Comparativa de Frameworks WEB

Índice

Capítulo 1. Introducción

1. ¿Qué es un framework?

Según la wikipedia:

En el desarrollo de software, un Framework es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, un framework puede incluir soporte de programas, librerías y un lenguaje de scripting entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

[http://en.wikipedia.org/wiki/Framework]

FreeDictionary extrae una definición de framework de Jonhson y Foote 1988

Un "Software Framework" es un diseño reusable de un sistema (o subsistema). Está expresado por un conjunto de clases abstractas y el modo en que sus instancias colaboran para un tipo específico de software. Todos los frameworks de software son diseños orientados a objetos"

[http://encyclopedia.thefreedictionary.com/Software+framework]

2. ¿Qué es un Framework Web?

Si aplicamos la definición anterior al desarrollo web, podemos llegar a la conclusión que un framework web es una estructura definida, reusable en el que sus componentes facilitan la creación de aplicaciones web. En cierto sentido podemos afirmar que nos proveen una capa de abstracción sobre la arquitectura original ocultándola o adaptándola para no tener que utilizar el protocolo http de manera nativa y así acelerar los tiempos de desarrollo y mantenimiento.

3. Arquitectura básica de la web

La web o www está basada en el protocolo http. Este protocolo, que apareció a principios de los 90, tiene dos tipos de mensajes: request y response, que se suceden sincrónicamente, es decir, no hay response sin un request previo y no puede haber más de un request consecutivo sin response ni viceversa. Estos mensajes se envían una vez establecida la conexión entre el servidor y el cliente. Hasta la versión 1.0 inclusive, por cada conexión sólo se podía enviar un request y un response antes de que esta se cerrara. A partir de la versión 1.1 se permiten las conexiones persistentes y es posible enviar una secuencia de estos mensajes (intercalados) mientras que ninguna de las partes decida cerrar la conexión.

Para los desarrolladores, este protocolo tiene una característica bastante particular que incide directamente en la arquitectura del sistema a construir y es que es un protocolo stateless (sin estado), o lo que es lo mismo, el servidor no recuerda si antes de un determinado mensaje se enviaron otros desde el mismo cliente ni recuerda los datos que se enviaron.

Los mensajes están estructurados internamente para contener distinto tipo de información.

Un mensaje request posee:

- Dirección de destino
- Tipo de Mensaje (GET, POST, HEAD, etc)
- Encabezado de Mensaje
- Cuerpo de mensaje (Si existe)

El tipo de mensaje indica como debe responder el servidor. En los encabezados se indica ciertos valores que debe tener el cuenta el servidor al tratar la información como por ejemplo, tipo de contenido enviado, tipo de contenido de respuesta admitido por el cliente, etc,etc...En el cuerpo del mensaje puede ir cualquier tipo de contenido siempre que se lo especifique correctamente en el encabezado. Sin embargo, generalmente o no se envía nada o se envía información en formato *urlencoded* que consiste en una cadena atributo-valor codificada de una manera especial para que el servidor no confunda la información con los caracteres de control

Por otro lado, un mensaje response posee:

- Código de resultado (200, 404, 500, etc,etc)
- Encabezados de mensaje
- Cuerpo de mensaje (Si existe)

El código indica como resultó la operación solicitada. El primer dígito indica el tipo de resultado (1=informativo, 2=exitoso, 4=error del cliente, 5=error del servidor). Los encabezados indican por lo general el tipo de contenido y temas referidos por ejemplo a si el navegador debe guardar o no el contenido en cache). Por otro lado, el cuerpo del mensaje, si existe, contiene el resultado de la operación solicitada. Generalmente está en formato html, xml, o de imagen pero puede contener cualquier tipo de datos siempre que se lo especifique correctamente en los encabezados

4. Evolución de los frameworks WEB

Internet fue evolucionando y con ella lo hicieron los frameworks webs, a medida que los desarrollos abandonaban el modelo cliente servidor para pasarse a los clientes livianos se puso más énfasis en la construcción de marcos de desarrollo más potentes

4.1. La web estática

Al principio, en los primeros años de los noventa, el protocolo http sólo se utilizaba para transferir texto almacenado en documentos. El servidor respondía a las peticiones (request) enviando el contenido del archivo solicitado sin mayor procesamiento. No era posible parametrizar nada ni enviar formularios ni nada parecido, sólo la dirección del documento que se deseaba leer en el navegador, nada se podía generar dinámicamente, todo era contenido estático.

4.2. La interfaz CGI

La interfaz CGI (Common Gateway Interface) apareció en 1993 y definía una interfaz a través de la cual los programas y el servidor web se podían comunicar entre sí. Esto permitió que cualquiera con un navegador web pudiera ejecutar programas en la computadora que hace de servidor. A través de la interfaz cgi, el servidor y los programas que se encuentran en un área protegida (generalmente el directorio cgi-bin) se comunican de la siguiente manera:

1. Cuando llega una petición http-request al servidor y este detecta que se la debe transferir al programa, lo instancia y le pasa, a través de la entrada estándar y las variables de entorno, toda la información que necesite (CGI define unas variables de entorno mínimas a pasar, la más conocida llamada QUERYSTRING).
2. El programa lee esas variables y la entrada estándar, ejecuta su lógica y escribe el resultado en la salida estándar.
3. El servidor toma la salida estándar del programa y la envía al navegador como mensaje http-response.

Con este agregado se empezaron a construir las llamadas Aplicaciones WEB. Estas estaban escritas generalmente en lenguajes de scripting como Perl o, mas tarde, PHP, aunque también existían cgis compilados programados en lenguajes como c o c++. Sin embargo, trabajar con la interfaz sola hacía difícil la construcción de aplicaciones grandes. Entre las desventajas principales estaban:

- Problemas de escalabilidad: Se creaba una instancia del programa por cada petición
- Se debía chequear recurrentemente que la entrada tuviera el formato correcto
- Era difícil de separar el contenido estático del contenido generado por el programa

A medida que las aplicaciones webs se fueron volviendo cada vez más importantes, el modelo de de interfaz entre el servidor y los ejecutables empezó a ser insuficiente y se buscaron soluciones alternativas.

4.3. Los Servidores de Aplicaciones WEB

Con la aparición de java y sus servlets, SUN creó una abstracción de los mensajes que recibe el servidor en objetos (GenericServlet, HttpServlet) de los que se debía heredar para redefinir su comportamiento. Estos objetos residen en un contenedor llamado Servlet Container que es el encargado de gestionar su ciclo de vida y de transformar la entrada http en objetos que se pasaban como parámetro a estos servlets. Así se solucionaban varios de los problemas que tenía la tecnología CGI como la escalabilidad y el chequeo inicial de los datos pero seguía existiendo el problema de la mezcla de contenido estático (html) con el código programado, sobre todo porque el código estático, que era la mayor parte de lo que se enviaba se escribía línea por línea mediante funciones del lenguaje lo que lo hacía completamente ilegible e inmantenible. Por este motivo SUN creó la especificación JSP que permitían la escritura de una página html con algunas etiquetas especiales en las que se podía impregnar código java. Esta página, la primera vez que se ejecutaba se compilaba y se transformaba en un servlet con la misma funcionalidad que los originales, pero este detalle de poder escribirla como una página web común y corriente fue lo que logró que se desarrollaran aplicaciones mucho más rápido y más mantenibles

4.4. Frameworks Modelo I y Modelo 2

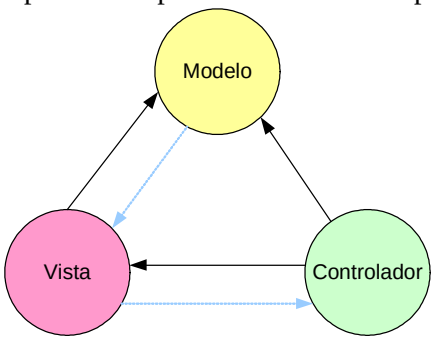
Los términos Modelo I y Modelo II surgen de borradores de las primeras especificaciones de Java Server Pages (JSP) en donde se describían dos patrones básicos para construir aplicaciones basadas en esa tecnología.

El modelo 1 carece del concepto de controlador central que sí tiene el modelo 2.
Una arquitectura de Modelo 1 consiste básicamente en que desde la página JSP accedida desde el navegador se tuviera acceso a las clases del modelo, se les aplicara la lógica del negocio y se seleccionara la siguiente vista a ser enviada al navegador. Cada página procesaba su propio input.

En una arquitectura Modelo 2 existe un servlet que actúa de controlador central que recibe todos los requests y a partir de la dirección de origen, de los datos de entrada, el estado actual de la aplicación selecciona la siguiente vista a mostrar. En este servlet se pueden concentrar todos los temas relativos al conjunto total de las llamadas como la seguridad y la auditoría (logging). Las aplicaciones son más extensibles porque las vistas no se relacionan con el modelo ni entre sí. Por otro lado, a la las clases que implementan la lógica de la aplicación les llega la información digerida para su tratamiento (por ejemplo, en struts los datos forman parte de un formulario). Este servlet central recibe el nombre de Front Controller.

4.5 El modelo MVC

El modelo MVC es un patrón de arquitectura creado por gente de la comunidad Smalltalk que consiste en separar una aplicación en tres componentes principales



- El controlador, que es el que único que puede recibir acciones del los usuarios externos.
- El modelo, que posee el estado interno de la aplicación (determinado por el estado de sus entidades) y las reglas del negocio.
- Las vistas que reflejan el estado algún aspecto del estado del modelo.

La mayoría de los frameworks webs buscan separar estos tres componentes por lo que en muchos manuales y tutoriales se podrá ver escrito que fueron desarrollados siguiendo el patrón MVC. Sin embargo, los amantes de smalltalk dirán que dichos frameworks no tienen una implementación pura de MVC sino una adaptación realizada para adecuarse las características propias de la web (ej: sincronismo de mensajes). Una de las formas que utilizan para implementar el patrón MVC es a través del patrón FrontController en el cual el servlet que procesa todas las peticiones se encarga de comunicarse con el modelo y seleccionar la siguiente vista proveyendo abstracción entre los componentes

Capítulo 2: Frameworks

Evaluar todos los frameworks webs excede a esta publicación y, supongo, a cualquier otra ya que son incontables. De hecho, cada desarrollador puede armarse su propio framework de acuerdo a su conveniencia.
A continuación se describen algunos de los frameworks más significativos que hay en el mundo del desarrollo de aplicaciones web.

1. Struts

Struts es por el momento el más difundido de los frameworks opensource en el ámbito java. Está basado en el modelo 2 y consta, por lo tanto, de un servlet que actúa de controlador central que recibe todas las peticiones de los clientes.
Las facilidades de desarrollo que ofrece son:

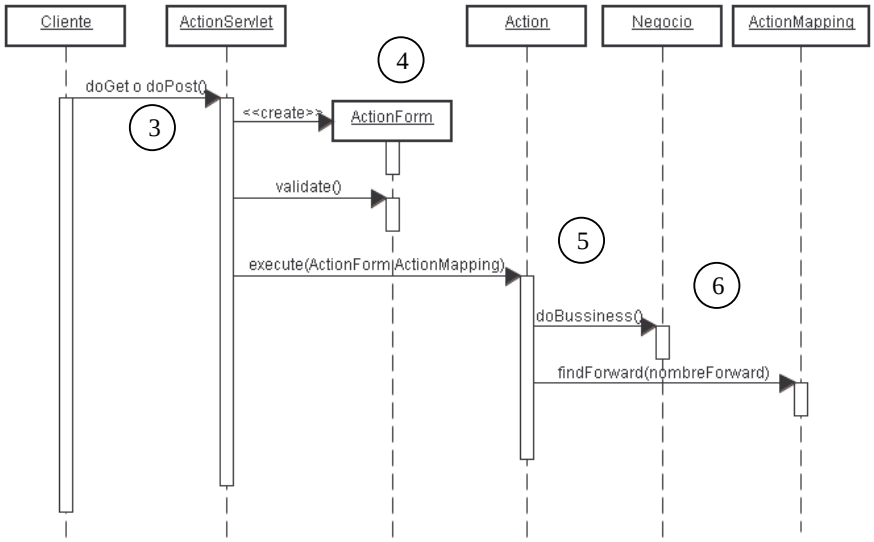
- Lógica de navegación entre páginas
- Binding entre java y el html
- Validación de entradas
- Internacionalización
- Independencia del motor de visualización
- Maquetación

Lógica de navegación entre páginas

Struts tiene un vocabulario específico para definir cono funciona los términos más importantes de definir son los siguientes:

- **Actions:** Posible acción a invocar. Son objetos que heredan de la clase Action donde se escribe que es lo que se hará. Por ejemplo se puede decidir invocar alguna regla de negocio y en base a su resultado mostrar la vista que corresponda.
- **ActionMaping:** mapea las URLS (estructura con la que se manejan los clientes webs) a acciones (objetos). Es decir, se le da un nombre a cada clase acción de manera que puedan ser invocadas desde el cliente como un string.
- **ActionServlet:** Es el servlet controlador.
- **ActionForm:** Encapsulan los parámetros de las peticiones de los clientes presentándolos como datos de un formulario. Representan los datos de entrada de la acción a realizar. Un formulario se puede compartir entre varias peticiones de manera que se pueda ir llenando de a partes antes de invocar a la acción.

Funcionamiento básico:



1. El cliente solicita una página que contiene datos a completar. *(no mostrado)*
2. El servidor le envía la página. *(no mostrado)*
3. El cliente, con los datos completados envía de regreso la página. El ActionServlet verifica la ruta con la que se lo invocó y extrae el path de esa ruta y busca en los actionMappings cual es la Acción a invocar y que formulario necesita recibir como entrada.
4. El controlador crea o reutiliza el Formulario dependiendo el ámbito en que es ejecutada la petición, carga los datos en el formulario, los valida y luego crea la acción y le pasa el formulario como parámetro.
5. La acción recibe el formulario y con sus datos invoca a las reglas del negocio (generalmente delegadas en otros objetos).
6. A partir de la respuesta recibida, carga los valores de salida y selecciona la siguiente vista a enviar.

La inteligencia del controlador se define en un archivo xml llamado struts-config.xml. En este archivo se guardan los mapeos, las acciones y los formularios existentes con los que trabajará el framework Un ejemplo de xml sería el siguiente:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC ... >
<struts-config>
    <form-beans>
        <form-bean name="loginForm" type="com.empresa.LoginForm"/>
    </form-beans>
    <action-mappings>
        <action path="/inicio" forward="/jsp/inicio.jsp"/>
        <action path="/login" forward="/jsp/login.jsp"/>
        <action path="/slogin" type="com.empresa.LoginAction" name="loginForm"
                scope="request" validate="true" input="/jsp/login.jsp">
            <forward name="exito" path="/jsp/inicio.jsp"/>
            <forward name="falla" path="/jsp/login.jsp"/>
        </action>
        <action path="/salir" type="com.empresa.SalirAction">
            <forward name="exito" path="/jsp/salir.jsp"/>
        </action>
    </action-mappings>
    <message-resources parameter="resources.application"/>
</struts-config>
```

En esta configuración se definen un formulario llamado loginForm que es de la clase `com.empresa.LoginForm` y que se utiliza como entrada de la acción `slogin`.
Luego se definen cuatro acciones; 2 sólo redireccionan y las otras dos tienen su implementación en las clases `LoginAction` y `SalirAction`. Todas las acciones tienen definido un path de acceso para poder realizar el mapeo entre la URL y la acción. Por otro lado se ve que los resultados posibles de la acción slogin (`LoginAction`) son éxito o falla que redirigen a diferentes vistas. De la misma manera, la acción salir tiene un único resultado posible que es éxito y redirige a la vista `salir.jsp`. Por último, se utiliza el xml para avisarle al framework que existe un archivo de recursos para los mensajes llamado `resources.application`.

Binding entre java y html.

Si por el lado del controlador existe un servlet que se encarga de todo, por el lado de la vista struts entrega una serie de bibliotecas de TAGs para embeber en el html de manera que sea más fácil acceder a los beans y generar las vistas.
Estos tags se dividen en 4 grupos.

- struts-bean: manejo de los beans a los que tiene acceso la página
- struts-html: renderiza los componentes html comunes
- struts-logic: permite direccionar el flujo de ejecución de la página según condiciones
- struts-nested: permite el anidado de componentes

Estas etiquetas permiten generar vistas conteniendo sólo tags y sin código java. La manera de relacionar los datos entre la aplicación java y la vista se realiza de la siguiente forma:
En la vista, cada etiqueta que necesite tener un valor accesible desde el acción deberá tener un nombre igual al del form asociado con el action. De esta manera Struts realiza el binding entre el valor del tag y el valor del atributo del formulario de manera automática.

Por ejemplo, si en el form html hay una entrada texto que se debe guardar en un bean

```
<html:text property="nombre">
```

en el bean de formulario asociado al action deben existir los correspondientes

```
public string getNombre()
public string setNombre()
```

Por otro lado, si desde la vista se desea acceder a valores (objetos, propiedades, etc) establecidos desde el action. Se puede utilizar los tags correspondientes para acceder a los objetos. Es posible navegar los objetos mediante un lenguaje especial para acceder a sus propiedades.

Por ejemplo, si se desea recorrer una colección de tiendas y el nombre de su responsable imprimiendo sus valores se podría hacer así

```
<table>
  <logic:iterate id="unatienda" name="tiendas"
   scope="request" type="com.empresa.Tienda" >
   <tr>
     <td><bean:write name="unatienda" property="nombre" /></td>
     <td><bean:write name="unatienda" property="responsable.nombre" /></td>
   </tr>
  </logic:iterate>
</table>
```

En este caso, en el bean del formulario debe existir un método `getTiendas` que posea una colección de objetos de la clase `Tienda`. La Tienda a su vez deberá tener un método `getResponsable` que devuelva un objeto que tenga un método llamado `getNombre`. A cada ítem de la colección se le hace referencia dentro del tag `bean:write` a través del nombre `unaTienda`.

Internacionalización

Struts brinda soporte para internacionalización extendiendo la funcionalidad que ya provee java. Permite internacionalizar una aplicación utilizando archivos de texto conteniendo conjuntos de datos con el formato clave=valor y referenciando estas claves en los archivos de la vista

De esta manera, escribiendo en un archivo de texto (denominado por ejemplo recursos.properties) algo como

```
…
app.titulo=Página Principal
form.Nombre=Nombre
form.Apellido=Apellido
boton.enviar=Enviar
….
```

es posible referenciarlos en el html de la siguiente manera

```
<html:html>
<head>
  <title><bean:message key="app.titulo"/></title>
</head>
<body>
  <html:form action="/unaaccion">
     <html:message key="form.nombre"/>:<html:text property="nombre">
     <html:message key="form.apellido"/>:<html:text
property="apellido">
     <html:submit>
        <bean:message key="boton.enviar"/>
     </html:submit>
  </html:form>
</body>
</html:html>
```

Luego, si se quiere localizar por ejemplo para idioma inglés sólo hace falta crear un nuevo archivo recursos_En.properties copiando el contenido del original y reemplazando los valores del lado derecho por los del idioma correspondiente.

Validación de entradas

Struts provee mecanismos de validación de las entradas ingresadas. Existen dos maneras principales: Redefiniendo el método `validate()` de los ActionForms o a través de lo que primero fue un plugin y luego se incorporó a la versión principal y que se denomina struts-validator.

Esta parte del framework permite agregar validadores a los campos de los formularios que se ejecutarán tanto del lado del cliente (mediante javascript) como del lado del servidor así como también definir las rutinas de validación más utilizadas. Todo esto se configura agregando las reglas de validación en un archivo de configuración denominado validation-rules.xml y expresando las restricciones de los campos de cada formulario en el archivo validation.xml.

Un ejemplo de archivo validation-rules.xml en que se define una función de validación denominada validador1 sería el siguiente

```
<form-validation>
  <global>
    <validator name="validador1"
          classname="clase.java.de.validacion"
             method="MetodoDeLaClase"
       methodParams="java.lang.Object,com.un.Tipo"
                 msg="error.mensajes.nombre">
      <javascript>[codigo javascript]</javascript>
    </validator>
  </global>
</form-validation>
```

y un ejemplo de archivo validation.xml donde al formulario `FormInicio` se le establece que el campo `nombreusuario` será requerido sería así

```
<form-validation>
  <formset>
    <form name="FormInicio">
      <field property="nombreusuario" depends="required">
        <arg0 key="forms.nombreusuario"/>
      </field>
    </form>
  </formset>
</form-validation>
```

De esta manera se puede observar que la validación de entradas se mantiene de manera externa a la aplicación java y es posible agregar o quitar restricciones sin volver compilar.

Maquetación

La maquetación de la aplicación WEB es facilitada a través de Struts Tiles, un plugin que permite componer a partir de porciones de página, la página definitiva que será enviada al cliente. La composición de las partes se puede definir de tres maneras:

- A través de un xml
- Dentro de las páginas jsp
- Programáticamente desde las Actions

Algunos aspectos interesantes para mencionar son el soporte de internacionalización (composición de partes según el Locale); las composiciónes se pueden heredar y redefinir; es posible tener varias composiciones y seleccionar una de acuerdo a una clave.

Independencia del motor de visualización

Struts en principio es independiente del motor de visualización aunque generalmente se elija jsp para mostrar las vistas. Existen formas de que struts envíe las vistas para que sean procesadas por motores de plantillas como velocity, transformadores de estilos de documentos XSLT u otros frameworks de presentación como JSF. Por lo tanto struts no está ligado a un motor de visualización particular sino que puede convivir con varios de estos, incluso utilizándolos en simultaneo.

2. Tapestry

Tapestry es otro framework open-source modelo 2 mantenido por la comunidad Apache y una de sus principales características es que está basado en un modelo de componentes. Esto provee una estructura consistente, permitiendo al framework asumir responsabilidades sobre conceptos como la construcción de URL, el despacho, el almacenamiento del estado en el cliente o en el servidor, la validación del usuario, la localización/internacionalización, el manejo de reportes, etc. Un componente es un objeto que tiene sus responsabilidades definidas por el diseño y la estructura del framework en el cual se encuentra. Es decir, sigue una serie de convenciones (nomenclatura, implementación de ciertas interfaces, etc) que le exige el framework. Tapestry, al igual que todos los frameworks de modelo 2, tiene un servlet que centraliza toda la lógica de comunicación.

Existen ciertos conceptos clave que se definen para entender el funcionamiento:

página: las aplicaciones consisten de una colección de páginas identificadas unívocamente a través de su nombre. Cada página contiene un template y otros componentes
Template: Un template puede ser para una página o un componente. Contiene html plano con algunos tags marcados con un atributo especial para indicar que en ese lugar deben situarse los componentes.
Componente: Un objeto reusable que puede ser usado como parte de una página. Los componentes generan html cuando se renderiza la página y pueden participar cuando se selecciona un enlace o se envía un formulario. También se pueden utilizar para crear nuevos componentes.
Parámetro: Los componentes tienen parámetros que sirven para enlazar las propiedades de los componentes con las propiedades de la página. Los componentes generalmente leen los parámetros pero a veces pueden modificarlos haciendo que se actualicen las propiedades de la página que estaban relacionadas con ese parámetro.

Desarrollar con Tapestry permite:

- Transparencia en la construcción de las vistas
- Binding entre java y html
- Manejo de eventos
- Construcción de componentes
- Validación de entradas
- Internacionalización

Transparencia en la construcción de las vistas

Las vistas de tapestry no son ni más ni menos que archivos en html estándar. No existen las bibliotecas de tags ni código java. La única diferencia es la existencia de algunos atributos extras en los elementos que aparecen.

Un atributo de los que aparece es el jwcid. Jwcid significa Java Web Component ID por lo que el valor de ese atributo es ni más ni menos que el tipo de componente al que se está refiriendo. Por ejemplo:

```
<input type="text" jwcid="@TextField" value="ognl:inputValue"/>
```

indica que el elemento html input hace referencia un componente TextField. Como ya se definió anteriormente, una página generada con estos atributos se denomina *template Html*. Estos templates, al renderizarse generan el html que es enviado al cliente.

Binding entre java y html

Para vincular el código java con el html, Tapestry utiliza un lenguaje especial en los templates html llamado OGNL que significa Object Graph Navigation Languaje y permite expresar el acceso a un valor de algún objeto en forma de cadena de texto. A partir de un objeto que sirve como punto de partida se puede navegar a través de sus propiedades hasta llegar al elemento deseado. Por ejemplo, si deseo acceder al precio del primer producto del carro de compras en que java lo haría como

```
elCarro.getItemCarro[0].getProducto().getPrecio()
```

En OGNL equivalente sería

```
elCarro.itemCarro[0].producto.precio
```

El ONGL se utiliza para vincular valores y métodos con propiedades y manejadores de eventos respectivamente.

Por otro lado, cada página, además de un html tiene otras dos partes: una especificación y una clase asociada.

El archivo de especificación posee la extensión .Page y es un xml que indica cual es la clase que se hará cargo del template y cuales son los tipos de las propiedades a utilizar.

Así, un ejemplo con las 3 partes de una página se verían así

html

```
<html>
    <head><title>Ejemplo</title></head>
    <body>
        <form jwcid="@Form" listener="ognl:listeners.enviar">
            <input type="text" jwcid="@TextField" value="ognl:valor"/>
            <input type="submit" jwcid="@Submit" value="Enviar"/>
        </form>
    </body>
</html>
```

.Page

```
<?xml version="1.0"?>
<!DOCTYPE ...>
<page-specification class="Clase">
    <property-specification name="valor" type="java.lang.String"/>
</page-specification>
```

Clase.java

```
public abstract class Clase extends BasePage
{
    public abstract String getValor();
    public abstract void setValor(String valor);

    public void enviar(IRequestCycle cycle)
    {
        ....
    }
}
```

Es importante notar que tanto la clase como los métodos de acceso son abstractos esto es porque Tapestry crea una instancia concreta heredando en tiempo de ejecución donde le agrega métodos con implementación propia.

Manejo de eventos

El manejo de eventos se realiza a través de la suscripción de listeners a los componentes que lanzan dichos eventos. En el ejemplo anterior se puede ver como se indica al formulario que el método `enviar()` es encargado de escuchar los eventos que este genera.

```
<form jwcid="@Form" listener="ognl:listeners.enviar">
```

Construcción de componentes

En Tapestry todos son componentes, incluyendo las páginas. Por lo tanto, al crear páginas estamos creando nuevos componentes. La creación de componentes se realiza de manera similar a la creación de páginas. Por lo general se requiere un template html, un archivo de definición (xml) y un archivo con la implementación (java). Es posible crear componentes que incluyan componentes y la facilidad con la que se pueden crear nuevos componentes es casi la misma que para crear nuevas páginas.

Validación de entradas

Los componentes que reciben la entrada del usuario, permiten la validación a través de dos parámetros: displayname y validators.

```
<label jwcid="@FieldLabel" field="component:userName">Nombre: </label>
<input jwcid="userName@TextField" value="ognl:userName"
       validators="validators:required"
       displayName="Nombre" size="30" />
```

En este ejemplo le estamos diciendo a la entrada de texto que tendrá asignado un validador que obliga que haya texto ingresado al momento de enviar el form.

Inicialmente, este código se renderizará como

```
<label for="userName">Nombre: </label>
<input name="userName" id="userName" value="" size="30"/>
```

Sin embargo, si al enviar el form, no se completó el contenido del campo userName, el código se renderizará de esta manera

```
<font color="red"><label for="userName">Nombre: </label></font>
<input name="userName" id="userName" value="" size="30"/> 
<font color="red">**</font>
```

Que se vería más o menos así

Nombre: _____ **

Otra forma de asignar validadores es enlazando los parámetros en la especificación de la página o del componente mediante el elemento binding.

```
<page-specification>
…
 <component id="inputPassword" type="TextField">
  <binding name="validators" value="validators:required,minLength=4"/>
 </component>
</page-specification>
```

En este caso se indica que el componente inputPassword requerirá un valor de entrada con una longitud mínima de 4.

Internacionalización

Para la internacionalización Tapestry utiliza lo que denomina catálogos de mensajes que vendrían a ser algo similar a los ResourceBoundles de java donde se guardan pares de cadenas con el formato clave=valor. Cada componente puede tener un set de catálogos de mensajes. Estos catálogos se nombran con el mismo nombre que el componente pero su extensión es .properties. Si una clave no se encuentra en ninguno de los catálogos Tapestry no informa ningún error sino que genera un valor propio.

Para indicar una referencia a una clave dentro del html se utiliza la palabra message

```
<html jwcid="@Shell" title="message:titulo">
```

Otra manera de hacer lo mismo es a través del elemento span del html

```
<span key="titulo">Un Título</span>
```

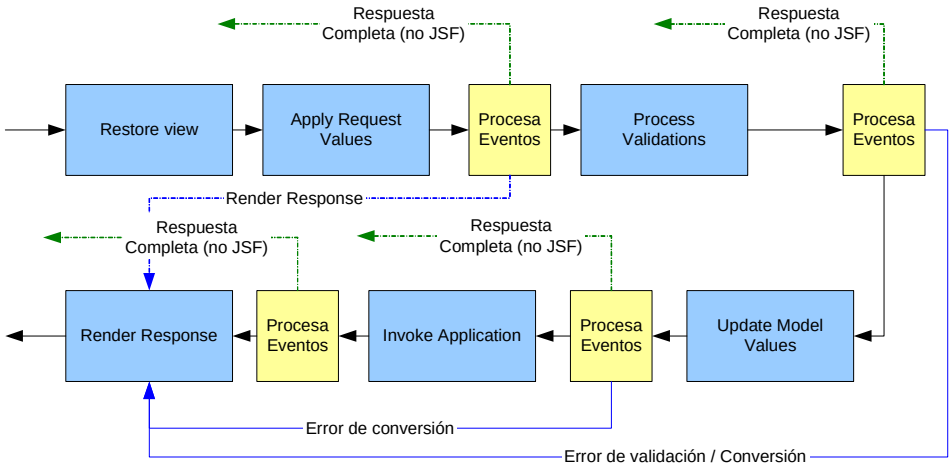
Esta segunda opción es más transparente para los diseñadores y no requiere del prefijo message.

3. Java Server Faces

Lo primero que se puede decir sobre Java Server Faces es que no es una implementación sino una especificación (JSR 127) aprobada por el Java Community Process (JCP) para construir interfaces de usuario para las aplicaciones que corren en un servidor. Esto quiere decir que es un estándar y pueden existir varias implementaciones mientras que cumplan con lo que exija la especificación.

JSF es otro framework modelo 2 que posee un controlador central (FrontControler) que se encarga de manejar todas las peticiones del cliente y gestionar su ciclo de vida. Está basado en un modelo de componentes para la interfaz de usuario. Un componente JSF es un elemento reusable y configurable que se puede utilizar en la interfaz de usuario. Los componentes se pueden anidar. Por ejemplo, una página contiene un componente mapa que a su vez contiene un componente botón. El diseño del framework permite navegar a través del árbol de componentes para acceder a sus propiedades. Además, los componentes pueden reaccionar a diferentes eventos y son capaces de almacenar su estado interno.

Un aspecto fundamental de JSF es el ciclo de vida de una petición web. El ciclo de vida está separado en 6 fases principales



1. **Restore View:** Crea el arbol de componentes de la página solicitada y carga el estado si esta ya había sido solicitada previamente.
2. **Apply Request Value:** Itera sobre el arbol de componentes recuperando el estado de cada uno asignándole los valores que viene desde el cliente.
3. **Process Validations:** Se realizan las validaciones de cada componente
4. **Update Model Values:** Se actualizan los valores de los backing beans del modelo cuyas propiedades estaban vinculadas a propiedades de los componentes de la vista.
5. **Invoke application:** Se ejecuta la lógica del negocio y se selecciona la próxima vista lógica.
6. **Render Response:** Se arma la vista con el estado actualizado de los componentes y se la envía al cliente.

Es posible generar, en cualquier momento, una respuesta que no tenga componentes framework y, por lo tanto, no necesite rendering, como por ejemplo una redirección a una página html estática.

Otra concepto importante es el de los beans administrados. JSF provee un contenedor de inversión de control para administrar los beans que realizan el binding entre la vista y el modelo. En estos beans se recuperan los valores ingresados una vez que estos fueron validados, es decir, en la fase Update Model Values. También se puede definir funcionalidad para controlar el flujo de alguna página o invocar los servicios de la capa de negocio.

Desarrollar con JSF permite:

- Lógica de navegación entre páginas
- Binding entre la vista y los beans de negocio
- Manejo de eventos
- Internacionalización
- Validación de entradas
- Independencia del dispositivo de presentación
- Construcción de componentes

Lógica de navegación entre páginas

El punto de entrada a la aplicación es el FacesServlet, que se encarga de controlar el ciclo de vida de cada petición. JSF permite definir la lógica de navegación a través de uno o más archivos de configuración (faces-config.xml). Dicha lógica se construye a través de reglas de navegación. Cada regla se activa según se cumpla el patrón indicado en el elemento `from-view-id`.

```
<navigation-rule>
  <from-view-id>/buscador.jsp</from-view-id>
  <navigation-case>
    <from-outcome>exito</from-outcome>
    <to-view-id>/resultado.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

El tag `from-view-id` puede contener comodines como por ejemplo /formulario-*.jsp. Dentro de las reglas de navegación pueden haber de cero a muchos casos de navegación. El caso de navegación se selecciona en función del valor utilizado para indicar la acción a realizar (si es que hay alguna) y del valor de retorno de las llamadas al método invoke() de dicha acción que es expresado en el tag `from-outcome`. Por último, se selecciona la vista a mostrar indicada en el elemento `to-view-id` del caso.

Es posible definir casos que se activen ante la ejecución de determinadas acciones y opcionalmente del resultado que estas devuelvan.

Por ejemplo la siguiente configuración se muestran dos casos que se activan en caso de que se ejecute la acción buscar del bean `UnBean`. La diferencia es que el primer caso también requiere como condición de activación que el resultado de la ejecución retorne “exito”.

```
<navigation-rule>
  <from-view-id>/buscador.jsp</from-view-id>
  <navigation-case>
    <from-action>#{UnBean.buscar}</from-action>
    <from-outcome>exito</from-outcome>
    <to-view-id>/resultado.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-action>#{UnBean.buscar}</from-action>
    <to-view-id>/busqueda-error.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

En este caso, si en buscador.jsp se presionara un botón asociado al método buscar del bean UnBean, se realizaría la búsqueda y en caso que la operación sea exitosa la próxima vista a mostrar sería resultado.jsp pero en caso contrario se mostraría busqueda-error.jsp.

Binding entre la vista y los beans de negocio

El modelo de componentes permite el enlace de valores (value binding), de métodos (method binding) y de componentes (component binding).

Binding de Valores:

Todos los componentes de interfaz permiten enlazar sus atributos y propiedades con valores de alguna propiedad de algún bean. Para enlazar los valores con los atributos se debe encerrar entre #{} el campo que se desea enlazar. Un ejemplo puede ser el siguiente.

```
<h:outputText value="#{usuario.nombre}"/>
```

Usuario es un bean que tiene una propiedad nombre cuyo valor aparecerá en la pantalla cuando se renderice el campo de texto.

Si bien se mostró un ejemplo del tipo `#{objeto.propiedad}`, también se permiten otro tipo de expresiones como `#{objeto.dicc[“clave”]}`, `#{objeto.array[4]}`, `#{objeto.propBooleana==true}`, `#{objeto.unNum*5+3}`, etc,etc.

Para poder enlazar correctamente los valores de un componente de interfaz con los de un bean, las propiedades que se enlazan tienen que ser de tipos compatibles o debe haber un convertidor (Converter) asociado. JSF provee un amplio set de convertidores pero también es posible definir nuevos. Para indicar el convertidor a utilizar, se lo puede hacer de así

```
<h:outputText value="#{unBean.fecha}">
   <f:convertDateTime type="date" dateStyle="medium"/>
</h:outputText>
```

Binding de Métodos

Las expresiones para enlazar métodos son una variante de la anterior y permiten la ejecución de un método particular enviando parámetros y recibiendo, si es que existe, la respuesta. La cantidad de parámetros y la respuesta están determinadas por el tipo de método que se espera. Por ejemplo, para enlazar el atributo action con un método, se espera que se lo invoque sin parámetros y que retorne una cadena de texto. Para enlazar un método con el atributo ActionListener, se espera como parámetro un `ActionEvent` y nada como respuesta. Otros tipos de métodos se necesitan para los atributos `validator` y `valueChangeListener`. La sintaxis para enlazar métodos es similar a la utilizada para enlazar valores.

Binding de Componentes

Por último, el enlace de componentes sirve para vincular directamente un componente de interfaz con una propiedad de un bean de manera de poder manejarlo programáticamente. El vínculo se realiza a través del atributo `binding` de los componentes que lo tienen. El objeto enlazado ser una propiedad de un bean que pueda leerse y escribirse y debe descender de UIComponent.

Manejo de Eventos

JSF implementa un modelo que permite la notificación mediante eventos y la subscripción a dichos eventos mediante listeners de manera similar al modelo utilizado en, por ejemplo, Swing. Así, una subclase de UIComponent puede informar de los cambios de estado que considera significativos avisándole a todos los listeners que hayan registrado su interés por los eventos.

Todos los eventos generados desde los componentes de la interfaz son subclases de FacesEvent. Las dos subclases estándares que derivan de FacesEvent son ActionEvent y ValueChangeEvent. Action event generalmente se utiliza cuando se refleja el uso de un control, como por ejemplo, el presionar un botón. ValueChangeEvent se utiliza cuando se quiere reflejar el cambio de algún valor de importancia en el componente.

Del otro lado está la interfaz FacesListener que define los métodos básicos para poder escuchar eventos. ActionListener y ValueChangeListener son las implementaciones correspondientes para los eventos comentados anteriormente.

Los eventos se encolan a medida que van apareciendo. El manejo de eventos ocurre al finalizar varias de las etapas del ciclo de vida de la petición. Un evento debe indicar en que etapa quiere ser entregado informándolo a través del método getPhaseId(). Es posible indicarle una fase en particular o, simplemente, que lo entregue al finalizar la fase en la que fue encolado.

Para registrar los listeners, los componentes deben tener métodos para agregarlos y quitarlos. Estos métodos deben seguir una convención de nombres la cual excede a esta publicación.

Mientras que se ejecuta una fase, se pueden encolar eventos invocando al método queueEvent() de UIComponent. Estos eventos se procesaran una vez terminada la fase.

Al retrasar la difusión de los eventos al finalizar la fase se asegura el correcto procesamiento de todo el árbol de componentes que permiten dejar dicho árbol en un estado consistente para el momento en el que se envían los eventos.

Internacionalización

La internacionalización de JSF está construida sobre la base del soporte que ya brindaba java, la especificación de servlets y y la de JSPs.

JSF maneja el concepto de Locale (configuración local) activo. Este se utiliza cuando se accede a los recursos, cuando se utilizan los conversores, etc. Los posibles Locales que tendrá la aplicación se definen en el archivo de configuración.

```
<application>
  <locale-config>
    <default-locale>es</default-locale>
    <supported-locale>en</supported-locale>
    <supported-locale>fr</supported-locale>
  </locale-config>
</application>
```

El locale se puede establecer programáticamente llamando a `UIViewRoot.SetLocale()`

Para escribir texto internacionalizado en las vistas sólo es necesario informar cual es el recurso (ResourceBoundle) del que se leerán los datos

```
<f:loadBundle basename="mensajes.properties" var="mensajes" />
```

y luego utilizar las claves precedidas por el nombre que se le dio al recurso al cargarlo

```
<h:outputText value="#{mensajes.titulo}" />
```

JSF permite localizar los mensajes internos del framework. Para esto, cada implementación posee un ResourceBoundle `llamado javax.faces.Messages` que contiene todas las claves de los mensajes estándar.

Por ejemplo la clave

`javax.faces.validator.LongRangeValidator.MINIMUM`

que por defecto tiene un valor como

`"Value is less than allowable minimum of ''{0}''".`

puede sobrescribirse y localizarse si se desea.

Validación de entradas

La inteligencia de la validación de entrada reside en los Validadores (Validators). Un validador se encarga de realizar comprobaciones sobre el valor un componente durante la fase *Process Validations*. A Cada componente que recibe la entrada de valores por parte del usuario se le pueden registrar 0 o más validadores. También es posible llamar a los validadores en cualquier momento a través del validate() del componente.

JSF incluye varios validadores estándar pero también permite crear validadores nuevos implementando una interfaz y definiendo los atributos que se utilizarán para configurarlo.

Para registrar un validador en un componente, se lo debe declarar en la vista

```
<h:inputText id="nombre" value="#{usuario.nombre}">
      <f:validateLength minimum="5" maximum="25" />
</h:inputText>
```

o de manera programática mediante e método `addValidator()`.

Validación a nivel de aplicación:

Además de las validaciones de entradas, es posible realizar validaciones a nivel de aplicación o negocio. Supongamos que el existe un formulario que posee un campo donde se ingresa el nombre de una persona y un botón que tiene asociado un método mostrar() que envía a una página donde muestra sus datos.

```
<h:form id=”fromPersona”>
   <h:inputText id=”nombrePersona” value=”persona.nombre” />
   <h:message for=”nombrePersona”>
   <h:commandButtonvalue="#{msg.aceptar}" action="#{UnBean.mostrar}"/>
</h:form>
```

Luego, en el método mostrar se realiza la validación mediante `esPersonaValida` y en caso de no pasar la validación se carga el mensaje y se lo envía a la vista para ser mostrado por el componente h:message.

```
public string mostrar()
{
   if (esPersonaValida(persona.nombre))
   {
     ... /*Ejecuta las reglas de negocio*/
     return “exito”;
   }
   else
   {
      //Obtengo el contexto
      FacesContext context = FacesContext.getCurrentInstance();

      //Obtengo el ResourceBoundle para mostrar el
      //mensaje de error según el locale del cliente
      ResourceBundle bundle = ResourceBundle.getBundle("err.mensajes",
                                 context.getViewRoot().getLocale());
```

```
      String msg = bundle.getString("error.persona_novalida");

      // Agrego el mensaje que será mostrado por el tag <h:messages>
      context.addMessage (“nombrePersona”, new FacesMessage(msg));

      return “error”;
   }
}
```

Independencia del dispositivo de presentación

La codificación de los valores de los componentes para mostrarlos en la vista y la decodificación necesaria de los valores que llegan de las peticiones varía dependiendo del dispositivo. En JSF, esta codificación/decodificación se puede realizar de dos maneras, utilizando un modelo de implementación directa o utilizando un modelo de implementación delegada. En el modelo de implementación directa cada componente posee la lógica para codificarse y decodificarse a si mismo. En cambio, cuando se utiliza el modelo de implementación delegada, esta lógica se deposita en el “Renderizador” que cada componente tiene asociado y que se especifica en la propiedad RenderedType. Con la primera opción se facilita la creación de componentes pero con la segunda se pueden crear componentes que, dependiendo la situación, se le presenten al usuario de diferente manera. Por ejemplo se puede indicar que se utilice un Renderizador determinado para las peticiones que provienen desde un celular o que se presenten de manera especial para un idioma predeterminado.

Construcción de Componentes

JSF permite la creación de componentes propios con o sin render asociado. Sin embargo, la creación no es sencilla y se necesita crear varios archivos dependiendo el tipo de componente que se desea crear. En este texto no se entrará en detalles sobre como crear componentes con JSF al considerar que excede el alcance de este artículo.

4. ASP.NET WebForms

ASP.NET es un conjunto de tecnologías definidas por Microsoft para la capa de presentación WEB que forma parte del .NET Framework. En pocas palabras, una página ASP.NET es un archivo de texto con extensión aspx que el servidor sabe que debe procesar de una manera especial. El texto de las páginas puede ser html junto con código scripting que se compila dinámicamente y se ejecuta en el servidor. La página aspx se compila (sólo la primera vez) a código ejecutable .net cuando algún cliente la solicita al servidor. Para incluir código embebido en la página se utilizan los separadores <% y %>. En este sentido es similar al funcionamiento de las páginas JSP de java. Sin embargo la potencia de este framework no reside en estas características sino en las que se describen a continuación.

Las páginas ASP.NET pueden tener controles que se ejecutan del lado del servidor (server controls) que son objetos que representan elementos de la interfaz de usuario que se ejecutan en el servidor y generan código html como resultado de su ejecución. Los controles tienen propiedades, métodos y eventos a los que pueden responder y mediante los que se puede modificar su estado y comportamiento. Este comportamiento se puede declarar en los atributos de su declaración html o de manera programática.

Los controles permiten contener otros controles dentro de ellos y es posible, al igual que cualquier objeto, heredar y redefinir parte de su comportamiento.

Un control de servidor se identifica en una página html por su atributo runat="server". De esta manera un webform es una página html que contiene en algún lado una etiqueta del estilo

```
<form runat=”server”>
…
</form>
```

Trabajar con ASP.NET permite

- Separación del html y el código .NET.
- Binding entre los elementos de la vista y el código .net
- Validación de entradas
- Manejo de eventos
- Creación de componentes propios
- Internacionalización (*)
- Maquetación (*)

Los ítems marcados con (*) sólo están disponibles en ASP.NET 2.0

Separación del html del código .NET

Existen varias formas de estructurar el código escrito en ASP.NET. Las más conocidas son Code-Inside y Code-Behind.

- Code-Inside consiste en meter todo el código (tanto html como el ejecutable) dentro de la misma página. En este caso no existe separación. Esta opción no es recomendable salvo para ejemplos y aplicaciones muy chicas.
- Code-Behind consiste en separar el código html del código ejecutado en el servidor (c#, vb.net, etc) utilizando dos archivos. El .aspx donde se coloca toda la estructura de la página en html y tags asp.net y el archivo de código fuente donde se escribe la clase que tendrá como responsabilidad manejar las interacciones que ocurran a través del ASPX.

Con la aparición de ASP.NET 2.0 se creó el concepto Code-Beside que toma forma a partir de una característica incorporada en C#2.0 que son las clases parciales. Las clases parciales son clases que se pueden escribir en varios archivos. De esta manera, al crear una página aspx con su clase controladora en realidad pasamos a tener 3 archivos. La clase controladora aspx.cs es ahora parcial y es donde el programador ingresa toda la lógica de control de la página aspx. Paralelamente, .NET crea otra clase parcial oculta al programador donde realiza la inicialización de todos los componentes declarados en los tags de la página aspx. De esta manera, se genera código más limpio a la vista aunque con la desventaja de tener código oculto.

Binding entre los elementos de la vista y el código .NET

El binding entre los elementos de la vista y el código .NET se realiza de manera natural debido a que cuando se declaran controles dentro de las páginas aspx estos no son más que objetos con sus atributos y métodos que están definidos como atributos de la clase controladora de la página. Por regla general, a cada control que existe en el aspx se le corresponde un atributo en la clase controladora (heredera de WebForm) cuyo tipo es el del control declarado. Por ejemplo, si tenémos un aspx como este

```
<%@ Page language="c#" Codebehind="pagina.aspx.cs"
AutoEventWireup="false" Inherits="MyApp.Pagina" %>

<html>
  <body>
    <form runat=”server”>
      <asp:label id="lbMensaje" runat="server"/>
    </form>
  </body>
</html>
```

deberá existir una clase MyApp.Pagina que se encuentre en el archivo pagina.aspx.cs que contendrá un código como este

```
namespace MyApp
{
    public class Pagina : System.Web.UI.Page
    {
        . . .
        protected System.Web.UI.WebControls.Label lbMensaje;
        . . .
        . . .
    }
}
```

De esta manera, desde algún método dentro de la clase página se puede cambiar el estado del objeto lbMensaje. Por ejemplo se le puede cambiar el valor del texto con un código como este

```
Public void CambiarValorTexto(string valor)
{
      lbMensaje.Text = valor;
}
```

y cuando se envíe la página al cliente, esta se renderizará con el nuevo texto. La recolección de los datos enviados por el cliente se realiza de la misma manera.

NOTA: A partir de .NET2.0 la declaración del atributo y la inicialización forman parte de la clase parcial generada de manera oculta por el framework .NET.

Validación de entradas

ASP.NET provee controles y métodos para realizar validaciones de las entradas. A cada campo a completar se le puede asociar uno o más validadores. Existen validadores de existencia, de formato, de rango, etc. Estos controles permiten realizar las validaciones tanto del lado del cliente (javascript) como del lado del servidor. Para activar estas verificaciones es posible establecer la propiedad `CausesValidation` de los controles que se utilizan para enviar datos (como los botones) en `true`. La otra opción activarlas de manera programática mediante el código `Page.IsValid()`. Los controles utilizados también permiten definir las descripciones de los errores a mostrar y la forma en que serán mostrados.

Manejo de eventos

Todos los controles ASP.NET tienen eventos y es posible definirle eventos adicionales a los controles creados por el usuario. El manejo de eventos en ASP.NET es idéntico al manejo de eventos que se realiza en el resto del framework. Esto se debe a que los controles son objetos cuyas instancias aparecen como atributos de la página.
Como ejemplo supongamos que tenemos un control botón situado en el aspx.

```
<asp:button id="btnAceptar" runat="server" Text="Aceptar" />
```

al que queremos añadirle un manejador del evento `Click` para definirle la funcionalidad cuando se lo presiona. Como el control se representa como un atributo de la clase que implementa su Code-Behind. Sólo es necesario asignarle el evento de la misma manera que se hace con cualquier objeto en el .NET framework.

```
btnAceptar.Click += new System.EventHandler(this.btnAceptar_Click);
```

donde `btnAceptar` es un atributo del tipo `WebControl.Button` y `btnAceptar_Click` es un método de la clase que contiene la lógica de la página.
No es necesario modificar nada en el archivo aspx.

Otra opción para vincular eventos con sus manejadores es realizarlo de manera declarativa como atributo del control en el aspx de la siguiente manera

```
<asp:button id="btnAceptar" runat="server" onclick="btnAceptar_Click"
Text="Aceptar" />
```

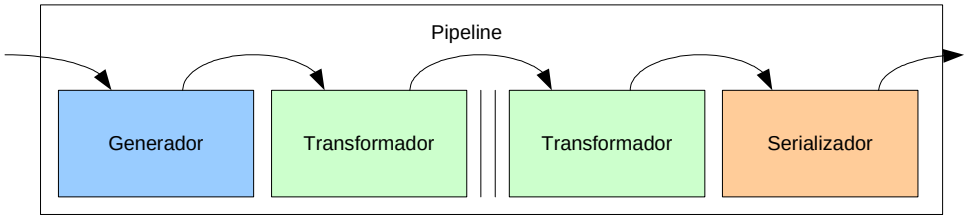
NOTA: gran parte de este código se realiza automáticamente mediante el IDE Visual Studio. La intención de este artículo es mostrar como es realmente el funcionamiento del framework sin importar el entorno de desarrollo.

Creación de componentes propios

ASP.NET permite la creación de componentes propios (en este caso, controles propios) de dos maneras: la primera utilizando los denominados UserControls y la segunda heredando de cualquier WebControl y extendiendo la funcionalidad.

5. Cocoon

Cocoon es un framewok creado en 1999 por Stefano Mazzocchi. Está desarrollado en java y es completamente diferente a los que se detallaron anteriormente. Se basa en un modelo de componentes y en el concepto de tuberías (pipelines) de componentes. Este concepto se asemeja a una línea de producción donde cada componente se especializa en una operación en particular (existen componentes generadores, transformadores, serializadores, etc). El contenido ingresa en la tubería y es modificado por las sucesivas etapas hasta generar la respuesta que es enviada al cliente. De esta manera se busca separar el contenido, el estilo, la lógica y la administración de un sitio web basado en contenido XML.



El elemento global que utiliza para definir una aplicación se denomina sitemap. Este sitemap incluye todo lo que tendrá el sitio web y es configurado a través del archivo xml sitemap.xmap. En este archivo se especifican los componentes, las vistas, los recursos, los conjuntos de acciones y las tuberías que tendrá la aplicación.

Los componentes pueden ser:

- **Generadores (generators):** generan XML como eventos SAX y dan inicio al procesamiento del pipeline
- **Transformadores (transformers):** transforman los eventos SAX recibidos en otros eventos SAX
- **Serializadores (serializers):** transforman los eventos SAX en streams binarios o de texto que pueden ser entendibles por el cliente.
- **Selectores (selectors):** permiten implementar lógica condicional básica (if-then o switch)
- **Mapeadores (matchers):** mapea un patrón de entrada con un recurso
- **Acciones (actions):** son invocadas desde el pipeline y ejecutan algun tipo de operación antes de continuar con el resto del proceso. Las acciones tienen un método act() en donde ejecutan su lógica y devuelven un conjunto de resultados en forma de diccionario (map). Generalmente son las encargadas de invocar a la lógica del negocio.

Todos los componentes poseen un nombre y una clase donde tienen su implementación

Ej:

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

El mecanismo básico de trabajo de Cocoon es tomar un documento XML y ponerlo en una tubería que lo transformará hasta generar la salida correspondiente. Cada tubería empieza con un generador, sigue con cero o más transformadores y termina con un serializador. Los mapeadores y selectores permiten seleccionar la tubería a utilizar y las vistas permiten definir puntos de salida dentro de cada tubería. Además, es posible definir como manejar los flujos de excepción si ocurre algún error durante el procesamiento dentro de la tubería.

Construir aplicaciones con Cocoon permite

- Control de flujo de navegación
- Separación de incumbencias
- Internacionalización
- Independencia del dispositivo de presentación
- Validación de entradas

Control de flujo de navegación

Cocoon provee un control de flujo avanzado permitiendo describir el orden en que las páginas deben ser enviadas al cliente.

Las aplicaciones web tradicionales modelan el control de flujo equiparándolas a máquinas de estados finitos donde la aplicación podía estar en varios estados y las peticiones de los clientes permiten las transiciones entre estos estados. La gente de Cocoon piensa que esto sirve para aplicaciones pequeñas que tienen pocos estados pero no sirven para aplicaciones grandes, es decir, no es escalable hacia arriba. Las aplicaciones tradicionales son orientadas a eventos donde, al recibir un evento se modifica el estado interno y se emite una respuesta. Esto genera complicaciones a la hora de requerir entradas de datos en varios pasos y de definir en donde se ejecuta la lógica que genera la respuesta. Cocoon busca una aproximación más secuencial en el cual se pueda escribir una conversación entre el cliente y el servidor de manera natural. Un ejemplo de código de control de flujo se muestra a continuación y es un ejemplo escrito en javascript que simula un juego en el que hay que adivinar el número elegido por la máquina

```
function main() {
  var numero =  Math.round( Math.random() * 9 ) + 1;
  var texto = "Ingresa un número!"
  var intentos = 0;

  while (true) {
    cocoon.sendPageAndWait("ingresar_nro.jxt", { "numero" : numero, "texto" :
texto, "intentos" : intentos} );

    var ingresado  = parseInt( cocoon.request.get("ingresado") );
    intentos++;

    if (ingresado) {
      if (ingresado > numero) {
        hint = "Incorrecto. Ingresa un número menor"
      } else if (ingresado < numero) {
        hint = "Incorrecto. Ingresa un número mayor"
      } else {
        break;
      }
```

```
    }
  }

  cocoon.sendPage("exito.jx", {"numero" : numero, "ingresado" : ingresado,
"intentos" : intentos} );
}
```

En este código se puede ver que cuando se requieren datos del cliente se envía la página mediante el método SendPageAndWait. La ejecución queda detenida hasta que el cliente complete los datos y los envíe devuelta al servidor. Esto requeriría mantener en ejecución el hilo original mientras que se completan los datos lo cual no es una buena práctica. Por lo tanto Cocoon implementa lo que denomina continuations.

Una continuation es un objeto que, llegado un punto de ejecución, guarda una copia del estado actual del hilo incluyendo la pila, los valores de las variables y la posición de la próxima instrucción a ejecutar. De esta manera, cuando se requiere la entrada de datos de un cliente. Todo el estado del hilo que estaba en el servidor se almacena en una continuation. Cuando el cliente devuelve los datos requeridos informa a Cocoon cual es la continuation que debe restaurar.

Para que Cocoon entienda que necesita recuperar una continuation, en la página enviada al cliente debe existir un código como este

```
<form method="post" action="${cocoon.continuation.id}.kont">
```

y en el archivo sitemap.xmap debe estar definido el mapeador correspondiente para que sepa que si llega una petición con la extensión kont debe restaurar la continuation.

```
<map:match pattern="*.kont">
  <map:call continuation="{1}"/>
</map:match>
```

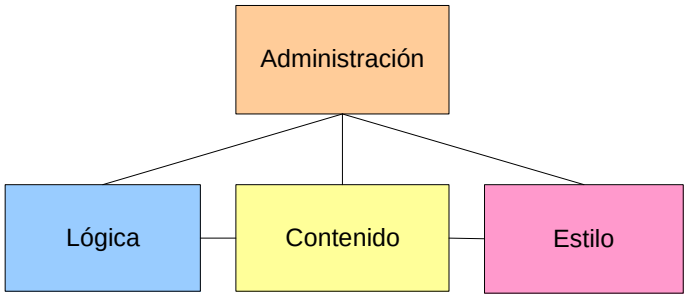
Nota: si bien se utiliza javascript para definir la lógica del controlador, este código es ejecutado en el servidor y no en el cliente como normalmente ocurre. Esto se logra a través de una biblioteca javascript denominada Rhino.

Separación de incumbencias

En un desarrollo en el que están involucradas varias personas difícilmente puedan llevar a cabo sus tareas sin interferirse. Cada integrante tendrá diferentes habilidades y es más que probable que se destaque en algunas y no tanto en otras. No todos somos buenos diseñadores de interfaces ni todos somos buenos programadores.

En general, los integrantes se separan en grupo según sus habilidades para aumentar la productividad, pero difícilmente puedan llegar a realizar su trabajo sin interferirse a menos que tengan bien definidas sus “interfaces” o contratos de interoperabilidad.

Cocoon identifica 4 áreas principales que se pueden aislar y en las que los participantes generalmente tienen habilidades bien definidas. Estas cuatro áreas son Lógica, Contenido, Estilo y Administración. La figura muestra lo que se denomina pirámide de contratos donde se pueden observar las cuatro áreas y cinco líneas que representan los contratos que deben cumplirse entre las áreas.



Este framework fue diseñado para poder aislar estas cuatro áreas respetando estos cinco contratos. Como nota importante hay que observar que no existe el contrato entre todas las áreas sino que hay sólo 5 contratos. Por ejemplo la lógica y el estilo no necesitan contrato por lo que quedan completamente aislados. Este es un aspecto importante a tener en cuenta ya que las aplicaciones web generalmente tienen problemas de interferencia entre estas dos áreas.

Para ilustrar este concepto, se muestra un ejemplo tomado del sitio oficial.

Supongamos que el grupo de contenido define el siguiente XML

```
<page>
  <content>
   <para>Hoy es: <dynamic:today/></para>
  </content>
 </page>
```

En este xml se define una etiqueta <dynamic:today/> que imprime la fecha del día. No es necesario que el grupo de contenido sepa como se obtiene este valor, ni siquiera hace falta saber en que lenguaje se va a programar. Por lo tanto este elemento pasa a ser el contrato entre la lógica y el contenido.

Por otro lado, la estructura del documento funciona como contrato entre el contenido y el diseño, ya que los diseñadores gráficos sólo deben saber como será la estructura y no dependen del contenido específico. El equipo de diseño puede generar diferentes estilos para el mismo documento otorgándole diferente *look and feel* a la aplicación sin que los demás equipos se vean afectados por los cambios.

Por último, la administración de la aplicación se realiza desde el xml de configuración que permite definir la secuencia de los pipelines permitiendo intercambiar el orden de procesamiento y modificando el comportamiento sin necesidad de saber como funcionan internamente los componentes ni afectando a los demás equipos.

Internacionalización

La internacionalización y localización se logra a través de un transformador llamado I18nTransformer que utiliza diccionarios XML para los datos que deben estar en varios idiomas. Es posible traducir texto, atributos, texto con parámetros y localizar formatos de fecha/hora, numeración y moneda.

Por ejemplo, para internacionalizar un simple texto, se lo debe encerrar dentro de un tag `<i18n:text>`.

```
<i18n:text>texto</i18n:text>
```

El valor encerrado dentro del tag sirve como clave de búsqueda en el diccionario.

También es posible internacionalizar los valores de los atributos de algún elemento xml utilizando el atributo `i18n:attr`

```
<para title=”titulo” name=”Articulo” i18n:attr=”title, name”>
```

Aquí se indica que los atributos title y name podrán ser traducidos.

A los diccionarios que contienen las traducciones se los denomina catálogos y son archivos XML con el siguiente formato.

```
<catalogue xml:lang=”locale”>
    <message key=”texto”>texto</message>
    <message key=”titulo”>Lo que el viento se llevo</message>
    ....
</catalogue>
```

Los catálogos se guardan con el formato

`[nombre-catalogo]_[lenguaje]_[pais]_[variante].xml`

pudiendo ignorar las últimas tres secciones de acuerdo al nivel de localización al que se desea llegar.

Los catálogos se deben definir la parte del transformador i18n en el archivo sitemap.xmap aclarando cual es el que se utilizará por defecto.

Independencia del dispositivo de presentación

Cada tubería finaliza con un serializador que convierte el contenido xml en un stream binario o de texto. Cocoon provee serializadores para html, xml, pdf, xhtml, wml (wap), svg, vrml, zip, ps, etc,etc. A la vez es capaz de reconocer el tipo de dispositivo del cual proviene la petición. Esto permite construir la aplicación sin depender del dispositivo en el que se visualizará. Por ejemplo, si primero se pueden utilizar serializadores html para presentar el contenido en navegadores webs, pero más tarde se pueden agregar serializadores wap para mostrar la misma información en dispositivos móviles sólo modificando la manera en que se renderizan los documentos de salida.

Validación de entradas

Cocoon tiene su propio modelo de formularios para obtener datos de los clientes llamado Cforms. Un form está compuesto por un conjunto de widgets. Un widget es un objeto que sabe como leer un valor que llega desde un Request de un cliente, sabe como validarlo, y posee una representación en XML de si mismo. A la vez, un widget es capaz de guardar su estado interno por lo que no es necesario utilizar objetos externos para almacenar su valor.

Para crear un form se necesitan dos cosas, un modelo y un template.
Los modelos se definen a través de archivos XML en donde aparecen cada uno de los widgets que serán utilizados, que tipo de datos aceptará cada uno y cuales serán sus métodos de validación.

```
<fd:form xmlns:fd="http://apache.org/cocoon/forms/1.0#definition">
  <fd:widgets>
    <fd:field id="nombre" required="true">
      <fd:label>Nombre:</fd:label>
      <fd:datatype base="string"/>
      <fd:validation>
        <fd:length min="2"/>
      </fd:validation>
    </fd:field>
    <fd:field id="password" required="true">
      <fd:label>Password:</fd:label>
      <fd:datatype base="string"/>
      <fd:validation>
        <fd:length min="5" max="20"/>
      </fd:validation>
    </fd:field>
    <fd:booleanfield id="recordarme">
      <fd:label>Recordarme</fd:label>
    </fd:booleanfield>
</fd:form>
```

Notar que el modelo no tiene partes correspondientes a como se representa.

Luego se escribe el template que estará basado en este modelo.
...

```
<head>
  <title>Ingreso al sistema</title>
</head>
<body>
  <h1>Registration</h1>
  <ft:form-template action="#{$continuation/id}.cont" method="POST">
    <ft:widget-label id="Nombre"/>
    <ft:widget id="Nombre"/>
    <br/>
    <ft:widget-label id="password"/>
    <ft:widget id="password">
      <fi:styling type="password"/>
    </ft:widget>
    <ft:widget id="recordarme"/>
    <ft:widget-label id="recordarme"/>
    <br/>
    <input type="submit"/>
  </ft:form-template>
</body>
```

Este template también es un XML pero a diferencia del modelo, sólo hace referencia a los widgets y contiene código de presentación intercalado (
, <body>, etc).

Un transformador especial (el form template transformer) será el encargado de realizar el reemplazo de las referencias de los widgets del template por el código original.

Cuando llega una petición web solicitando un formulario, el controlador crea el formulario basado en la definición del form y lo envía a través de un pipeline al cliente. Cuando este lo completa y lo vuelve a enviar, el controlador ejecuta los validadores de los widgets y, en caso de éxito ejecuta la lógica asociada al siguiente paso. En caso de error, reenvía el formulario al cliente para su corrección.

6. Ruby on Rails

Ruby on Rails (ROR) es un framework para desarrollar aplicaciones webs basado en el lenguaje Ruby. En realidad no sólo aplica para la capa de presentación sino que es posible definir desde la lógica de navegación hasta el acceso a datos. ROR busca simplificar el desarrollo de aplicaciones promoviendo las convenciones sobre la configuración. De hecho, el framework no posee archivos de configuración (en realidad posee uno pero es para temas generales como dirección de la base de datos, etc).

Para crear una aplicación con ROR se utiliza un script que genera toda la estructura base con los directorios donde se deben situar los diferentes archivos de código. Por ejemplo, hay directorios para el modelo, la vista y el controlador ya que ROR también está basado en el patrón MVC.

En las vistas, el código ruby es embebido en el código html de manera similar a jsp/php/asp de manera que no se logra una separación total entre el html y el código de implementación. Sin embargo, la lógica de la aplicación, las reglas del negocio, se ejecutan en código ruby puro accedidas a través de los controladores.

Desarrollar aplicaciones con ROR permite

- Mapeo transparente de URLs a métodos
- Mapeo transparente objeto-relacional
- Desarrollo rápido de aplicaciones CRUD

Mapeo transparente de URLs a métodos

ROR permite un mapeo transparente de url a métodos. Esto se logra por la convención que impone el framework para desarrollar los controladores.

En ROR, un controlador hereda de ApplicationControler y su nombre debe terminar con la palabra Controler. Un controlador podría ser, por ejemplo, ProductoControler. Dentro de la clase ProductoControler se pueden definir métodos que invocarán a las reglas del negocio. Por ejemplo se puede definir un método listar.

```
Class ProductoControler < ApplicationControler
  def index
    Render_text “comportamiento del controlador por default”
  end
  def listar
    Render_text “lista los productos disponibles”
  end
end
```

De esta manera, si desde el navegador realizamos una conexión a

```
http://host/Producto/listar
```

Obtendremos como resultado

```
lista los productos disponibles
```

Mapeo transparente objeto-relacional

La persistencia de ROR está basada en el patrón de diseño ActiveRecord que trata los objetos del dominio como si fueran registros de una tabla. Es decir, no ignora que los datos se guardan en una BD. Existe una clase base, desde donde deben heredar todos los objetos del dominio, que provee funcionalidades básicas para acceder y modificar los objetos. Los campos de la base de datos se nombran de la misma manera que los atributos. Y las tablas llevan el mismo nombre que las clases sólo que las tablas tienen sus nombres en plural y las clases en singular.

Una tabla

```
Productos
Id: integer
Nombre: varchar
```

Mapea contra la clase

```
Class Producto < ActiveRecord::Base
end
```

Es posible navegar entre los objetos informando a la clase cual es el sentido en el que se debe navegar y cual es el tipo de relación que existe entre los objetos.

Por ejemplo, si queremos crear una relación bidireccional entre el catálogo y sus productos deberíamos informarlo de la siguiente manera

```
Class Producto < ActiveRecord::Base
  belongs_to :Catalogo
end
```

```
Class Catalogo < ActiveRecord::Base
  has_many :Producto
end
```

Para que esto funcione en la tabla productos debe haber un campo llamado Catalogo que posea una clave foránea a la tabla catálogos. Para quitar alguno de los sentidos de la navegabilidad sólo hay que eliminar alguna de las definiciones.

Desarrollo rápido de aplicaciones CRUD

El fuerte de ROR es, sin duda el desarrollo de aplicaciones CRUD (Create / Retrieve / Update / Delete). Definiendo las tablas y ejecutando unos pocos scripts es posible tener una aplicación totalmente funcional que realice las operaciones básicas.

Suponiendo que se tiene la tabla Productos definida en la base de datos llamada Prueba

Al ejecutar el script

```
rails Ejemplo
```

se crea toda la estructura de la aplicación en el directorio en el que estemos parados. Luego se indica cual es la base de datos editando el archivo config.database.yml.

Para crear el objeto del modelo se invoca a

```
ruby script\generate model Producto
```

Esto genera la clase `Producto` (producto.rb) que hereda de `ActiveRecord::Base` en el directorio model.

Para crear el controlador a través del cual se accederá al objeto se invoca a

```
ruby script\generate controller Producto
```

Esto crea el controlador `ProductoController` (producto_controller.rb) en el directorio controllers.

Si editamos y agregamos al controlador la línea

```
Class ProductoControler < ApplicationControler
  Scaffold:producto
end
```

le otorgamos a la clase todas las funcionalidades para realizar operaciones CRUD sobre producto. No es necesario programar nada más.

```
http://host/Producto/new    //Nuevo producto
http://host/Producto/list   //Lista los productos
http://host/Producto/show   //Muestra el producto
http://host/Producto/edit   //Edita el producto
http://host/Producto/delete //Elimina el producto
```

Además, es posible cambiar el comportamiento redefiniendo los métodos en el controlador.

Como se puede ver, sólo se requirió editar un archivo de configuración y agregar una línea de código para lograr una aplicación funcional básica.

Capítulo 3. El Futuro Próximo

Así como tiene un pasado, la web también tiene un futuro, a continuación una serie de conceptos, tecnolgías y frameworks que aparecen como el futuro cercano en la web.

1. WEB 2.0

Pocos tienen claro que es este nuevo concepto de la web 2.0 y los que sí lo tienen no suelen coincidir entre ellos. Según Tim O'Reilly la web 2.0 tiene como concepto principal a "la red como plataforma". Las aplicaciones web 2.0 son las que toman ventajas de esa plataforma entregando software como si fuera un servicio continuamente actualizado que se torna mejor a medida que más gente lo usa, consumiendo y analizando información de diferentes fuentes (incluyendo a los usuarios) mientras provee sus propios datos y servicios de forma de permitir que otros los consuman creando efectos de red a través de una especie de "arquitectura de participación".

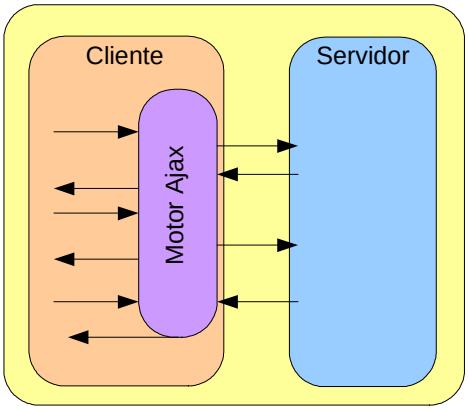
Más allá de la definición, la web 2.0 busca mostrar que existe una diferencia de la forma de utilización de la web de hace unos años a ahora. Mientras que antes las aplicaciones se basaban en mejorar la experiencia del usuario interactuando con el de manera directa a través de su portal o aplicación web, las nuevas aplicaciones permiten interactuar de manera indirecta exponiendo los servicios en diferentes formatos para que no sólo se pueda acceder de la manera corriente sino que otras aplicaciones pueden consumir estos servicios y combinarlos con otros logrando que usuarios lleguen a nuestra información. Esto crea una red de información y servicios potenciada por la interacción que existe entre las comunidades de usuarios de las diferentes aplicaciones. La idea entonces es compartir, abrirse al mundo, socializarse.

Algunas tecnologías que acompañan a la web2.0 son los webservices, RSS/ATOM, XML-RPC y AJAX. Conceptos como las folksonomies, redes sociales, sistemas de confianza, filtros colaborativos, búsquedas personalizadas, agregadores de noticias, social bookmarking, blogs y los contenidos liberados bajo licencias permisivas como creative commons son claves a la hora de entender por donde pasa las web 2.0

En realidad, este tema excede al artículo pero sólo lo menciono para hacer ver cual es el tipo de interacción que espera un usuario en una aplicación web orientada a internet (no creo que se aplique en las intrantets aunque en empresas grandes hay algunos experimentos dando vueltas para ver que resultados trae entre los empleados)

2. AJAX

AJAX es el acrónimo de Asyncronous Javascript and XML. Es decir, no es una tecnología sino un término que agrupa alguna de ellas para no tener que andar nombrándolas todas cada vez que se quiere hablar de una aplicación que las utiliza. La palabra clave de este término es Asincronismo. El asincronismo permite eliminar la restricción que imponía el protocolo http. De esta manera, ya no es necesario esperar la respuesta del servidor y el cliente puede continuar su operación. En el gráfico se puede observar como la interacción entre el servidor y el cliente generalmente tiene un intermediario del lado del cliente que se encarga de procesar las llamadas, enviar las



que corresponden al servidor y notificar cuando arriban las respuestas o cuando otro tipo de información es enviada desde el servidor. Las flechas muestran que no tiene por que haber relación entre lo que envía el cliente al motor AJAX y lo que es enviado realmente al servidor. Lo mismo pasa con las notificaciones en sentido contrario.

El motor AJAX en la mayoría de los casos está basado en el objeto javascript XMLHttpRequest que permite mantener una conexión permanente entre el cliente y el servidor. Si a esto le sumamos que los navegadores modernos (que respetan los estándares) implementan la navegación y modificación del árbol DOM del documento html mediante javascript, obtenemos como resultado aplicaciones que tienen una conexión permanente con el servidor y que pueden cambiar su aspecto modificando parcialmente el html que le es presentado al cliente. De esta manera sólo se refrescan los cambios necesarios y no viaja toda la página como lo hacía con http.

El uso de AJAX permite aplicaciones altamente interactivas dentro de un navegador que pueden mantener un estado en el propio cliente y que se actualizan más fluidamente que las aplicaciones WEB tradicionales al transmitir sólo lo necesario de manera asincrónica.

Como desventajas se puede mencionar que AJAX no es soportado por todos los navegadores y que las implementaciones de javascript varían de navegador en navegador (incluyendo algunos que carecen completamente de el). Esto es un problema si se considera que el objetivo de la web es presentar el contenido sin depender de cómo lo visualiza el cliente favoreciendo la accesibilidad. Por lo tanto, al plantear el diseño de una aplicación AJAX debe considerarse que no se podrá acceder desde gran cantidad de navegadores. Por lo general se recomienda usar esta tecnología como apoyo pero no como base. Es decir, la aplicación debería seguir funcionando si el navegador no soporta AJAX. Usar pero no abusar.

3. Frameworks

A continuación algunos frameworks de esta nueva generación

3.1. DWR

DWR es una librería open source que permite incorporar tecnología AJAX a las aplicaciones WEB desarrolladas en Java. Permite que el código del navegador web use funciones java que se ejecutan en el servidor como si se ejecutaran desde el navegador. DWR está separado en 2 partes principales: Por un lado existe una parte que consiste en código javascript que se encarga de la comunicación con el servidor y de la actualización dinámica de la página. Por otro lado, en el servidor hay un servlet que recibe los requests provenientes de este código javascript, los procesa y devuelve la respuesta.

La utilización de bibliotecas como DWR tiene grandes beneficios sobre la implementación de frameworks AJAX propios. El principal es la portabilidad. Generalmente el javascript utilizado tiene funciones que son dependientes de los navegadores y que, por lo tanto, generan cierta incompatibilidad entre plataformas. El uso de un motor AJAX permite abstraer estas complicaciones y utilizar las funciones provistas por el framework dejando que este se ocupe de las cuestiones particulares de cada navegador.

DWR genera dinámicamente código javascript basado en las clases de java. Así, el programador puede acceder a las clases java desde el cliente a través del javascript.

Como ejemplo, supongamos que se desea llenar un combo con un listado obtenido del bean AdminPaises que se encuentra en el servidor. Para que el bean sea accesible directamente desde el navegador, debe ser declarado en el archivo de configuración que utiliza DWR. El archivo dwr.xml quedaría como este

```
<dwr>
  <init>
    ...
  </init>
  <allow>
    <create creator="new" javascript="Paises" scope="...">
       <param name="class" value="com.empresa.beans.AdminPaises"/>
       <include method="getAllPaises"/>
    </create>
      ...
  </allow>
  <signatures>
    ...
  </signatures>
</dwr>
```

Con este código estamos diciéndole al framework que utilice el constructor por defecto (creator=”new”), que desde javascript se accederá al bean a través del nombre Países (javascript=”Paises”), que la clase a crear es AdminPaises (<param.../>) y que es posible acceder desde javascript al método getAllPaises.

Mientras que en java la invocación de métodos se realiza de manera sincrónica (se espera a que la ejecución del mismo concluya antes de ejecutar la siguiente línea), en javascript se realiza de manera asincrónica. Para solucionar esta incompatibilidad, DWR propone agregar un parámetro extra a cada llamada que se realiza desde el javascript en donde se indica cual es la función que debe procesar lo que devuelva el método java.

De esta manera, en el cliente tendríamos lo siguiente

```
function getPaises()
{
   Paises.getAllPaises(llenarComboPaises);
}

function llenarComboPaises(listado)
{
    DWRUtil.addOptions(“listadoPaises”,listado);
}
```

Y en el servidor, tendríamos el bean AdminPaises que sería parecido a esto

```
public class AdminPaises
{
      public String[] getAllPaises()
      {
           return new String[“Argentina”, “Brasil”, “Uruguay”....];
      }
}
```

Cuando alguna acción en el cliente llama a la función javascript `getPaises`, esta invoca, a través de DWR, el método `getAllPaises` del bean `AdminPaises` que se encuentra en el servidor. En la llamada se indica que `llenarComboPaises` procesará el resultado. La función javascript `getPaises` termina inmediatamente después de enviar a procesar el método en el servidor. De esta manera se pueden seguir procesando eventos en el cliente. Cuando el servidor termina de ejecutar el método y se envía el resultado, DWR toma el valor que obtuvo e invoca la función `llenarComboPaises`. Allí dentro se llama a `AddOptions` que es una función de la biblioteca de utilidades de javascript que provee DWR que se encarga de localizar el combo dentro de la página cuyo id sea "`listadoPaises`" y le agrega los ítems que se le pasaron como parámetro.

DWR no funciona con todos los navegadores pero si lo hace con los de última generación. Además tiene facilidades para integrarse con otros frameworks como spring, struts, hibernate, etc.

3.2. OpenLaszlo

OpenLaszlo es una plataforma de desarrollo de Aplicaciones de Internet Ricas (RIA) basado, principalmente, en Flash como tecnología de presentación, aunque aseguran que es independiente del producto de Macromedia. El framework fue inicialmente diseñado por Laszlo Systems de manera cerrada pero, a partir de octubre de 2004 se lo liberó con licencia libre.

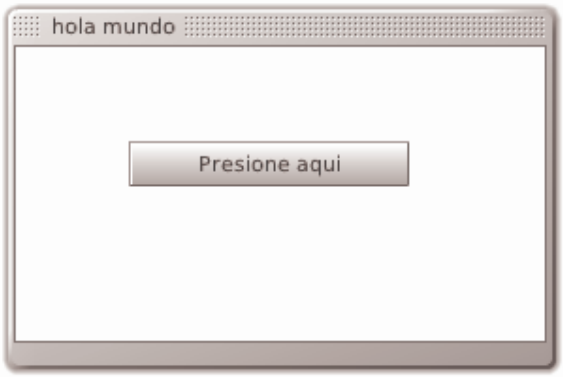
NOTA: En la versión 4 de Laszlo (actualmente en beta) aparece una alternativa a la renderización de componentes en formato Flash y es que ahora existe la posibilidad de que los mismos sean renderizados en DHTML lo cual brinda una gran flexibilidad al no necesitar depender del plugin de Macromedia para poder ejecutar nuestra aplicación.

Las aplicaciones desarrolladas con openlaszlo se escriben en un lenguaje llamado LZX que combina XML, XPATH y JavaScript. Es un lenguaje basado en eventos y orientado a objetos. El LPS (Laszlo Presentation Server) se encarga de recibir las peticiones de los cliented y genera contenido en flash permitiendo desarrollar interfaces de gran interactividad y alto impacto visual en el cliente. Cabe destacar que sólo la primera vez se compila el archivo lzx a flash

Un código como este

```
<canvas>
  <window x="10" y="10" width="300" height="200" title="hola mundo">
  <button x="60" y="50" width="150" height="25">Presione aqui</button>
  </window>
</canvas>
```

Genera una vista como esta



La ventana generada se puede arrastrar y mover por toda el área del navegador.
El sistema de presentación se puede comunicar con los servlets java para acceder a sevicios de la capa de negocio y hacer de gateway entre servicios de otros proveedores utilizando webservices.

La orientación a objetos permite heredar y extender los componentes básicos que provee el framework permitiendo la creación de componentes más complejos y adaptados al problema a resolver.

Las primeras aplicaciones de uso masivo están apareciendo por estos días y, aunque el resultado visual es impactante, por el momento es necesario tener el plugin de flash instalado en el cliente lo que quita muchos de los beneficios de las aplicaciones web como la independencia de la presentación del contenido, accesibilidad para que puedan leer las personas con capacidades diferentes, etc.

Capítulo 4. Comparativa de frameworks actuales

Habiendo repasado algunos de los frameworks más utilizados, se ofrece una evaluación completamente parcial sobre los pros y contras observados en cada framework

Struts:

Pros:

- Más de 6 años demostrando que funciona. Gran cantidad de desarrollos de gran envergadura concretados exitosamente.
- Es el framework más popular de la comunidad java por lo que existen infinidad de material disponible en la web. Buenas prácticas conocidas.
- Documentación muy buena
- Permite crear sitios internacionales de manera rápida y efectiva.
- Curva de aprendizaje mediana.
- Open Source (Licencia Apache)

Contras:

- No abstrae completamente al desarrollador del funcionamiento del protocolo http.
- Aunque se adapta a las incorporaciones de diferentes bibliotecas de tags no está diseñado para facilitar la creación de componentes propios
- No es natural el mapeo de los datos ingresados a los objetos del negocio.

- Las vistas quedan atadas al dispositivo en el cual se renderizan. No facilita el armado de vistas independientes del dispositivo.
- No es una especificación.

Tapestry

Pros:

- Open Source (Licencia Apache)
- Permite el desarrollo de componentes propios
- Los diseñadores web no necesitan aprenderse tags nuevos ni diferentes lenguajes ya que los templates se codifican en html estándar.
- La creación de componentes es relativamente sencilla.
- Separación completa entre la lógica y la presentación (html de java)

Contras:

- Comunidad de desarrolladores pequeña-mediana.
- Escasa documentación. Pocos libros editados. Poca información en la web.
- Se requiere configurar 3 archivos para cada página a crear.

ASP.NET

Pros:

- Curva de aprendizaje baja
- Permite el desarrollo de controles propios y utilizar controles de terceros
- Gran comunidad de desarrolladores
- Soporte oficial y amplia documentación.
- Permite binding directo entre los componentes y los orígenes de datos.
- Permite desarrollo con herramientas RAD

Contras:

- Propietario de Microsoft. Sólo funciona con Information Server (*)
- Requiere un IDE como Visual Studio para un desarrollo productivo. Lo que deviene en un costo por desarrollador por el licenciamiento del IDE.
- El control de navegación no está centralizado.
- Código cerrado. Ante la aparición de bugs dentro del framework se depende de Microsoft para solucionarlo.
- Varias de las funcionalidades importantes (maquetación, internacionalización) sólo están disponibles a partir de la versión 2.0.
- Requiere javascript y cookies para funcionar correctamente.
- El estado interno de la vista (viewstate) viaja codificado dentro de un campo hidden. Esto trae problemas de performance y si se utiliza mal, problemas de seguridad.

(*) Apache cuenta con un módulo para la implementación libre de ASP.NET que corre bajo el proyecto mono.

Cocoon

Pros:

- Permite separar claramente el contenido de la presentación y de la lógica.
- Open Source (Licencia Apache)
- Permite modificar el comportamiento de la aplicación sin conocer el lenguaje en el que está implementado.

Contras:

- Requiere amplios conocimientos de hojas de estilo XSL por parte de los diseñadores gráficos.
- Comunidad relativamente pequeña
- Curva de aprendizaje elevada
- La transformación de XMLs requiere bastante capacidad de proceso.

Java Server Faces

Pros:

- Permite separar claramente el contenido de la presentación y de la lógica.
- Es una especificación, lo que permite tener varias implementaciones (tanto de código cerrado como de código abierto)
- Permite modificar el comportamiento de la aplicación sin conocer el lenguaje en el que está implementado.
- No es necesario conocer el framework en detalle para poder comenzar a utilizarlo.
- Comunidad y herramientas de soporte en aumento.

Contras:

- La creación de componentes propios es compleja.
- Requiere javascript

Ruby on Rails

Pros:

- Alta productividad para desarrollar aplicaciones de tipo CRUD.
- Solución TODO en 1. Desde la presentación hasta la persistencia
- Es posible mantener ambientes separados de prueba y producción
- No necesita configuración (al menos no mucha)
- Gran aceptación en la comunidad de desarrolladores

Contras:

- Aún no existe constancia de aplicaciones de gran envergadura desarrolladas con este framework más allá de varias aplicaciones web masivas.
- Utiliza lenguaje interpretado y débilmente tipado, difícil de depurar.

Conclusiones

La evolución de los frameworks web marca una tendencia clara a la abstracción del protocolo en el que se sustenta (http) para beneficiarse de los modelos basados en controles y componentes que tanto éxito tuvieron en el ámbito del escritorio. Las diferentes aproximaciones de los frameworks muestran que, en la mayoría de estos, se atacan los mismos problemas: navegación, internacionalización, manejo de errores, validación de entradas, escalabilidad, etc. Los más antiguos como struts permiten un manejo más directo de los datos que se procesan mientras que los más modernos como JSF, ASP.NET o Tapestry buscan la abstracción casi total del protocolo pero a cambio de generar modelos de componentes fácilmente extensibles y orientados a eventos.

En la mayoría se busca solucionar el problema de la separación de incumbencias, un tema que traía más de un dolor de cabeza a los equipos en los que había tanto desarrolladores como diseñadores gráficos. No todos lo solucionan de la mejor manera. Por ejemplo ROR no lo tiene en cuenta mientras que en Tapestry es uno de los mayores fuertes y en Cocoon es casi el paradigma sobre el que se impulsa su desarrollo.

Cada framework apunta a solucionar objetivos generales pero sale beneficiado cuando ataca problemas particulares. Por ejemplo, mientras que ROR permite crear aplicaciones de manera rapidísima pero a costa de sencilléz extrema y falta de flexibilidad (si se empieza a configurar y redefinir, ya no es tan rápido el desarrollo), otros como Cocoon y JSF fueron diseñados para no depender del dispositivo de presentación, lo que otorga varios puntos a las aplicaciones que buscan verse bien en dispositivos móbiles como celulares y pdas. ASP.NET por su parte permite una productividad interesante a costa de limitar el trabajo a la herramienta que provee su fabricante y quedando atado a las futuras decisiones de la compañía.

El futuro próximo parece marcado por las interfaces ricas, una interactividad sin necesidad de realizar consultas permanentes al servidor ni realizar recargas completas con la posibilidad que desde el cliente se puedan acceder a diferentes servidores a la vez, lo que marca un cambio en la concepción del navegador web, que pasa de ser una interfaz sin inteligencia ni estado (salvo por las cookies) a otra con la posibilidad de procesar la información y mantener un estado propio.

Como reflexión final, expresar que lo mejor de conocer varios frameworks es saber que no existe uno que sea la bala de plata y que sea la solución a todos los problemas sino que cada uno fue desarrollado con objetivos diferentes y es necesario ver cual de todos se alinea mejor con los objetivos de nuestro proyecto evaluando ventajas y desventajas de cada uno.

Acerca del Autor

Marcio Degiovannini es argentino de 25 años. Actualmente se desempeña como Líder de Proyecto para la consultora CyS Informática. En el área educativa, desde hace 5 años es docente de la facultad de ingeniería de la Universidad de Buenos Aires y actualmente también forma parte del equipo de instructores del centro de capacitación en tecnologías de objetos ObjectLabs.

Para ponerse en contacto con el autor puede escribir un mail a la dirección
contacto ARR marcio.com.ar

Licencia del artículo

Referencias:

Rolling with Ruby on Rails
Curt Hibbs - http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html - 01/20/2005

Tapestry in Action
Howard M.Lewis Ship - Manning - 2004

The open laszlo platform
http://www.openlaszlo.org/

Tutorial on Laszlo Presentation Server (LPS) Persistant Connection and Backend Agent
Jack Hung - http://jackhung.tripod.com/lpsagent/quoteAgent/main.html - 10/24/2004

JavaServerFaces vs Tapestry - A Head-to-Head Comparison
Phil Zoio - http://www.theserverside.com/articles/article.tss?l=JSFTapestry - Agosto 2005

Introduction to Jackarta Tapestry
Rob Smith - http://www.ociweb.com/jnb/jnbMay2004.html

Cocoon as a web framework
Neal Ford - http://www.theserverside.com/articles/article.tss?l=Cocoon - Marzo 2004

Developing Web Applications with JavaServer Faces
Qusay H. Mahmoud - http://java.sun.com/developer/technicalArticles/GUI/JavaServerFaces/ - August 2004

Java Server Faces FAQ
http://java.sun.com/j2ee/javaserverfaces/faq.html

Java Server Faces vs Struts
Roland Barcia - http://homepage1.nifty.com/algafield/barcia.html - Septiembre de 2004

Using de Validator Framework with struts

Chuck Cavaness - http://www.onjava.com/pub/a/onjava/2002/12/11/jakartastruts.html - 12/11/2002

JavaServer Faces (JSF) Tutorial
http://www.exadel.com/tutorial/jsf/jsftutorial-kickstart.html - 05-27-2005

A first look at JavaServer Faces, Part 1
http://www.javaworld.com/javaworld/jw-11-2002/jw-1129-jsf.html - David Geary

JSF for nonbelievers: The jsf application lifecycle
http://www-128.ibm.com/developerworks/java/library/j-jsf2/

JSF v1.1 Specification Final Release
http://java.sun.com/j2ee/javaserverfaces/download.html

The J2EE 1.4 Tutorial
Eric Armstrong (...) - http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html -Junio 2005

DWR-Direct Web Remotin
http://getahead.ltd.uk/dwr/ - Agosto 2005

Que es ASP.NET
http://asp.programacion.net/

Internationalize toyr ASP.NET Application
Ollie Cornes - http://www.devx.com/dotnet/Article/6997 - Octubre 2002

Microsoft ASP.NET QuickStart Tutorial
http://es.gotdotnet.com/quickstart/aspplus/

Guía básica de ASP.NET
http://support.microsoft.com/?scid=kb;es-es;E305140 - Septiembre 2005

Tiles en Struts
Pedro del Gallego Vida - http://www.javahispano.org/articles.article.action?id=67

Struts, an open source MVC implementation
Malcolm G. Davis - http://www-128.ibm.com/developerworks/java/library/j-struts/#h2 - 01 Feb 2001

Struts in a Nutshell
http://struts.apache.org/struts-doc-1.2.x/index.html

Introducing Apache Cocoon
http://cocoon.apache.org/2.1/introduction.html - 11/18/2005

Introduction of Tapestry
http://jakarta.apache.org/tapestry/QuickStart/index.html

AJAX
http://en.wikipedia.org/wiki/AJAX

AJAX: A new approach to web applications
Jesse James Garrett - http://www.adaptivepath.com/publications/essays/archives/000385.php - Feb 18, 2005