/ie.xsl" />
                </map:when>
                <map:when test="lynx">
                        <map:transform src="stylesheets/text-based.xsl" />
                </map:when>
                <map:otherwise>
                        <map:transform src="stylesheets/html.xsl" />
                </map:otherwise>
        </map:select>
        <map:serialize/>
</map:match>
```

Selectores en Cocoon

Los selectores disponibles en Cocoon son:

- **BrowserSelector:** comprueba el valor del parámetro “test” con el agente del cliente en el encabezado HTTP, permite reconocer el navegador que generó la solicitud.
- **CodeSelector:** verifica un fragmento de código Java dado como parámetro "test" con el ambiente.
- **HostSelector:** verifica el parámetro "test" con el Host en el encabezado de la solicitud
- **ParameterSelector:** verifica la cadena especificada en el parámetro "test" con un parámetro interno de Cocoon (por ejemplo sitemap);
- **HeaderSelector:** igual que ParameterSelector, pero verifica con el encabezado de la solicitud.
- **RequestSelector:** igual que ParameterSelector, pero verifica con los parametros de la Solicitud;
- **SessionSelector:** igual que ParameterSelector, pero verifica con un atributo arbitrario de la sesión.

Trabajando con XSP en Apache Cocoon 2

¿Qué es XSP?

XML Server Pages es una tecnología de Cocoon 2 que activa la creación dinámica de fuentes de datos XML para alimentar datos en las tuberías de Cocoon 2. Estas fuentes de datos se describen usando una combinación de XML y lógica de aplicación que es luego automáticamente compilada en clases Java por el motor de Cocoon 2.

XSP proporciona una plataforma flexible para desarrollar aplicaciones usando Cocoon 2. Por ejemplo, la información en una aplicación de base de datos existente puede ser presentada por aplicaciones Cocoon 2, aumentando la variedad de opciones para la entrega de datos. XSP permite a Cocoon 2 ser usado en un ambiente de integración de aplicaciones tales como middleware y publicación de documentos porque provee la forma de presentar la fuente de datos a través de un interfaz XML.

Comparación entre XSP y JSP

Java Server Pages es la forma más usada para generar interfaces dinámicos Web usando una combinación de Java y HTML o XML. El documento resultante es entrelazado con etiquetas personalizadas y/o piezas de código Java que agregan secciones dinámicas. Al preservar la estructura de documentos hace el mantenimiento de páginas JSP mas fácil para los no programadores. Aunque el lenguaje de marcado es enfatizado para el usuario final, un contenedor de servlets compilará una página JSP a un Servlet Java estándar.

XSP tiene mucho en común con JSP. Ambas tecnologías:

- Consisten de una mezcla de código de programa y lenguaje de marcado
- Son compilados en formas binarias para ser ejecutados
- Permiten la creación de librerías de etiquetas personalizadas

Sin embargo, XSP se diferencia de JSP en dos cosas:

1. XSP provee un marco de trabajo para mezclar código en cualquier lenguaje de programación con etiquetas XML. Las páginas XSP en Cocoon 2 se crean principalmente usando Java, pero son posibles otras implementaciones. Por ejemplo, el servidor de aplicaciones **XML AxKit** proporciona una implementación de XSP que usa Perl. En contraste JSP es exclusivamente una tecnología Java.
2. La diferencia más práctica es que XSP genera ***datos*** dinámicos, no ***presentaciones*** dinámicas. JSP por

el otro lado es tecnología de la capa de presentación usada para producir HTML o XML generalmente al final de la serie de pasos de procesamiento. Las páginas XSP generan datos para las tuberías de Cocoon 2, el cual crea la presentación deseada.

¿Qué describe un XSP?

Para responder esta pregunta, primero veamos el siguiente ejemplo:

```
<xsp:page language="java" xmlns:xsp="http://apache.org/xsp">
        <mensaje>Hola Mundo!</mensaje>
</xsp:page>
```

Cuando se procesa esta página con Cocoon 2, se genera el siguiente documento XML:

```
<mensaje>Hola Mundo!</mensaje>
```

Aunque este es un ejemplo simple, muestra que el documento original contiene una mezcla de dos lenguajes:

- *elementos XSP*, que son interpretados por Cocoon 2 y,
- *elementos de usuario* que describen el documento de salida deseado.

Cuando Cocoon 2 lee un documento XSP, internamente convierte el lenguaje XSP en código fuente para el generador de Cocoon 2. Esta fuente luego es compilada y después es ejecutada. Los dos tipos de lenguaje juegan diferentes roles en el proceso. Los elementos XSP se usan como pistas para definir la ***estructura*** del código fuente generado, mientras que los elementos de usuario, contenido y otra lógica de aplicación son usados para generar ***instrucciones*** para el generador que describen como debe este construir su salida.

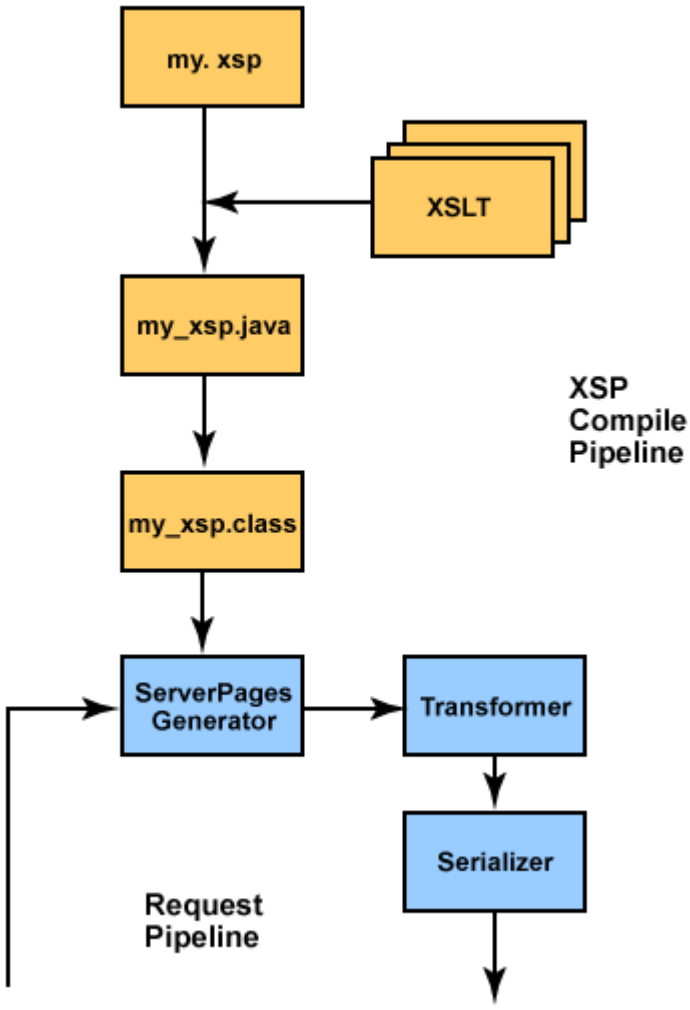
De lo anterior, se concluye que XSP sirve para generar automáticamente código de programa a partir de una descripción declarativa de lo que el código debe hacer. El ejemplo presentado declara que el código generado debe debe crear un elemento *mensaje* y darle el contenido de texto “Hola Mundo”. En la realidad, las páginas XSP no son siempre completamente declarativas porque es posible directamente incluir código de programa suplementario. Aunque el uso cuidadoso de *hojas lógicas* puede reducir o incluso eliminar la necesidad de hacer esto. Sobre las hojas lógicas se hablará mas adelante en este documento.

La forma en que Cocoon2 procesa las páginas XSP es análogo a la forma en que un contenedor JSP

procesa páginas JSP para crear el código de Servlets Java. El autor de la página no debe saber como escribir un Servlet, su trabajo es crear elementos HTML, agregar etiquetas personalizadas de una librería de etiquetas personalizadas y posiblemente agregar piezas de código Java. Es el trabajo del motor JSP usar la estructura de la página JSP para crear el servlet automáticamente, tal y como es el trabajo de Cocoon 2 el crear automáticamente un generador a partir de una página XSP.

Proceso de compilación XSP

El proceso de transformar una página XSP a un generador es de varios pasos, resumidos en el gráfico a continuación.



Una solicitud es recibida por Cocoon y este determina por la configuración del sitemap que los datos XML para esta solicitud deben ser creados a partir de una página XSP. Los datos son luego pasados a un transformador y después a un serializador que produce la respuesta final para el usuario.

Cuando es la primera vez que se recibe una solicitud para esta tubería, Cocoon debe compilar la página XSP requerida (una página XSP se recompilará si cambia la fuente original). Este trabajo lleva los siguientes pasos:

•Analizar el documento XSP
•Transformar la página XSP usando una hoja de estilo dedicada para generar el código fuente Java.
•Guardar el archivo Java después de haber completamente identificado y formateado la fuente usando un formateador de código.
•Compilar la fuente en un Generador.
•Cargar y ejecutar el Generador compilado con la tubería solicitada.

Es importante señalar que todo el proceso de compilación es completamente independiente de la solicitud de procesamiento de la tubería (indicado por colores diferentes en el diagrama).

Por lo tanto, los pasos de transformación usados para crear el código fuente del Generador no tiene acceso a la solicitud inicial. Y ya que la transformación no tiene acceso a los parámetros contenidos en la solicitud, su procesamiento no puede ser afectado por solicitudes individuales. Sin embargo, cuando el generador se ejecuta, tiene acceso al ambiente de la solicitud en la misma forma en que las páginas JSP tienen acceso a sus solicitudes, etc.

Usando páginas XSP

Cocoon 2 proporciona un componente llamado ServerPagesGenerator que coordina la compilación y ejecución de cada página XSP. Este generador se debe agregar a una tubería para disparar el procesamiento de una página XSP en particular.

Este componente se declara en el mapa de sitio de la siguiente manera:

```
<map:generators default="file">
        <map:generator  name="serverpages" src="org.apache.cocoon.generation.ServerPagesGenerator"/>
        <!-- ... otras declaraciones de Generadores ... -->
</map:generators>
```

Para usar este generador en una tubería, solo se debe indicar que debe ser usado (en lugar del implícito) e indicar donde puede encontrar la fuente de una página XSP en particular:

```
<map:pipeline>
        <map:match pattern="tutorial/*.xml">
                <map:generate type="serverpages" src="tutorial/{1}.xsp"/>
                <map:serialize type="xml"/>
        </map:match>
</map:pipeline>
```

La tubería match la solicitud para http://localhost:8080/cocoon/tutorial/holamundo.xml. El ServerPagesGenerator luego lee el archivo $CATALINA_HOME/webapps/cocoon/tutorial/holamundo.xsp y lo compila en un segundo generador que es ejecutado. La salida del generador luego es serializada al usuario.

El código fuente de XSP

Una vez que un XSP ha sido ejecutado, puede examinar el código que fue generado durante el proceso de compilación. Cuando se ejecuta Cocoon 2 con Tomcat, todo el código fuente de la páginas XSP se mantiene en: $CATALINA_HOME/work/localhost/<web-app-name>/cocoon-files/org/apache/cocoon/www/

La fuente de Hola Mundo se encontrarán en:
$CATALINA_HOME/work/localhost/cocoon/cocoon-files/org/apache/cocoon/www/tutorial/holamundo_xsp.java

Note que el nombre de archivo generado en Java se deriva del nombre del documento original XSP. (Por ejemplo: mi_archivo.xsp será mi_archivo_xsp.java).

Al examinar el código fuente se puede apreciar que consta de las siguientes características:

- La clase generada importa varias clases estándar Java y Cocoon 2
- La clase generada es subclase de XSPGenerator
- La clase tiene un método único generate() que ejecuta todo el procesamiento
- El método generado contiene llamadas a métodos API de SAX – startElement, characters, y métodos startElement – que describen el documento de salida deseado.

Es importante conocer donde se aloja el código fuente generado de una página XSP. Muchas veces los errores en el documento original XSP desembocan en errores de compilación en la clase Java generada. En este caso, Cocoon 2 genera una página de error que indica el número de línea del código con problema. Tener el código fuente a mano puede ayudar a resolver el problema.

Nota importante: No se recomienda cambiar el código fuente generado por Cocoon 2. Porque al cambiar de nuevo el XSP original, las fuentes generadas serán reescritas por la nueva versión y se perderán los cambios. Para insertar código extra donde se requiera, es mejor utilizar elementos XSP apropiados (Ejemplo: xsp:logic).

Sintaxis XSP

El elemento xsp:page

Es el elemento raíz de cada documento XSP. Debe tener un atributo de lenguaje que identifica el lenguaje de programación que contiene la página – específicamente "java". Otras implementaciones XSP pueden soportar otros lenguajes, pero Java es la mejor opción al usar Cocoon 2. Ejemplo:

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

  <!-- contenido de la página -->

</xsp:page>
```

Arriba, el elemento XSP declara el nombre de espacio http://apache.org/xsp, con el prefijo xsp. Todos los elementos XSP deben ser calificados con este prefijo de nombre de espacio. Para aclarar, a continuación los nombres de elementos serán referidos en este formato (por ejemplo: xsp:page, xsp:comment, etc).

El elemento de página (xsp:page) puede contener:

- Cualquier número de elementos xsp:structure
- Cualquier número de elementos xsp:logic
- **Solo** un elemento de usuario

La última restricción es muy importante "Elemento de usuario" significa cualquier elemento que no está en el nombre de espacio XSP, incluyendo elementos sin nombre de espacio. Esta restricción existe porque el elemento de usuario se convertirá en el elemento raíz del documento XML creado por la página XSP y por definición, un documento XML solo puede tener una sola raíz.

Los elementos xsp:structure y xsp:include

Cuando se agrega lógica de programa que usa API estándar o personalizado Java, se debe indicar en el XSP que se necesitan expresiones adicionales de tipo "*import*" en el código fuente generado para asegurar que la compilación sea exitosa.

Estos elementos se usan para proporcionar pistas adicionales al proceso de generación de código. Deben ser usados juntos con el elemento xsp:structure agrupando un número de elementos xsp:include. Cada elemento xsp:include define un paquete o clase Java adicional a importar. Por ejemplo:

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

        <xsp:structure>
                <xsp:include>java.util.Calendar</xsp:include>
                <xsp:include>java.text.*</xsp:include>
        </xsp:structure>
        <!-- contenido de página -->
</xsp:page>
```

Un elemento xsp:include incluye solo texto sin etiquetas adicionales. El contenido puede ser un paquete especifico o el nombre de una clase Java tal y como se utiliza en una expresión "*import*" de Java. Note que no se debe incluir la palabra clave ***import*** ni el punto y coma del final. El generador de código los agrega automáticamente. Incluirlos genera un error. El resultado del ejemplo anterior será:

```
import java.util.Calendar;
```

```
import java.text.*;
```

Una serie de importaciones implícitas se agregan durante el proceso de generación de código. Son clases expecificas de los paquetes Cocoon, Avalon y SAX como también algunas clases de utilidades del API estándar de Java.

El elemento xsp:logic

Se usa para agregar bloques de código Java.

Aunque estos elementos aparecen como hijos directos de xsp:page (fuera del elemento de usuario único) el bloque de código puede contener definiciones de métodos y/o variables miembros. Esto porque el código que aparece fuera del elemento de usuario no se incluye dentro del método del generador generate(). Según las reglas de sintaxis de Java dicen que este código debe ser variables miembros o de clase (por ejemplo: estáticos) o métodos.

A continuación se define un método que puede ser llamado desde cualquier parte del XSP y retorna la hora actual:

```
<xsp:page language="java"
                xmlns:xsp="http://apache.org.xsp”>
        <xsp:logic>
                public String getTime() {
                        return java.util.Calendar.getInstance().getTime().toString();
                }
        </xsp:logic>
        <document/>
</xsp:page>
```

Este elemento también puede ser usado en cualquier parte de la página XSP. En este caso debe contener expresiones Java que serán agregadas al método generate() del generador compilado. Esto hace al elemento xsp:logic similar a la sintaxis <% .... %> usada en las páginas JSP. A continuación un ejemplo de como se pueden crear bloques de código con este elemento:

```
<xsp:page language="java"
                xmlns:xsp="http://apache.org/xsp">
        <document>
                <xsp:logic>
                        SimpleDateFormat formato = new SimpleDateFormat("EEE, MMM d, yyyy");
                        String tiempo_Actual = formato.format(java.util.Calendar.getInstance().getTime());
                </xsp:logic>

                <time><xsp:expr>tiempo_Actual</xsp:expr></time>
```

```
            <!-- elementos adicionales -->
        </document>
</xsp:page>
```

Este código incluye lógica que guarda la hora y la fecha en que se evalúa el documento XSP. Este se agrega luego al documento dentro del elemento tiempo usando xsp:expr.

Evitar errores frecuentes

Una vez que el código se empieza a mezclar con XML, los problemas pueden surgir y evitar que el documento XSP se analice. Cualquier tipo de caracteres reservados XML (<, >, &) usados en el código del programa deben ser correctamente escapados usando una de las entidades predefinidas. Por ejempo:

```
if (a < b && c > d) { ... }
```

debe ser reescrito de esta forma si se agrega a un documento XSP:

```
if (a &lt; b &amp;&amp; c &gt;) { ... }
```

Una forma de evitar esto es usando una sección CDATA, que le dice al analizador XML que debe ignorar las reglas para esa sección de contenido:

```
<xsp:logic>
        <![CDATA[
            if (a < b && c > d) { ... }
        ]]&gt;
</xsp:logic>
```

Sin embargo, no se recomienda el uso de CDATA porque el analizador ignorará cualquier elemento XSP o de usuario que aparezca dentro de esta sección, lo tratará como texto plano en lugar de XML. Esto solo creará errores que consumen tiempo y son difíciles de hallar en lugar de hacer correctamente escapar el texto usando entidades.

La otra posible causa de error puede ser tratar de manipular las reglas XML con elementos xsp:logic:

```
<search-results>
        <xsp:logic>
            if (firstResult()) {
                <result id="first">
            }  else {
                <result>
            }
                 <!-- ...agregar aquí el código de generación resultante... -->
```

```
                    </result>
        </xsp:logic>
</search-results>
```

En este caso, el XSP produce una lista de resultados de búsqueda y trata el primer resultado como un caso especial porque se agrega un atributo. Desde el punto de vista del programador esto es correcto, pero para el analizador XML, rompe las reglas. Note que hay dos inicios de elemento result, pero solo una etiqueta final. Esto no está bien formado porque el analizador trata el código como texto plano. El proceso de compilación XSP no entiende código de programa o lo que este hace. Lo correcto es:

```
<search-results>
        <xsp:logic>
                if (firstResult()) {
                        <result id="first">
                                <!--... manipulación del primer resultado..-->
                        </result>
                } else {
                        <result>
                                <!--...manipulación de otros resultados...-->
                        </result>
                }
        </xsp:logic>
</search-results>
```

Entender la relación entre XML y la lógica de programa embebida en las páginas XSP es un importante paso hacia adelante para ser productivo en esta tecnología.

El elemento xsp:expr

Se usa para tomar una expresión cuyo valor debe ser agregado directamente en el documento de salida. En contraste, el elemento xsp:logic contiene código para el generador. El elemento xsp:expr es equivalente a la expresión <%= .... %> en JSP.

Es importante señalar que el elemento xsp:expr debe contener expresiones Java y no declaraciones. Ejemplo:

Expresiones	System.currentTimeMillis()	i++	new Date()	"username"
Declaraciones	System.currentTimeMillis();	i++;	Date d = new Date();	String s = "username";

Incluir una declaración como el valor de un elemento xsp:expr no será un error al momento de analizar, pero causará un error de compilación porque el contenido de un elemento de expresión es usado como un parámetro de método en el código fuente generado:

```
//Correcto
XSP:    <xsp:expr>"username"</xsp:expr>
Java:   XSPObjectHelper.xspExpr(contentHandler, "username");

//INCORRECTO!
XSP:    <xsp:expr>System.currentTimeMillis();/xsp:expr>
Java:   XSPObjectHelper.xspExpr(contentHandler, System.currentTimeMillis(););
```

Existen varios métodos sobrepuestos en la clase XSPObjectHelper que difieren en los parámetros que aceptan. Por ejemplo, hay métodos que aceptan los tipos primitivos Java, y hay otros que aceptan **String**, **Collection**, o **Object**. La expresión Java dada en el elemento xsp:expr debe retornar uno de estos tipos para que la compilación del código fuente generado sea exitosa.

Los elementos xsp:expr muchas veces se usan para referirse a variables o llamar métodos que están definidos en otro lado en un XSP dentro de un elemento xsp:logic. El siguiente ejemplo muestra como una variable en ciclo puede ser usada para crear contenido.

```
<elements>
        <xsp:logic>
                for (int i = 1; i < 11; i++) {
                        <element><xsp:expr>i</xsp:expr></element>
                }
        </xsp:logic>
</elements>
```

Un error que se comete frecuentemente con xsp:expr es tratar de agregar contenido de texto a un elemento. Los valores de tipo cadena siempre tienen que estar entre comillas, en otro caso serán interpretadas como referencias a variables durante la compilación. Esto luego resulta en un error (porque la variable no está definida) o peor aún funciona con éxito pero resulta con errores que son difíciles de encontrar!

Generando elementos dinámicos usando xsp:element

XSLT proporciona dos mecanismos para crear elementos como salida de una transformación. El primero, elementos resultado literales están presentes dentro de la hoja de estilo. El segundo, es mediante el uso de etiquetas xsl:element, que le permiten a un elemento ser creado dinámicamente durante una transformación. Por ejemplo, su nombre y/o atributos pueden ser calculados por la hoja de estilo.

XSP ofrece los mismos dos mecanismos. Elementos de usuario, agregados directamente a la página XSP son equivalentes a los elementos resultantes literales. Los elementos pueden ser también creados dinámicamente usando xsp:element que funciona de igual forma que su equivalente XSLT.

```
<xsp:element>
        <xsp:param name="name"><xsp:expr>"mi_Nombre_De_Elemento"</xsp:expr></xsp:param>
        Contenido del elemento
</xsp:element>
```

El ejemplo crea un elemento con un nombre creado dinámicamente que requiere el uso de los elementos xsp:element y xsp:param. Luego define un parámetro, en este caso el nombre del elemento cuyo valor es una expresión que es usada para calcular el nombre del elemento. El contenido se agrega al elemento en la forma corriente.

Los elementos creados de esta forma también pueden ser asociados a nombres de espacio especificos y prefijos. Ejemplo:

```
<xsp:element prefix="mi" uri="http://www.ejemplos.org">
        <xsp:param name="name"><xsp:expr>"mi_Nombre_De_Elemento"</xsp:expr></xsp:param>
        Contenido del elemento
</xsp:element>
```

Note que el parámetro nombre de espacio y el parámetro prefijo son requeridos. En caso contrario ocurrirá un error. El ejemplo generará la siguiente salida:

```
<mi:mi_Nombre_De_Elemento xmlns:mi="http://www.ejemplos.org">
        Contenido del elemento
</mi:mi_Nombre_De_Elemento>
```

También es posible generar dinámicamente el nombre de espacio URI y el prefijo. En lugar de agregar los atributos prefix y uri, use elementos adicionales xsp:param con los nombre apropiados. Este es el equivalente del ejemplo anterior:

```
<xsp:element>
        <xsp:param name="name"><xsp:expr>"mi_Nombre_De_Elemento"</xsp:expr></xsp:param>
        <xsp:param name="prefix">"mi"</xsp:param>
        <xsp:param name="uri">"http://www.ejemlos.org"</xsp:param>
        Contenido del Elemento
</xsp:element>
```

Generando atributos dinámicos usando xsp:attribute

Tal y como los elementos pueden ser creados dinámicamente en las páginas XSP, también se pueden crear dinámicamente los atributos. El elemento xsp:attribute trabaja en forma similar que xsp:element, permitiendo crear dinámicamente el nombre y el valor de un atributo:

```
<xsp:element>
        <xsp:param name="name"><xsp:expr>"mi_Nombre_De_Elemento"</xsp:expr></xsp:param>
        <xsp:attribute name="miAtributo">valor_De_mi_Atributo</xsp:attribute>
        Contenido del elemento
</xsp:element>
```

El nombre del atributo se define dentro del atributo name, al igual que en el caso de xsp:element, también puede ser definido usando un elemento hijo xsp:param. El valor del atributo se especifica como el contenido del elemento. Este puede ser un valor en texto simple o más útil, generado por un elemento xsp:expr.

La etiqueta xsp:attribute no solo debe ser usada con xsp:element. Puede ser colocada dentro de cualquier elemento de usuario y un atributo es agregado de la misma forma. Por ejemplo, el URL para un elemento imagen puede ser dinámicamente creada usando una expresión que llama un método definido en otro lado de la página XSP:

```
<image>
  <xsp:attribute name="href"><xsp:expr>calcularURLDeImagen()</xsp:expr></xsp:attribute>
</image>
```

Si el atributo generado se asocia a un nombre de espacio especifico, esto debe ser indicado usando atributos adicionales prefix y uri o elementos xsp:param, similar al método usado con xsp:element. De nuevo, es un error solo definir uno de ellos.

Creando comentarios e instrucciones de procesamiento

Los elementos xsp:comment y xsp:pi son usados para crear comentarios e instrucciones de procesamiento.

Crear comentario es fácil. El texto hijo del elemento xsp:comment se convierte en un comentario XML:

```
<xsp:comment>Esto es un comentario</xsp:comment>
```

Se convierte en:

```
<!-- Esto es un comentario -->
```

Note que cualquier otra etiqueta anidada es ignorada, pero cualquier texto será agregado al comentario.

Crear instrucciones de procesamiento es similar a crear elementos o atributos dinámicos. El elemento xsp:pi puede tener parámetros anidados que identifican el objetivo de la instrucción de procesamiento. El resto del contenido del elemento xsp:pi es evaluado como siempre. Un ejemplo:

```
<xsp:pi target="myApplication">
```

```
        <xsp:expr>"param1=value, param2=value, generatorTimestamp=" +  System.currentTimeMillis()</xsp:expr>
</xsp:pi>
```

La salida es:

```
<?myApplication param1=value, param2=value, generatorTimestamp=1017407796870?>
```

El objetivo de la instrucción de procesamiento puede también ser generada automáticamente creándola dentro de un elemento xsp:param, como por ejemplo:

```
<xsp:pi>
        <xsp:param name="target"><xsp:expr>"myApplication"</xsp:expr></xsp:param>
        <xsp:expr>"param1=value, param2=value, generatorTimestamp=" +  System.currentTimeMillis()</xsp:expr>
</xsp:pi>
```

Resumen de la sintaxis

Los bloques presentados son los bloques de construcción básicos de un XSP. Es posible crear contenido dinámico XML solo con estos bloques. Existe un aspecto adicional de los XSP que es muy importante y permite que esta sintaxis se amplíe con elementos personalizados. Este nuevo concepto se conoce como hojas lógicas que veremos a continuación:

- **xsp:page:** es el elemento raiz de un documento XSP. Solo puede contener un solo elemento de usuario.
- **xsp:structure** y **xsp:include:** permite importar clases Java adicionales a la versión compilada del XSP
- **xsp:logic:** permite bloques de código adicional a ser incluido en la versión compilada del XSP. Puede incluir variables miembros, métodos o lógica de aplicación.
- **xsp:expr:** permite expresiones Java a ser evaluadas. Su valor se incluye en el documento.
- **xsp:element:** permite crear elementos dinámicamente por el XSP, los elementos pueden ser creados con cualquier nombre y pueden ser asociados a un nombre de espacio.
- **xsp:attribute:** permite agregar dinámicamente atributos a los elementos, los atributos pueden ser creados con cualquier nombre y pueden ser asociados a un nombre de espacio
- **xsp:comment:** permite agregar comentarios al documento creado.
- **xsp:pi:** permite crear dinámicamente instrucciones de procesamiento y ser agregadas al documento.

¿Qué son las hojas lógicas?

Aunque la habilidad para generar contenido XML con XSP es una función importante, hay algunas desventajas al mezclar código con etiquetas XML. Como se ha dicho anteriormente, la mezcla de diferentes reglas de sintaxis pueden ocasionalmente crear problemas que no se ven a primera vista. El mayor problema de esta técnica es que al colocar grandes cantidades de lógica de aplicación en una página

XSP hace que el código y el documento sean difíciles de mantener.

Estas son dificultades que enfrenta cualquier tecnología que mezcla código y etiquetas de esta forma. En JSP estos problemas han sido aliviados introduciendo el concepto de librerías de etiquetas, que permiten al desarrollador Web agregar etiquetas personalizadas a las páginas JSP. El contenedor JSP asocia esas etiquetas con la implementación creada por un programador Java. Esto provee una buena separación del rol de desarrollo y el código.

Cocoon 2 ofrece una tecnología que es similar a las librerías de etiquetas JSP – ***hojas lógicas***. Las hojas lógicas son librerías de elementos personalizados que pueden ser agregados a las páginas XSP. A diferencia de las librerías de etiquetas JSP que son creadas como clases Java, las hojas lógicas de Cocoon 2 son implementadas usando transformaciones XSLT. Estas transformaciones introducen bloques de código adicional y/o etiquetas XSP dentro de un documento XSP y con esto extender las capacidades de la clase Generadora resultante.

Una de las principales ventajas de las hojas lógicas es que los documentos originales son muchos más claros porque el código ha sido eliminado y porque las mismas hojas lógicas pueden ser reusadas a través de múltiples documentos XSP. Esto evita la necesidad de tener que duplicar código en varios lugares.

Usando Hojas Lógicas

Cada hoja lógica tiene su propio nombre de espacio. Para usar una hoja lógica solo se debe declarar el nombre de espacio apropiado en el documento XSP y luego agregar elementos de ese nombre de espacio.

Por ejemplo Cocoon 2 incluye una hoja lógica de utilidades que puede agregar el tiempo actual al documento. A continuación una página XSP que usa esta función:

```
<xsp:page language="java"
                    xmlns:xsp="http://apache.org/xsp"
                    xmlns:util="http://apache.org/xsp/util/2.0">
          <reloj>
                    <día><util:time format="EE"/></día>
                    <mes><util:time format="MMMM"/></mes>
                    <año><util:time format="yyyy"/></año>
                    <hora><util:time format="HH:mm:ss 'de' dd/MM/yyyy"/></hora>
          </reloj>
</xsp:page>
```

Note que la página declara el nombre de espacio de la utilidad, http://apache.org/xsp/util/2.0 y usa un solo

elemento adicional sin insertar código Java. Cuando evaluamos esta página, produce:

```
<reloj>
        <día>Vie</día>
        <mes>Marzo</mes>
        <año>2001</año>
        <hora>15:14:27 de 29/03/2002</hora>
</reloj>
```

Los elementos del nombre de espacio util no aparecen en el documento de salida, en su lugar, se usan en el proceso de compilación XSP para crear código que genera fechas y horas en una serie de formatos.

Otras hojas lógica pueden ser usadas exactamente de la misma forma. Un documento XSP puede hacer uso de elementos de diferentes hojas lógicas, permitiendo crear construcciones más complejas a partir de funcionalidad reusable.

Hojas Lógicas integradas

Cocoon 2 proporciona una serie de hojas lógicas predefinidas que ofrecen una gran variedad de funciones sin necesidad de escribir código Java. Esta hojas lógicas predefinidas pueden ser clasificadas por el tipo de funciones que proporcionan:

- **Hojas Lógicas de Ambiente** - dan acceso al ambiente de procesamiento de Cocoon (ejemplo: la solicitud y la respuesta)
- **Hojas Lógicas de utilidades** – código de utilidad multipropósito (ejemplo: incluir archivos, enviar correo electrónico, etc)
- **Manipulación de datos** – acceso a validación de datos y funcionalidad relacionada a bases de datos.

Hojas Lógicas de Ambiente

Existen 4 hojas lógicas bajo esta categoría, cada una de ellas provee acceso a aspectos particulares del ambiente de procesamiento asociado con la solicitud Web.

La hoja lógica ***request*** proporciona acceso a propiedades de la solicitud, incluyendo los parámetros de la solicitud, los métodos de solicitud (ejemplo: GET, POST, etc) y los encabezados de solicitud. Esta hoja es útil cuando aspectos de los parámetros de solicitud son usados para alterar la generación del documento de salida.

La hoja lógica ***response*** provee acceso limitado a la respuesta HTTP asociado con la solicitud actual; solo

proporciona acceso al encabezado de respuesta. Un documento XSP no puede ejecutar un *include* o *forward* en la misma forma que un Servlet Java o página JSP debido a la separación del contenido que es una parte del núcleo arquitectónico de Cocoon 2. Esta funcionalidad se describe en el sitemap; una página XSP genera contenido XML y no hace procesamiento directo.

La hoja lógica ***session*** provee acceso al a información de sesión HTTP, incluyendo la habilidad de crear y borrar sesiones, como también crear y borrar atributos de las sesiones. Esta funcionalidad se usa mas en aplicaciones Web porque deben mantener una sesión de usuario por contexto. El administrador de sesiones de Cocoon 2 es completamente el equivalente al de JSP.

La hoja lógica ***cookie*** provee funcionalidad de mantenimiento de cookies, tales como agregar y borrar cookies, permitiendo guardar las preferencias en el navegador del usuario.

La funcionalidad de estas hojas lógicas es análoga a la que proveen los objetos implícitamente asociados con las páginas JSP (por ejemplo: los objetos request y response) y es tomada directamente del API Servlet HTTP.

Hojas Lógicas de utilidades

Hay tres hojas lógicas en esta categoría, cada una de las cuales provee funciones simples de utilidad.

La hoja lógica ***utility*** presentada anteriormente, además de permitir tomar la hora actual en múltiples formatos, le permite al autor XSP incorporar contenido XML directamente en los documentos XSP desde un URL o un archivo. Este contenido puede ser agregado como XML (ejemplo: elementos y atributos) o como texto plano.

La hoja lógica ***log*** permite escribir mensajes de registro desde un generador XSP mientras este procesa la solicitud del usuario – una función muy útil cuando necesitamos depurar páginas XSP complejas. Ejemplo, si suponemos que el nombre de espacio del registro (http://apache.org/xsp/log/2.0) ha sido definido, la siguiente linea escribe un mensaje de depuración en el archivo de registro implícito de Cocoon 2:

```
<log:debug>Este es un mensaje de depuración desde el generador XSP</log:debug>
```

Otro elementos como: info, warn, error y fatal-error, permiten escribir otro tipo de expresiones de registro.

La hoja lógica ***sendmail*** tiene un único elemento muy útil que provee acceso al API JavaMail desde una página XSP. El ejemplo asume que el nombre de espacio http://apache.org/cocoon/sendmail/1.0 se ha definido:

```
<sendmail:send-mail from="micorreo@correo.com" to="usuario@usuario.com" smtphost="servidor_SMTP@correo.com">
```

```
        <xsp:param name="subject"><xsp:expr>"El asunto de este correo..."</xsp:expr></xsp:param>
        <xsp:param name="body"><xsp:expr>"El cuerpo de este correo..."</xsp:expr></xsp:param>
</sendmail:send-mail>
```

Obviamente los atributos se deben alterar para reflejar las direcciones de correo from y to, servidor SMTP, etc. El asunto y el cuerpo del correo puede ser generado dinámicamente si es necesario. Si la dirección de correo debe ser creada dinámicamente (por ejemplo: leerla desde un parámetro de la sesión) entonces se puede usar un elemento xsp:param en lugar del atributo. El nombre del atributo de este parámetro debe reflejar la variable que describe.

Manipulación de Datos

Se conforma de dos hojas lógicas. Una de ellas provee mucho mas funcionalidad que la otra.

La hoja lógica ***form validator*** nunca se usa sola. Provee un interfaz puro a la acción FormValidator de Cocoon 2. Esta acción es capaz de ejecutar operaciones básicas de validación de los datos enviados a una aplicación Cocoon 2 a partir de un formulario HTML, incluyendo el control de valor mínimo y máximo de un entero, controlar el tamaño de una cadena y controlar que algunos parámetros son requeridos. Una pieza importante de funcionalidad es la habilidad de controlar cuando una variable enviada cumple una expresión regular.

Los resultados de la validación son guardados en un parámetro de solicitud. La hoja lógica *validator* proporciona la forma de interpretar estos resultados desde dentro de una página XSP. Esto permite generar dinámicamente mensajes de error para el usuario.

Luego de confirmar que los datos son válidos, el siguiente paso es almacenarlos en una base de datos. La hoja de estilo ***esql*** permite hacer esto y más, incluyendo búscar, borrar y actualizar una base de datos. En esencia permite embeber expresiones SQL directamente dentro del documento XSP. La hoja lógica luego genera el código JDBC apropiado para ejecutar la operación SQL lo que simplifica la manipulación y la solicitud de datos de la base de datos usando Cocoon.

Como trabajan las Hojas Lógicas

Algunos hechos importantes acerca las hojas lógicas ya se han explicado. Primero, las hojas lógicas se asocian a un determinado nombre de espacio y los elementos de este nombre de espacio están asociados con funcionalidad especifica – por ejemplo: al generar la hora o escribiendo un mensaje en la bitácora. Segundo, las hojas lógicas se implementan usando transformaciones XSLT, que agregan código Java a un documento XSP para crear la funcionalidad deseada.

Otra pieza del rompecabezas es el proceso de transformación del documento XSP a código Java, que provee ganchos para agregar pasos de transformación adicional antes de crear el código Java. En estos ganchos es donde las hojas lógicas juegan su papel.

Cuando Cocoon 2 procesa por primera vez un documento XSP, identifica todos los nombre de espacio del documento y los controla contra el archivo de configuración. Esta configuración asocia los nombres de espacio contra transformaciones (por ejemplo: una hoja lógica). Si se encuentra un match, entonces se ejecuta la siguiente transformación. Cocoon 2 hace este paso varias veces hasta que el único nombre de espacio que queda en el documento son los de la sintaxis XSP o los que no han sido mapeados a una transformación. En este punto Cocoon 2 ejecuta la última transformación para generar el código Java final.

Es por esto que las hojas lógicas esencialmente permiten que secciones de código se desglosen fuera del documento XSP y se guarden en una transformación XSLT separada. El código que se extrae del documento XSP se reemplaza por un elemento o secuencia de elementos de un nombre de espacio definido especialmente. Cuando Cocoon 2 procesa el documento XSP, reconoce que este nombre de espacio está asociado con una hoja lógica y ejecuta la transformación de la hoja lógica. Es trabajo de la hoja lógica insertar de vuelta el código original en el documento mientras preserva el resto del documento.

Escribiendo Hojas Lógicas

Es simple. Lo mas importante que se debe recordar es que toda hoja lógica debe preservar en el documento todo lo que no entienda, porque las transformaciones de las hojas lógicas se pueden ejecutar en cualquier orden,. En otras palabras, la plantilla básica de una hoja lógica es una *transformación de identidades*. Esta es una transformación que solo copia la entrada a la salida sin cambios. Un ejemplo de la *transformación de identidades* se proporciona a continuación como un punto de inicio:

```
<!-- Transformación de Identidades para ser usada como plantilla para escribir hojas lógicas XSP -->
<xsl:stylesheet
        version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:xsp="http://apache.org/xsp">
<!-- Descomentar esta sección de esta plantilla para agregar imports adicionales y/o lógica de clases -->
<xsl:template match="xsp:page">
        <xsp:page>
                <xsl:apply-templates select="@*"/>
                        <!-- Agregue imports Java adicionales usando elementos “include” adicionales.
                                <xsp:structure>
                                        <xsp:include></xsp:include>
                                </xsp:structure>
```

```
                        -->
                        <!-- Agregue métodos personales y/o variables miembros dentro de este elemento lógico
                                <xsp:logic/>
                        -->
                        <!-- Procesa el resto del documento -->
                <xsl:apply-templates />
        </xsp:page>
</xsl:template>
<!-- copie todo sin cambiarlo -->
<xsl:template match="node()|@*">
        <xsl:copy>
                <xsl:apply-templates select="@*"/>
                <xsl:apply-templates/>
        </xsl:copy>
 </xsl:template>
</xsl:stylesheet>
```

Esta hoja de estilo básica se puede ampliar, agregando nuevas plantillas (*templates*) para las cuales esta hoja lógica proporciona funcionalidad. El cuerpo de las plantillas puede incluir sintaxis XSP que describe la funcionalidad del elemento dado.

Como ejemplo, se creará una hoja lógica que implemente la misma función del elemento util:time:

```
<xsl:stylesheet
        version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:xsp="http://apache.org/xsp"
        xmlns:time=”http://www.ldodds.com/ns/time”>

<!-- Muestra de hoja lógica que implementa el elemento “time” -->
<xsl:template match="xsp:page">
        <xsp:page>
                <xsl:apply-templates select="@*"/>
                        <xsp:structure>
                                <xsp:include>java.util.Calendar</xsp:include>
                                <xsp:include>java.text.*</xsp:include>
                        </xsp:structure>
                        <!-- Agregue métodos personales y/o variables miembros dentro de este elemento lógico
                                <xsp:logic/>
                        -->
                        <!-- Procesa el resto del documento -->
                <xsl:apply-templates />
        </xsp:page>
</xsl:template>
```

```
<xsl:template match="time:time">
        <xsp:logic>
                SimpleDateFormat timeFormat = new SimpleDateFormat("<xsl:value-of select="@format"/>");
        </xsp:logic>
        <xsp:expr>timeFormat.format(java.util.Calendar.getInstance().getTime())</xsp:expr>
</xsl:template>

<!-- copie todo sin cambios -->
<xsl:template match="node()|@*">
        <xsl:copy>
                <xsl:apply-templates select="@*"/>
                <xsl:apply-templates/>
        </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

El elemento importante de esta hoja lógica es <xsl:template match="time:time">. Este elemento toma los elementos time y los reemplaza con el código de los elementos xsp:logic y xsp:expr para crear la fecha. El formato de fecha requerido lo define el usuario mediante el atributo format de este elemento.

Configurando una Hoja Lógica

Cocoon 2 mantiene el mapa entre los nombre de espacio y las hojas lógicas asociadas (hojas de estilo) en su archivo de configuración XML que se encuentra en $CATALINA_HOME/webapps/cocoon/cocoon.xconf.

Busque el elemento xsp-language en cocoon.xconf. Este elemento contiene elementos hijos target-language. Dentro de este elemento se deben declarar todas las hojas lógicas. Observe que las hojas lógicas integradas ya están definidas. Por ejemplo la entrada de la hoja lógica util:

```
<builtin-logicsheet>
        <parameter name=”prefix” value=”util”/>
        <parameter name=uri” value=”http://apache.org/xsp/util/2.0”/>
        <parameter name=”href” value=”resource://org/apache/cocoon/components/language/markup/xsp/java/util.xsl”/>
</builtin-logicsheet>
```

El elemento builtin-logicsheet tiene varios parámetros.

El parámetro prefix define el prefijo del nombre de espacio que es comúnmente asociado a esta o es recomendado para su uso con esta hoja lógica.

El segundo parámetro uri identifica el URI del nombre de espacio para esta hoja lógica. Teniendo

identificado el nombre de espacio asociado con esta hoja lógica.

El parámetro href identifica el lugar donde la hoja lógica puede ser encontrada. Esta hoja lógica será usada cuando se encuentre cualquier elemento del nombre de espacio indicado en una página XSP.

Mientras que las hojas lógicas integradas se colocan dentro de un archivo .jar junto a las clases de Cocoon, las hojas lógicas definidas por el usuario pueden ser almacenadas en el sistema de archivos e identificados por el URL resource:// apropiado. El URL de recursos indica archivos que están en el directorio de Cocoon WEB-INF/classes. Así que una vez que creamos la hoja lógica de ejemplo, llamada tiempo.xsl, una forma de almacenarla y hacer referencia a ella es:

$CATALINA_HOME/webapps/cocoon/WEB-INF/classes/logicsheets/tiempo.xsl

Luego el parámetro href se debe configurar de esta forma:

<parameter name=”href” value=”resource://logicsheets/tiempo.xsl”/>

Una vez que se han hecho los cambios al archivo de configuración, debe reiniciar Tomcat para que los cambios tengan efecto.

Consejos para el desarrollo de hojas lógicas

Tres consejos útiles y guías de diseño a tener en cuenta cuando se crean nuevas hojas lógicas:

Use clases de ayuda: Aunque crear hojas lógica colocará el código de programa fuera de sus página XSP, este código está aún amarrado en otro lado en una hoja de estilo XSLT. Esto hace el código difícil de mantener y probarlo de forma independiente fuera de Cocoon 2. Un buen consejo de diseño es colocar la mayor parte posible del código en clases de ayuda. Esto reduce la implementación de las hojas de estilo a solo recolectar parámetros y luego invocar los correspondientes métodos de ayuda. Las hojas lógicas integradas en Cocoon siguen este consejo y tienen una clase de ayuda por hoja lógica.

Cree hojas lógicas “macro”: Una macro es una forma sencilla de grabar y reproducir una serie de operaciones. La páginas XSP en una aplicación en particular pueden crear muchas veces las mismas estructuras o invocar otras hojas lógicas de la misma forma. Es útil unir este tipo de bloques de código común en las páginas XSP y crear elementos de mayor nivel (por ejemplo: macros) que describen estas estructuras repetidas. Ya que Cocoon 2 aplica recursivamente todas la hojas lógicas hasta que todos los nombres de espacio se hallan manejado, la implementación de una hoja lógica puede agregar elementos de otra. Este tipo de agrupamiento puede ayudar a reducir la complejidad de las páginas XSP.

Piense en el usuario: Cuando piense en que etiquetas proporcionar en la hoja lógica, piense en el usuario final. ¿Qué flexibilidad se requiere? Por ejemplo, ¿Qué parámetros se deben pasar a esta etiqueta? ¿Se pueden pasar los parámetros solo como atributos o anidados en elementos xsp:param? Este último es mas flexible porque permite que el usuario cree valores dinámicos usando xsp:expr, que no puede ser usado dentro de un atributo. También piense en la forma de nombrar la etiquetas. Deles nombres claros y fáciles de recordar y que sea claro que es lo que hacen. A nadie le gusta examinar la implementación de una hoja lógica para entender que es lo que hace. Una extensión obvia de este consejo es crear documentación de ayuda a la hoja lógica, esto le permitirá al usuario saber rápidamente que es lo que la etiqueta hace y como debe ser usada.

Hoja Lógica Session

Contiene todas las operaciones estándar de sesiones. Proporciona un interfaz XML a la mayoría de los métodos del objeto de Sesiones HttpSession.

Cada cliente obtiene su propia sesión única, que se crea la primera vez que ese accede a la página que crea o solicita una sesión. La sesión se identifica por su identificador único, el cual es generado por el servidor de servlets. Se traslada desde y hacia el cliente como un cookie o como un parámetro en la cadena de solicitud.

La hoja lógica de sesiones puede ser usada para obtener información acerca de la sesión misma, tal como el tiempo de creación y puede ser usada para almacenar o solicitar información en la sesión, tal como el nombre del usuario. La sesión normalmente es invalida después de algún tiempo, liberando la información almacenada. Es posible saber cuando una sesión es nueva y cuando y cuanto tiempo permanecerá valida entre las solicitudes del cliente. También es posible configurar cuanto tiempo la sesión permanecerá válida.

Uso

Como una hoja lógica XSP, la hoja lógica de sesión solo puede ser usada en páginas XSP. Para usarla se debe primero declarar el nombre de espacio de la sesión, mapeandola al URI http://apache.org/xsp/session/2.0. Además para asegurarse que tiene una sesión con la cual se puede trabajar se debe configurar el atributo create-session en el elemento xsp:page a verdadero. Este tomará la sesión existente o creará una nueva si la actual no es válida o no existe. Estos pasos resultan en un código como este:

```
<xsp:page
        xmlns:xsp="http://apache.org/xsp"
        xmlns:xsp-session="http://apache.org/xsp/session/2.0"
        create-session="true">
        ...
</xsp:page>
```

Luego es posible usar cualquiera de los elementos del nombre de espacio sesión.

Ejemplo

Este código almacena un valor en la sesión bajo el nombre “fruta”, luego lo toma a la salida. También toma el tiempo de creación de la sesión como una cadena. Claro que en lugar de presentar los valores tomados, es posible almacenarlos en elementos y procesarlos luego, por ejemplo mediante un hoja de estilo XSLT.

```
<?xml version="1.0"?>
<xsp:page
```

```
        xmlns:xsp="http://apache.org/xsp"
        xmlns:xsp-session="http://apache.org/xsp/session/2.0"
        create-session="true">
<html>
        <xsp-session:set-attribute name="fruta">Manzana</xsp-session:set-attribute>
        <b>Tu friuta favorita es:</b> <xsp-session:get-attribute name="fruta"/>
        <br/>
        <b>Tu sesión se creó el:</b> <xsp-session:get-creation-time as="string"/>
</html>
</xsp:page>
```

La salida de la página es mas o menos:

Tu fruta favorita es: Mango
Tu sesión se creó el: Wed Jun 13 17:42:44 EDT 2001

Interacción XSP

Las etiquetas de la hoja lógica Session pueden ser usadas intercaladas con código XSP que usa en forma directa el objeto session. Este objeto es una instancia de la clase HttpSession y está disponible dentro del elemento user de la página XSP, si el atributo create-session del elemento xsp:page a sido configurado a verdadero. La misma hoja lógica Session usa este objeto. Asi que ambas piezas de código hacen lo mismo:

Usando la hoja lógica Sesion:
Using the Session logicsheet:

```
<xsp:page
        xmlns:xsp="http://apache.org/xsp"
        xmlns:xsp-session="http://apache.org/xsp/session/2.0"
        create-session="true">
<page>
        <xsp-session:set-attribute name="fruta">Manzana</xsp-session:set-attribute>
        <fruta><xsp-session:get-attribute name="fruta"/></fruta>
</page>
</xsp:page>
```

Usando el objeto Session:

```
<xsp:page
        xmlns:xsp="http://apache.org/xsp"
        xmlns:xsp-session="http://apache.org/xsp/session/2.0"
        create-session="true">
<page>
        session.setAttribute("fruta", "Manzana");
```

```
        <fruta><xsp:expr>session.getAttribute("fruta")</xsp:expr></fruta>
</page>
</xsp:page>
```

Es posible mezclar libremente los elementos de Session con otro código Java XSP:

```
<xsp:logic>
        String fruit = <xsp-session:get-attribute name="fruit"/>;
        if (fruta != null) {
                fruta = fruta.toUpperCase();
        }
</xsp:logic>
<fruta><xsp:expr>fruta</xsp:expr></fruta>
```

Referencia de Elementos

Todos los elementos Session que requieren o brindan información adicional permiten proporcionar esta información como atributos o elementos hijos. Por ejemplo, los códigos a continuación son equivalentes:

Código A: <xsp-session:get-attribute name="fruta"/>
Código B: <xsp-session:get-attribute><xsp-session:name>fruta</xsp-session:name></xsp-session:get-attribute>

Todos los elementos de una sesión que toman datos de una sesión pueden darle salida a los datos en dos formas. El atributo del elemento as se usa para alternar entre las diferentes opciones de salida. La elección siempre está entre algún valor implícito para as y el valor del nodo. Usando el valor implícito para as (que es más fácil hacerlo dejando el atributo ya existente), los elementos session colocarán los resultados de su operación en un nodo <xsp:expr>. Esto permite usar el resultado en una expresión Java o convertirla en texto en el árbol de salida DOM. Sin embargo si usamos as=”node” la salida será embebida en un nodo o nodos como sea apropiado. Por ejemplo, el código a continuación

```
<xsp-session:get-attribute as="node" name="fruta"/>
```

Tendrá una salida:

```
<xsp-session:attribute name="fruta">manzana</xsp-session:attribute>
```

Esto es muy útil en los elementos que retornan múltiples piezas de información, tales como xsp-session:get-attribute-names. Sin usar as=”node”, los valores retornados se escribirán de punta a punta sin separación. Si la solicitud de salida del nodo se solicita, cada valor será escrito en un nodo separado, que luego pueden ser referidos separadamente.

Los elementos que proporcionan salida de nodos están marcados con “Si” en la columna “Nodo?” de la

tabla de abajo. A diferencia de los otros atributos usados en los elementos session, as no puede ser usado como un elemento hijo, siempre debe ser proporcionado como atributo se se usa.

Nota: Porque estos elementos tiene su equivalente en métodos HttpSession, la documentación de Java Servlet API, versión 2.2 es muy útil para entender estos elementos y su uso.

Nombre del Elemento	*Atributos / Elementos Hijos*	*Nodo?*	*Descripción*
xsp-session:get-attribute	name	Sí	Toma el valor del atributo nombrado que está almacenado en la sesión.
xsp-session:get-attribute-names		Sí	Toma los nombres de todos los atributos de la sesión.
xsp-session:get-creation-time		Sí	Toma el tiempo en que la sesión se creó. El atributo as de esta sesión puede ser long (implícito), string o nodo. Si es long, el valor de retorno es de tipo Java long que representa un valor Java date. Si es String, el valor se convierte a una presentación tipo String de la fecha. Ejemplo: “Sat Aug 31 05:38:10 CST”. Si es nodo, el valor long se coloca en el nodo de salida.
xsp-session:get-id		Sí	Toma el número de sesión, que generalmente es una cadena generada aleatoriamente por el servidor.
xsp-session:get-last-accessed-time		Sí	Toma el tiempo en que se accedió a esta sesión por última vez (la última vez que está página fue solicitada usando el mismo id de sesión. El atributo as trabaja igual que en xsp-session:get-creation-time
xsp-session:get-max-inactive-interval		Sí	Toma el tiempo mínimo en [s] que el servidor mantendrá esta sesión entre una solicitud y otra.
xsp-session:invalidate		No	Invalida la sesión actual y destruye los atributos almacenados en la sesión.
xsp-session:is-new		Sí	Indica si la sesión se acaba de crear.
xsp-session:remove-attribute	name	no	Elimina de la sesión el atributo nombrado.

Nombre del Elemento	*Atributos / Elementos Hijos*	*Nodo?*	*Descripción*
xsp-session:set-attribute	name	no	Guarda en la sesión el atributo nombrado. El valor a ser almacenado es el contenido del elemento. Por ejemplo: <xsp-session:set-attribute name="fruta">manzana</xsp-session:set-attribute>
xsp-session:set-max-inactive-interval	interval	no	Configura el tiempo mínimo en [s] que el servidor mantendrá esta sesión entre una solicitud y otra.

Marco de Autenticación

El punto central para construir una aplicación web es la autenticación. Aquí se presenta como utilizar los componentes que Cocoon ofrece para cubrir este importante requerimiento. El paquete de autenticación de Cocoon es un módulo flexible para autenticar y administrar usuarios. Un usuario puede ser legitimado usando cualquier información disponible de cualquier fuente, por ejemplo: una base de datos o LDAP. Con este mecanismo es muy fácil utilizar cualquier sistema de administración de usuarios o autenticación disponible.

Componentes del Sitemap

El marco de autenticación agrega algunas acciones al sitemap:

- auth-protect
- auth-login
- auth-logout
- auth-loggedIn

El *authentication manager* toma la configuración del marco de autenticación y las acciones controlan las tuberías. Las acciones *auth-login* y *auth-logout* controlan la autenticación y la acción *auth-loggedIn* controla el flujo de la aplicación.

Autenticación de Usuarios

Una de las funciones del marco es la autenticación de usuarios. Un documento puede ser accesible por todos o puede ser protegido usando este marco. El proceso de solicitud de un documento se describe a continuación:

1. El usuario solicita un documento
2. El marco verifica si el documento está protegido. Si no está protegido, como respuesta se enviará el documento solicitado.
3. Si el documento está protegido. El marco verifica si el usuario esta autorizado para ver este documento.
4. Si el usuario está autenticado, la respuesta es el documento original. Si no el marco lo redirije a un documento especial (redirect-to). Este documento es configurable libremente y puede contener por ejemplo información de acceso no autorizado y un formulario de ingreso al sistema.
5. Usando el formulario de ingreso al sistema se puede llamar a un recurso de autenticación con la información del usuario (nombre y contraseña). Este recurso de autenticación utiliza el marco para el proceso de autenticación.

6. En el caso que se autenticó exitosamente al usuario, el puede redirijirlo al documento original (o a cualquier documento de inicio)
7. Si la autenticación falla, el marco llama a otro documento que presenta información al usuario.

Este proceso es solo un ejemplo de la forma de uso del marco. Puede ser configurado para cualquier esquema de autenticación. Todos los recursos se pueden configurar libremente.

El manipulador de autenticación

El objeto básico para la autenticación es el manipulador de autenticación. Este controla el acceso a los recursos. Cada recurso puede ser relacionado exactamente a un manipulador de autenticación. Todos los recursos relacionados al mismo manipulador están protegidos de la misma forma. Si un usuario tiene acceso a un manipulador, el usuario tiene los mismo derechos de acceso para todos los recursos del mismo manejador.

Cada manipulador de autenticación debe tener:

- Un nombre único
- Un recurso de autenticación que ejecuta la autenticación real.
- Un documento de redirección hacia el cual el marco lo redirige en caso de acceso no autorizado.

Mediante el uso de nombre únicos, el marco puede manejar diferentes manipuladores. Así diferentes partes del sitemap pueden estar protegidos en forma diferente. Un documento puede estar protegido al llamar a su manejador utilizando la acción *auth-protect*. Esta acción debe incluirse dentro de la tubería del recurso. Se entrega la información del manipulador como parámetro:

```
<map:match pattern="recursoprotegido">
        <map:act type="auth-protect">
                <map:parameter name="handler" value="nombre_unico_de_manipulador"/>
                <map:generate src="fuente/recurso.xml"/>
        </map:act>
        ...
</map:match>
```

Si la tubería no usa la acción *auth-protect* o falta el parámetro „handler“, el documento será accesible para todos los usuarios.

El documento redirect-to

Si el documento solicitado no es accesible por el usuario, el marco lo redirijirá a un documento especial llamado redirect-to. Este documento se debe incluir en la configuración del manipulador de autenticación:

```
<autentication-manager>
  <handlers>
    <!-- aquí está la configuración del manipulador -->
    <handler name="unico">
      <redirect-to uri="cocoon://paginaLogin"/> <!-- Recurso de Login -->
    </handler>
  </handlers>
</autentication-manager>
```

Este documento redirect-to es un recurso no protegido en el sitemap. Para dar seguimiento al documento que fue solicitado, el documento redirect-to recibe un parámetro „**resource**“ con el valor. Adicionalmente todos los parámetros especificados dentro de la etiqueta „**redirect-to**“ dentro de la configuración del manipulador se envían al documento.

El documento **redirect-to** puede contener un formulario para autenticar usuarios. Este formulario debe llamar al verdadero documento de login que se describe a continuación.

El proceso de autenticación es hecho por la acción *auth-login*. El recurso de login contiene esta acción:

```
<map:match pattern="login">
        <map:act type="auth-login">
                        <map:parameter name="handler" value="unico"/>
                        <map:parameter name="parameter_userid" value="{request:name}"/>
                        <map:parameter name="parameter_password" value="{request:password}"/>
                        <map:redirect-to uri="autenticacion-exitosa"/>
        </map:act>
        <!-- autenticacion fallida: -->
        <map:generate src="aut_fallida.xml"/>
        <map:transform src="ahtml.xsl"/>
        <map:serialize/>
</map:match>
```

La acción **auth-login** usa los parámetros del manipulador para llamar al recurso de autenticación de este manipulador. Este recurso de autenticación necesita conocer los parámetros ingresados por el usuario. Para cada pieza de información se crea un parámetro propio cuyo nombre inicia con „**parameter_**“. Así en el ejemplo anterior, el recuros de autenticación recibe 2 parámetros: userid y password. Como los valores de estos parámetros fueron enviados por el formulario, estos se deben pasarse al recurso de

autenticación. Si usamos „**{request:...}**“ para el valor de un parámetro, la acción **auth-login** pasará el valor actual de ese parámetro de la solicitud al recuros de autenticación (mediante el uso del concepto de módulos de entrada de Cocoon).

Si el usuario aún no está autenticado por este manipulador, el marco llamará al recurso de autenticación con estos parámetros. Si la autenticación es exitosa, la acción retorna un mapa y se ejecutan los comandos dentro de <*map:act*> del sitemap. Si la autenticación falla, estos comandos no se ejecutan.

Si la autenticación es exitosa, se crea un objeto de sesión en el servidor (si aún no está creado). Si la autenticación falla, la información enviada por el recurso de autenticación se almacena en un contexto temporal llamado simplemente „*temp*“.

El recurso de autenticación

Ejemplo

Se presentará como autenticar un usuario mediante el uso de archivos XML. Cuando el usuario llama recursos protegidos, se verificará si este tiene permisos de acceso, si no se le solicita el nombre y la contraseña. Luego estos datos son autenticados y si tiene los permisos necesarios, se llamarán los recursos solicitados por el usuario.

Introducción

El concepto del marco de autenticación se basa en la noción de manipuladores. Todo se amarra a un manipulador. Los recursos (tuberías) se protegen al asociarlas a un manipulador. Una vez que un usuario se ha autenticado mediante un manipulador en particular, puede acceder a cualquier tubería bajo este manipulador.

El proceso de autenticación se separa en los siguientes pasos:

8. El el usuario solicita un documento
9. El marco verifica si el documento está protegido. Si no está protegido, como respuesta se enviará el documento solicitado.
10. Se sabe que el documento está protegido. El marco verifica si el usuario esta autorizado para ver este documento.

11. Si el usuario está autenticado, la respuesta es el documento original. Si no el marco lo redirije a un documento especial (redirect-to). Este documento es configurable libremente y puede contener por ejemplo información de acceso no autorizado y un formulario de ingreso al sistema.
12. Usando el formulario de ingreso al sistema se puede llamar a un recurso de autenticación con la información del usuario (nombre y contraseña). Este recurso de autenticación utiliza el marco para el proceso de autenticación.
13. En el caso que se autenticó exitosamente al usuario, el puede redirijirlo al documento original (o a cualquier documento de inicio)
14. Si la autenticación falla, el marco llama a otro documento que presenta información al usuario.

Definición de un manipulador nuevo

A continuación se presenta un simple ejemplo. Abrimos el sitemap y buscamos dentro de <map:component-configurations> el elemento <authentication-manager>. Dentro de este elemento es que se definen los manipuladores. (*authentication handler*). En este ejemplo solo se definirá un manipulador sencillo utilizando solo los recursos para autenticación.

Buscar <handlers> – como se presenta abajo. Agregue un manipulador nuevo como se presenta. Es posible nombrarlo como se desee – pero este nombre se debe recordar. Se usará "ags-handler". En los nombres de manipuladores sólo se permiten caracteres alfanuméricos.

```
<map:component-configurations>
        <authentication-manager>
                <handlers>
                        <handler name="agshandler">
                                <redirect-to uri="cocoon://ags-loginpage"/>
                                <authentication uri="cocoon:raw://ags-authuser" />
                        </handler>
                </handlers>
        </authentication-manager>
</map:component-configurations>
```

En la configuración del manipulador se hace referencia a dos tuberías:

- <redirect-to ...> Este es el URI que se llamará si el usuario aún no se ha autenticado. Este normalmente presenta una página de ingreso al sistema, donde el usuario coloca su nombre y contraseña.
- <authentication...> Este es el URI que se llamará cuando el usuario ya ha sido autenticado.

Note que se necesitará más adelante utilizar 4 acciones que deben estar definidas en <map:actions>. Estas

acciones son: **auth-protect**, **auth-login, auth-logout y auth-loggedIn**. Además se debe asegurar que se tiene definido el transformador **encodeURL** dentro de la sección <map:transformers>.

Construyendo las tuberías

Luego se deben construir las diferentes tuberías. Se necesita una tubería que presente la página de ingreso (**ags-loginpage**), una tubería (**ags-login**) que será llamada desde el formulario **ags-loginpage**. El retorno de la tubería *ags-login* hará que se llame la tubería de autenticación de usuarios, llamada (**ags-authuser**).

La primera tubería presenta el formulario al usuario:

```
<map:match pattern=”ags-loginpage”>
        <map:generate src=”loginpage.xml”/>
        <map:transform src=”loginpage.xsl”/>
        <map:transform type=”encodeURL”/>
        <map:serialize/>
</map:match>
```

En este ejemplo, la redirección que se ejecuta al autenticar exitosamente al usuario siempre irá a la misma tubería. En realidad en este caso, el parámetro “*resource*” del XML a continuación siempre es ignorado. Este tema se explicará después. Los nombres de los campos de entrada son importantes, se requieren más adelante. A continuación el archivo loginpage.xml:

```
<?xml version="1.0"?>
<content>
        <form>
                <url>ags-login?resource=ags-protected</url>
                <field name="name" type="text" length="24" description="User"/>
                <field name="password" type="password" length="10" description="Password"/>
        </form>
</content>
```

A continuación el archivo loginpage.xsl. Note como la dirección url del XML de arriba se usa en la acción POST:

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

<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="content">
        <html>
                <form method="post" target="_top">
```

```
                        <xsl:attribute name="action"><xsl:value-of select="normalize-space(form/url)"/></xsl:attribute>
                        <table>
                                <xsl:apply-templates select="form/field"/><br/>
                        </table>
                        <input type="submit" value="Login"></input>
                </form>
        </html>
</xsl:template>

<xsl:template match="field">
        <tr>
                <td>
                        <font face="Arial, Helvetica, sans-serif" size="2"><xsl:value-of select="@description"/>:</font>
                </td>
                <td>
                        <input>
                                <xsl:attribute name="name"><xsl:value-of select="@name"/></xsl:attribute>
                                <xsl:attribute name="type"><xsl:value-of select="@type"/></xsl:attribute>
                                <xsl:attribute name="size"><xsl:value-of select="@length"/></xsl:attribute>
                        </input>
                </td>
        </tr>
</xsl:template>

<!-- Copie todo y aplique plantillas -->
<xsl:template match="@*|node()">
        <xsl:copy>
                <xsl:apply-templates select="@*|node()" />
        </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Ahora se presenta la tubería que se llama del formulario (*ags-loginpage*) que se creó arriba:

```
<map:match pattern="ags-login">
        <map:act type="auth-login">
                <map:parameter name="handler" value="agshandler"/>
                <map:parameter name="parameter_name" value="request:name"/>
                <map:parameter name="parameter_password" value="request:password"/>
                <!-- Si la autenticación es exitosa => se ejecutará la redirección -->
                <map:redirect-to uri="ags-protected"/>
        </map:act>
        <!-- La autenticación falló -->
        <map:generate src="login-failed.xml"/>
        <map:transform src="login-failed.xsl"/>
```

```
        <map:serialize/>
</map:match>
```

La tubería usa la acción auth-login y hace que el manipulador agshandler llame los recursos de autenticación con los parámetros ingresados por el usuario. Note el uso de los nombres del formulario. Si la autenticación es exitosa (lo veremos en un momento) entonces se ejecuta una redirección fija hacia el único recursos que se ha incluido en la tubería.

Esto es igual al ejemplo del portal, la redirección siempre lleva a la página actual del portal. La redirección en realidad debe ir hacia la página que el usuario llamó previamente. Aquí esta como esto debe funcionar en realidad:

En general: cuando el manipulador redirije al usuario hacia la página de login, pasa un parámetro "resource" que contiene el URI que el usuario intentó originalmente llamar. En la hoja de estilo que construye la página, este parámetro debe ser agregado al <url>. En este ejemplo es fijo. Entonces este parámetro será pasado a la tubería de autenticación y luego puede ser usado en la redirección.

Los archivos XSL y XML presentados arriba son simples para hacer el ejemplo sencillo. Pero en realidad es posible hacer cualquier cosa que se desee.

Luego tenemos la tubería que autentica el nombre y contraseña de usuario contra una base de usuarios (en este caso un archivo XML sencillo). Para seguridad se recomienda que esta tubería sea de uso interno.

```
<map:pipeline internal-only="true">
        <map:match pattern="ags-authuser">
                <map:generate src="ags-users.xml" />
                <map:transform src="ags-users.xsl">
                        <map:parameter name="use-request-parameters" value="true"/>
                </map:transform>
                <map:serialize type = "xml"/>
        </map:match>
</map:pipeline internal-only="true">
```

A continuación el archivo ags-users.xml. Parecido al del ejemplo del portal:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<authentication>
        <users>
                <user>
                        <name>cocoon</name>
                        <password>cocoon</password>
```

```
                <role>admin</role>
                <title>Mr.</title>
                <firstname>Walter</firstname>
                <lastname>Cocoon</lastname>
                <company/>
                <street/>
                <zipcode/>
                <city/>
                <country/>
                <phone/>
                <fax/>
                <email/>
                <bankid/>
                <bankname/>
                <accountid/>
        </user>
        <user>
                <name>guest</name>
                <password>guest</password>
                <role>guest</role>
                <title>Mrs.</title>
                <firstname>G.</firstname>
                <lastname>Guest</lastname>
                <company>Cocoon</company>
                <street>Cocoon Street</street>
                <zipcode>33100</zipcode>
                <city>Cocooncity</city>
                <country>Somewhere</country>
                <phone/>
                <fax/>
                <email>guest</email>
                <bankid/>
                <bankname/>
                <accountid/>
        </user>
    </users>
</authentication>
```

Ahora se necesita de la hoja de estilo ags-users.xsl. Toma los parámetros solicitados y el archivo de usuarios y luego verifica si el usuario se encuentra ahí:

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

<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:param name="password"/>
<xsl:param name="name"/>

<xsl:template match="authentication">
	<authentication>
		<xsl:apply-templates select="users"/>
	</authentication>
</xsl:template>

<xsl:template match="users">
	<xsl:apply-templates select="user"/>
</xsl:template>
<xsl:template match="user">
	<xsl:if test="normalize-space(name) = $name and normalize-space(password) = $password">
		<id><xsl:value-of select="name"/></id>
		<role><xsl:value-of select="role"/></role>
		<data>
			<name><xsl:value-of select="name"/></name>
			<role><xsl:value-of select="role"/></role>
			<ID><xsl:value-of select="name"/></ID>
			<user><xsl:value-of select="name"/></user>
			<title><xsl:value-of select="title"/></title>
			<firstname><xsl:value-of select="firstname"/></firstname>
			<lastname><xsl:value-of select="lastname"/></lastname>
			<company><xsl:value-of select="company"/></company>
			<street><xsl:value-of select="street"/></street>
			<zipcode><xsl:value-of select="zipcode"/></zipcode>
			<city><xsl:value-of select="city"/></city>
			<country><xsl:value-of select="country"/></country>
			<phone><xsl:value-of select="phone"/></phone>
			<fax><xsl:value-of select="fax"/></fax>
			<email><xsl:value-of select="email"/></email>
			<bankid><xsl:value-of select="bankid"/></bankid>
			<bankname><xsl:value-of select="bankname"/></bankname>
			<accountid><xsl:value-of select="accountid"/></accountid>
		</data>
	</xsl:if>
</xsl:template>
</xsl:stylesheet>
```

Y este es un recurso que ahora está protegido. Se protege agregando la acción auth-protect a la tubería:

```
<map:match pattern="ags-protected">
	<map:act type="auth-protect">
```

```
            <map:parameter name="handler" value="agshandler"/>
            <map:generate src="ags-resource.xml"/>
        </map:act>
        <map:transform src = "ags-resource.xsl"/>
        <map:serialize />
</map:match>
```

Así que para proteger un recurso mediante el manipulador agshandler, solo debe agregar la acción auth-protect como se indica arriba.

Autenticar en una base de datos

Si se requiere autenticar los usuarios contra una base de datos, se debe modificar la tubería de uso interno **ags-authuser**. Esta es una tubería normal, así que se puede usar cualquier componente de Cocoon 2 que se desee. El ID de usuario y la contraseña se pasan como parámetros de solicitud a la tubería así que se pueden usar en el proceso de autenticación.

Si la autenticación tiene éxito, se debe retornar un XML parecido a este:

```
<authentication>
        <ID>ID único del usuario en el sistema</ID>
        <role>El rol del usuario</role>
        <data>
            Cualquier información adicional se puede incluir aquí. se guardará en el contexto de la sesión
        </data>
</authentication>
```

Logout

El proceso de salida se dispara con la acción “auth-logout”. En el ejemplo al salir se envía al usuario a la página de inicio de sesión.

```
<map:match pattern=”salir”>
        <map:act type “auth-logout”>
            <map:parameter name=”handler” value=”ags-handler”/>
        </map:act>
        <map:redirect-to uri=””/>
</map:match>
```

Esta acción saca al usuario del manipulador elejido y elimina toda la información almacenada en la sesión acerca de este manipulador.

Manejo de Usuarios

Manejo de Aplicaciones

Manejo de Módulos

Administración de Usuarios

Resumen de Configuración

Plantillas de Tuberías

Mas información

Para obtener más información, visite: http://xml.apache.org/cocoon/developing/webapps/index.html