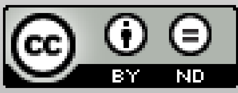
Zope Page Templates

y metal macros

Pablo Ambrosio pandres@menttes.com

Emanuel Sartor emanuel@menttes.com



Función

Zope favorece la separación entre presentación y lógica proveyendo objetos destinados expresamente a la presentación.

Definimos como presentación la tarea de de definir dinámicamente la estructura de las páginas web. Los objetos de presentación generan código HTML.

Zope provee dos métodos para presentación Zope Page Templates (ZPT) y Document Template Markup Language (DTML).

ZPT apunta a directamente a generar código HTML válido insertando código XML para definir el comportamineto dinámico.

DTML tambien permite generar páginas web insertando tags especiales, pero estas no son HTMl válido.

¿Que son?

Los Zope Page Templates son documentos XHTML, lo que significa que pueden ser vistos y editados usando herramientas compatibles con XHTML.

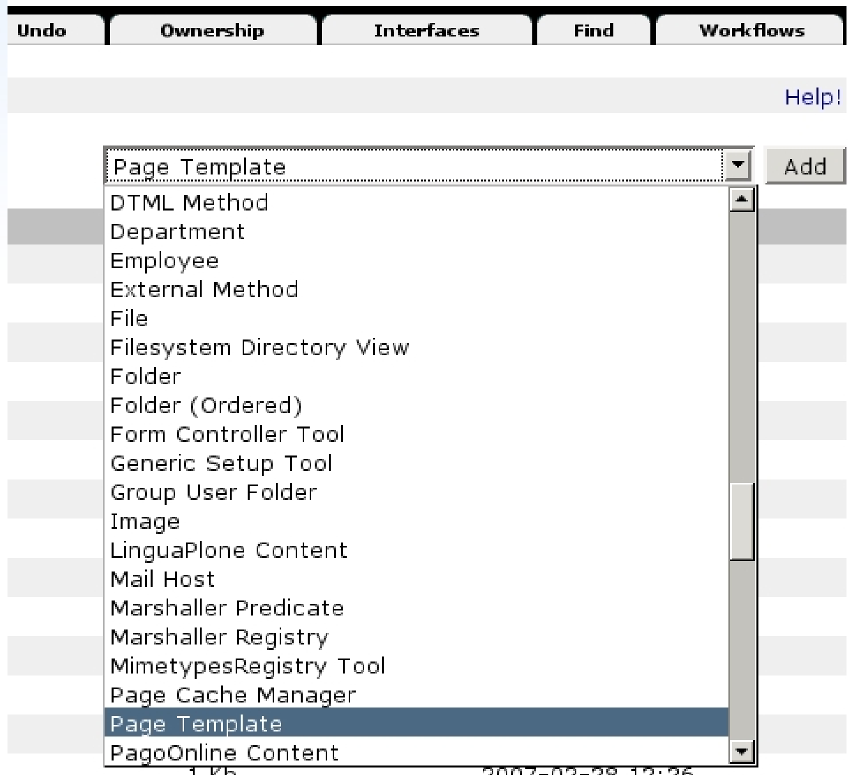
ZPT (Zope Page Templates) esta basado en los lenguajes TAL (Template Attribute Language) y METAL (Macro Expansion Template Attribute Language), ambos son específicos de Zope. estos lenguajes proveen facilidades para extraer elementos de la base de datos y mostrar sus propiedades en HTML.

La implementación de ZPT es la misma tanto para Zope 2 como para Zope 3.

Creando un Page Template

Crear una carpeta “Demo” en el *root.*

Ingesar a “Demo” y crear un *Page Template* eligiendo el item en la lista desplegable de arriba a la derecha.



Elegir el *id* (ej: demo) y hacer click en *Add* o *Add and Edit* para acceder al cuadro de edición.

Editando un Page Template

Luego de hacer *click* en “Add and Edit”, o haciendo *click* sobre un *page template* preexistente en la *folder* donde se esta trabajando ingresamos a la pantalla de edición *web.* Si no se ha ingresado código luego de la creación la interfaz mostrará un esqueleto de código básico sobre el que se puede comenzar a trabajar.

Edit | Test | Properties | History | Undo | Ownership | Interfaces | Security | Cache

Page Template at /demo/testskin/Demo/demoUno Help!

Title

Content-Type text/html

Last Modified 2007-03-05 05:42 PM

Browse HTML source

Expand macros when editing

```
<html>
  <head>
    <title tal:content="template/title">The title</title>
  </head>
  <body>

    <h2><span tal:replace="here/title_or_id">content title or id</span>
        <span tal:condition="template/title"
              tal:replace="template/title">optional template title</span></h2>

    This is Page Template <em tal:content="template/id">template id</em>.
  </body>
</html>
```

Save Changes | Taller | Shorter | Wider | Narrower

You can upload the text for demoUno using the following form. Choose an existing HTML or XML file from your local computer by clicking *browse*. You can also click here to view or download the current text.

Una vez realizados los cambios que se requieran hacer *click* en el botón “Save Changes” para que estos tengan efecto.

Ejercicio: Editar un Page Template

- Crear un nuevo *Page Template.*

- Editarlo, reemplazando el texto por defecto por algún código HTML simple, como el siguiente:

```
<html>
    <body>
        <h1>¡Hola Mundo!</h1>
    </body>
</html>
```

Subiendo un Page Template

Si se quiere editar un template desde un editor externo o simplemente cargar el código del *template* desde el sistema de archivos, desde la página de creación se puede hacer *click* en el botón “Browse“, desde donde elegiremos el archivo a agregar.

Add Page Template

Page Templates allow you to use simple HTML or XML attributes
local file by typing the file name or using the *browse* button.

Id

File Browse...

Add Add and Edit

Subiendo un Page Template

You can upload the text for demoDos using the following form.
can also click here to view or download the current text.

File [ ] Browse...

Encoding utf-8

Upload File

En la parte inferior de la página de edición se puede acceder a la opción de navegar el sistema de archivos desde donde elegiremos de donde tomar el código para el *template,* una vez elegido se debe hacer *click* en “Upload File”.

Ejercicio: Subir un Zope Page Template

Crear con su editor preferido el siguiente archivo demoUno.pt:

```
<html>
  <head>
    <title tal:content="template/title">The title</title>
  </head>
  <body>

    Este es el Page Template <span tal:content="template/id">id</span>.

  </body>
</html>
```

y subirlo al sitio Plone en la carpeta /demo.

Vista de un Page Template

Un *Page Template* puede ser visto haciendo *click* en la pestaña (*tab*) *Test* desde la pantalla de edición del *template.*

Tambien puede ser visto visitando su URL de Zope asociado, ej:

http://misitio/puntodemontaje/miportal/micarpeta/mitemplate .

Algunas propiedades de los Page Template

El mecanismo de XML y HTML de Zope:

- conserva el *template* como código XML bien formado
- intenta ser no invasivo usando atributos registrados como *namespaces* (TAL)
- provee soporte para macros (METAL)
- provee soporte de para internacionalización

Template Attribute Language (TAL)

El **Template Attribute Language (TAL)** es un lenguaje expresado como atributos en las etiquetas (*tags*) HTML.

Las etiquetas tienen la forma:

```
<p tal:comando="expresion">Texto</p>
```

Todas las declaraciones en TAL consisten de atributos en etiquetas cuyos nombres comienzan con “tal:” , y todas tienen valores asociados, que siempre van entre comillas.

Estas etiquetas son código HTML válido, o sea que estos documentos se pueden editar con cualquier editor de HTML.

Template Attribute Language Expression Syntax (TALES)

Las expresiones TALES evalúan código de un tipo específico y devuelven su resultado. Las distíntas expresiones son manejadas por el motor de TALES. En una expresión TALES su tipo se especifica al comienzo. Estas son algunas de las expresiones mas comunes:

- "path: " - Toma una cadena con pinta de URL para evaluarla y recorrerla hasta obtener un objeto. Es la expresión por defecto en el motor de TALES.
- "string: " - Devuelve un *string,* pero tambien puede interpretar un *path* interpolado.
- "python: " - Devuelve el resultado de una expresión *python*.
- "not: " - Si el resultado de la expresion evaluada es un booleano, niega ese resultado.
- "exists: " - Determina si un objeto existe dado un *path*.

Algunas Expresiones Simples

La expresion "template/title" en nuestro ejemplo de edición de *Page Templates* es de tipo *path*. Este es el tipo mas común de expresión. Existen varios otros tipos de expresiones definidas en la especificación de TAL *Expression Syntax* (TALES).

La expresión "template/title" obtiene la propiedad *title* del *template*. Estas son algunas otras expresiones de tipo *path* comunes:

- 'request/URL': EL URL del *request* actual.
- 'user/getUserName': El nombre de *login* del usuario autenticado.
- 'container/objectIds': Una lista de los Ids de los objetos en la misma carpeta que el *template*.

Todos los *path* comienzan con un nombre de variable. Si la variable contiene el valor buscado, podemos deternos alli. Si no, agregamos una barra (/) y el nombre del sub-objeto o propiedad. Habitualmente se recorre un camino de varios sub-objetos hasta llegar al valor deseado.

Insertando Contenido

Para insertar texto dinámico dentro de otro texto, típicamente se usa *tal:replace* en un *tag span*. El comando *replace* reemplaza el *tag* donde esta con el resultado de evaluar su expresión.

Agregemos las siguientes líneas a nuestro template de ejemplo:

```
<br>
El URL es <span tal:replace="request/URL">URL</span>.
```

El *span tag* es estructural, no visual, luego esto lucirá como "El URL es URL." si vemos el código en un navegador. La version renderizada se verá como:

El URL es http://localhost:8080/demoUno.

Insertando Contenido

Hay que tener cuidado cuando se edita de no romper el *tag span* o colocar *tags* de formato como b o *font* dentro de él, ya que estos tambien serían reemplazados.

Si se quiere insertar texto en un *tag* pero dejar el propio *tag* intacto, se puede usar el comando *tal:content*. Para mostrar el título de nuestro ejemplo como la propiedad título del *template,* agregamos las siguientes líneas a nuestro código:

```
<head>
    <title tal:content="template/title">Título</title>
</head>
```

Si abrimos el *tab* "Test" en una nueva ventana podremos ver los cambios.

Estructuras de Repetición

Agregemos un poco mas de contenido a nuestro *template*, por ejemplo, una lista con los objetos que esta en la misma carpeta que nuestro *template*. Escribamos una tabla con filas numeradas por cada objeto y columnas con el id, el metat-type y el título. Agregemos estas líneas al final de nuestro *template*:

```
<table border="1" width="100%">
  <tr>
    <th>#</th><th>Id</th>
    <th>Meta-Type</th><th>Título</th>
  </tr>
  <tr tal:repeat="item container/objectValues">
    <td tal:content="repeat/item/number">#</td>
    <td tal:content="item/getId">Id</td>
    <td tal:content="item/meta_type">Meta-Type</td>
    <td tal:content="item/title_or_id">Titulo</td>
  </tr>
</table>
```

Estructuras de Repetición

El comando *tal:repeat* en la fila de la tabla significa "repetir esta fila para cada item en mi contenedor de la lista de valores de objetos". El comando *repeat* pone los objetos de la lista en la variable item de a uno por vez (a esta variable se le llama la variable de repetición), y hace una copia de la fila usando esa variable. El valor de "item/getId" en cada fila es el Id del objeto para esa fila, lo mismo ocurre con "item/meta_type" e "item/title".

```
<tr tal:repeat="item container/objectValues">
```

Cualquier nombre puede ser usado para la variable de repetición ("item" es solo un ejemplo), siempre que comience con una letra y contenga solo letras, números y guiones bajos (_). La variable de repetición solo esta definida para el *tag* donde se hace la repetición. Si se intenta usar por encima o debajo del *tag tr* ocurrira un error.

```
<td tal:content="item/getId">Id</td>
```

Estructuras de Repetición

La variable de repetición tambien se puede usar para obtener información sobre acerca de la repetición actual. Insertándola delante de la variable predefinida *repeat* en un *path,* se puede acceder a la cuenta de la repeticion desde cero (index), desde uno (number), desde “A” (Letter), y de varias otras formas. Luego la expresión

```
<td tal:content="repeat/item/number">#</td>
```

vale 1 para la primera fila, 2 para la segunda y así sucesivamente.

Dado que un ciclo *tal:repeat* puede ser insertado dentro de otro, puede haber mas de uno activo a la vez. Por eso se usa la expresión “repeat/item/number” en lugar de “repeat/number”, se debe especificar a cual ciclo se hace referencia incluyendo el nombre de la repetición.

Ejercicio

Crear un *page template* con el código de repeticion visto.

Mostrar la página, notar como lista todos los objetos que estan en la misma carpeta que el *template.*

Agregar y borrar objetos de la carpeta y notar como la página refleja estos cambios.

Elementos Condicionales

Usando *Page Templates* podemos consultar dinámicamente nuestro entorno e insertar text en forma selectiva dependiendo de las condiciones. Por ejemplo, podemos mostrar alguna información específica en respuesta a una *cookie:*

```
<p tal:condition="request/cookies/verbose | nothing">
  Información extra.
</p>
```

Este párrafo solo será mostrado en la salida cuando la *cookie* “verbose” este seteada. Ya que la expresión “request/cookies/verbose | nothing” devuelve *true* solo si la *cookie* llamada “verbose” esta seteada.

Elementos Condicionales

Usando la declaración *tal:condition* se pueden chequear toda clase de condiciones. La declaración *tal:condition* deja el *tag* y su contenido si la expresión devuelve el valor T*rue*, y las remueve si el valor es *False*.

Zope considera al número cero, al *string* vacío, a la lista vacía, ya la variable incorporada *nothing* como valores falsos. Cualquier otro valor es verdadero, incluyendo números distíntos de cero y *strings* no vacíos (tambien espacios).

Otro uso común de condiciones es probar una secuencia para ver si es vacía antes de iterar sobre ella.

Ahora veremos como agregar un control en la página de forma que si ls lista de objetos es vacía nop se muestre la tabla.

Elementos Condicionales

Agregemos esto al final de nuestro *Page Template*:

```
<table tal:condition="container/objectValues"
       border="1" width="100%">
  <tr>
    <th>Number</th>
    <th>Id</th>
    <th>Meta-Type</th>
    <th>Title</th>
  </tr>
  <tr tal:repeat="item container/objectValues">
    <td tal:content="repeat/item/number">#</td>
    <td tal:content="item/getId">Id</td>
    <td tal:content="item/meta_type">Meta-Type</td>
    <td tal:content="item/title">Title</td>
  </tr>
</table>
```

Ejercicio: Elementos Condicionales

Agregar tres carpetas llamadas “1”, “2” y “3” a la carpeta “/demo” en donde se encuetran nuestros *templates.* Abrir el *template* “demoUno” y ver la salida que genera usando el *tab* “Test”.

Veremos una tabla parecida a esto:

Number	Id	Meta-Type	Title
1	demoUno	Page Template	demoUno
2	1	Folder	1
3	2	Folder	2
4	3	Folder	3

Notar que si la expresión “container/objectValues” fuera falsa (si no hubiese objectValues por ejemplo), toda la tabla sería omitida.

Ejercicio: cambiar la expresión por “container/1/objectValues” y observar la salida.

Cambiando Atributos

El comando tal:attribute nos permite asignar dinámicamente valores a los atributos del *tag*.

La mayoría sino todos los objetos listados por el *template* tienen una propiedad *icon* que contiene el path al ícono de ese tipo de objeto.

Para mostrar este ícono en la columna *meta-type*, es necesario este *path* en el atributo *src* de un *tag img.*

Cambiando Atributos

Ejercicio: editar en la tabla la celda de la columna *meta-type* del ejemplo anterior con las siguientes líneas:

```
<td><img src="/misc_/OFSP/Folder_icon.gif"
        tal:attributes="src item/icon">
    <span tal:replace="item/meta_type">Meta-Type</span>
</td>
```

La declaración tal:attributes reemplaza el atributo *src* del tag img con el valor de “item/icon”.

Insertando Texto Estructurado

Normalmente las declaraciones tal:replace y tal:content convierten *tags* HTML en el texto donde son insertadas en una forma “escapada” que se muestra en la salida como como texto plano en lugar de HTML. Por ejemplo el caracter '<' se reemplaza por '<'. Si se desea insertar texto como parte de la estructura HTML evitando esta conversión se debe preceder la expresión con la palabra clave *structure.*

Esto es útil para cuando se quiere insertar un fragmento HTMl o XML que esta almacenado en una propiedad o generado por otro objeto Zope. Por ejemplo, podemos tener un item de noticias que contiene código HTML simple (como mostrar partes en negrita o itálica) y se lo quiere preservar cuando se lo inserte en la página de noticias. En ese caso se podría usar algo como:

```
<p tal:repeat="newsItem here/topNews"
   tal:content="structure newsItem">
  Item de noticias con codigo <code>HTML</code>.
</p>
```

Insertando Texto Estructurado

Realemente no importa si el texto reemplazante tiene cóidigo HTML o no, la palabra *structure* solo dice “dejar este texto sin tocar”.

Este compartamiento no es el por defecto ya que la mayoría del texto insertado no contiene HTML, mientras que puede tener caracteres que interfieran con la estructura de la página.

Ejercicio: Probar en un *template* los siguientes fragmentos de código

```
<span tal:content="string:<b>hola</b>”>texto<span>

<span tal:content="structure string:<b>hola</b>”>texto<span>
```

Elementos de Maqueta

Se pueden incluir elementos en la página que sean visibles en el *template* pero no en el texto genereado usando la variable incorporada *nothing* :

```
<tr tal:replace="nothing">
  <td>10213</td><td>Item de ejemplo</td><td>$15.34</td>
</tr>
```

Esto es útil para completar partes de la página que serán pobladas con contenido dinámico. Por ejemplo, una tabla de diez filas en el template tiene solo una (combinada en un *tal:repeat*). Agregando nueve filas *dummy* la estructura del *template* va a semejarse mucho al resultado final.

Contenido *Default*

Se pueden dejar los contenidos de un elemento (*tag*) intactos usando la expresión *default* junto con un tal:content o un tal:repeat, por ejemplo:

```
<p tal:content="default">Spam</p>
```

Esto se renderiza como:

```
<p>Spam</p>
```

Esto se usa para incluir contenido por defecto en forma selectiva, como en:

```
<p tal:content="python:here.getFood() or default">Spam</p>
```

Si el método “getFood” evalúa verdaro entonces se mostrará su resultado, si es falso se insertrá la opción por defecto “Spam”.

Repetición Avanzada

Las variables de repetición vienen incorporadas al mecanismo de *template* y proveen información sobre la iteración actual. Son:

- index – índice de la repetición, empezando de cero.
- number – índice (o número) de la repetición, empezando desde uno.
- even – verdadera para iteraciones con índice par (0, 2, 4, ...).
- odd - verdadera para iteraciones con índice impar (1, 3, 5, ...).
- start – verdadera para la repetición inicial (índice 0).
- end – verdadera para la última repetición.
- length – longitud de la secuencia, que es el número total de repeticiones.
- letter/Letter – cuenta las repeticiones con letras en minúscula/mayúscula respectivamente ("a" - "z", "aa" - "az", "ba" - "bz", ...)

Repetición Avanzada

Los contenidos de una variable se repetición se pueden acceder usando expresiones *path* o expresiones Python.

En una expresión *path* que conste de tres partes que sean el nombre *repeat,* el nombre de la variable de repetición, y el nombre de l ainformación que se desea, por ejemplo “repeat/item/start”.

En una expresión Python se usa la notación habitual de diccionarios para obtener la variable *repeat,* luego el acceso a atributos par aobtener la información, por ejemplo, “python:repeat['item'].start”. La razón por la que no se usa “repeat/start” es que los tal:repeat pueden estar anidados, asi que necesitamos una forma de especificar a cual de ellos le estamos pidiendo la información.

Ejercicio: probar distíntas variables en el *template* de pruebas de repetición, por ejemplo, colorear distinto a las iteraciones pares.

Omitiendo Tags

Se puede remover un *tag* con la declaración tal:omit-tag. Esta remueve los *tags* que abren u cierran el elemento, sin afectar su contenido:

```
<b tal:omit-tag=""><i>esto</i> queda</b>
```

Renderiza como: <i>esto</i> queda

Usado así tal:omit-tag funciona casi como un tal:replace="default". Sin embargo, tal:omit-tag tambien puede swer usado con una expresión de verdad, en cuyo caso sólo remueve los *tags* si la expresión evalúa a verdader. Por ejemplo:

```
Comidas: <span tal:repeat="comida comidas">
  <b tal:omit-tag="not:comida/preferida"
     tal:content="comida/nombre">Spam</b>
</span>
```

Esto devuelve una lista de comidas, con el nombre de la preferida en negrita.

Trucos con Repeticiones

A veces se necesita repetir una parte de una página, pero no hay candidato natural para el tag. En es caso se puede elegir un elemento para contener la declaración, pero se quiere evitar que aparezca en la página renderizada. Esto se puede hacer usando un tal:omit-tag :

```
<div tal:repeat="section here/getSections"
     tal:omit-tag="">
  <h4 tal:content="section/title">Título</h4>
  <p tal:content="section/text">contenido</p>
</div>
```

No se trata solo de ahorro de
caractéres en la salida. Incluir los *tags div* puede modificar el aspecto de la página, especialmente si usa plantillas de estilo. La declaración tal:omit-tag desincluirá los *tags* de *div* (el que abre y el que cierra) conservando su contenido.

Trucos con Repeticiones

Un ejemplo de tal:repeat anidado. Cada declaración tiene una variable de repetición con distínto nombre:

```
<table border="1">
  <tr tal:repeat="x python:range(1, 13)">
    <td tal:repeat="y python:range(1, 13)"
        tal:content="python:'%d x %d = %d' % (x, y, x*y)">
        X x Y = Z
    </td>
  </tr>
</table>
```

Ejercicio: ver que linda queda en la pantalla.

Trucos con Repeticiones

Una forma de ordenar las listas es usando la función *sequence.sort.* Acá un ejemplo de como ordenar una lista de objetos por títulos y luego por fecha de modificación:

```
<table tal:define="objects here/objectValues;
                  sort_on python:(('title', 'nocase', 'asc'),
                                  ('bobobase_modification_time', 'cmp', 'desc'));
                  sorted_objects python:sequence.sort(objects, sort_on)">
  <tr tal:repeat="item sorted_objects">
    <td tal:content="item/title">título</td>
    <td tal:content="item/bobobase_modification_time">modificado</td>
  </tr>
</table>
```

En este ejemplo se definió el criterio de ordenación de los argumentes separado para mas prolijidad. La función *sequence.sort* toma una secuencia y una descripción de como ordenarla, en este caso la descripción esta definida en la variable “sort_on”.

Control de Atributos Avanzado

La declaracion tal:attributes se puede usar para reemplazar dinámicamente atributos. Se puede reemplazar mas de un atributo a la vez separándolos con una “;”.

```
<a href="link" tal:attributes="href here/getLink;
                               class here/getClass">link</a>
```

Ejercicio: Insertar el siguiente código en un *template* y probar el resultado. *“here”* hace referencia al lugar desde donde es llamado el *template*.

```
<a tal:attributes="href here/absolute_url">enlace</a>
```

Adquisición y navegación de URLs

Zope separa un URL y lo compara con la jerarquía de objetos, recorriéndola hasta que encuentra una coincidencia para cada parte. Este proceso se conoce como caminata de URL. Por ejemplo, cuando Zope recibe el URL “Zoo/Animales/leon/alimentar”, comienza desde el directorio raíz buscando un objeto llamado Zoo. Luego se mueve a la carpeta Zoo y busca un objeto llamado Animales. Se mueve a la carpeta Animales y busca un objeto llamado leon. Se mueve dentro de leon y busca el objeto llamado alimentar. Supongamos que el *script* alimentar no se encuentra en el objeto leon y es encontrado en la carpeta Zoo usando el mecanismo de adquisicion. Zope siempre comienza a buscar un objeto comenzando por el último objeto recorrido, en este caso leon, Como leon no contiene nada, Zope vuelve al contenedor de leon, Animales. El *script* alimentar no esta allí, asi que Zope vuelve un paso mas en la jerarquía hasta el contenedor Zoo, donde encuentra el *script*.

Adquisición y navegación de URLs

Ahora Zope ha alcanzado el final del URL y ha encontrado coincidencias de objetos para los elementos del URL. Zope reconoce que el último objeto hallado, alimentar, se puede llamar (es una función o método), y lo llama en el contexto del anteúltimo objeto encotnrado, leon. Asi es como el script alimentar es invocado para el objeto leon.

Asi mismo se podría llamar al método bañar en el objeto leon visitando el URL "Zoo/Animales/leon/bañar". Este método podría adquirirlo de la carpeta Animales.

Adquisición y Navegación de URLs

La adquisición un asunto de contenedores

El concepto detras de la adquisición es:

- Los objetos estan situados dentro de otros objetos. Estos objetos actúan como sus “contenedores”.
- Los objetos pueden adquirir el comportamiento de sus contenedores.

Ejercicio:

• Correr los *templates de* prueba desde los URLs de “/demo” y de “/demo/1” y observar las diferencias.

• En el template de repetición cambiar “container/objectValues” por “here/objectValues” y correrlo en las carpetas indicadas arriba.

Definiendo Variables

Podemos definir nuestras propias variables usando el atributo tal:define. Un uso común de esto es evitar escribir repetidamente algunas expresiones en la página. Otra es hacer una llamada a un método computacionalmente caro una sola vez dentro de la página (y almacenar su valor para usos posteriores). Las variables definidas dentro de un *tag* pueden usarse todas las veces que sea necesario siempre que estén en elementos abarcados por el *tag* donde se definió. En este ejemplo se define una variable para luego testearla e iterar sobre ella:

```
<ul tal:define="items container/objectIds"
    tal:condition="items">
    <li tal:repeat="item items">
        <p tal:content="item">id</p>
    </li>
</ul>
```

Ejercicio: evaluar las diferencias contra el ejemplo de tal:repeat.

Definiendo Variables

Como vimos la variable definida sólo está disponible en el *tag* o elementos abarcados por este. Si ponemos la palabra clave *global* delante del nombre de la variable podemos hacer que la variable dure desde donde se la define hasta el final del *template*:

```
<span tal:define="global items container/objectIds"></span>
<h4 tal:condition="not:items">No hay items</h4>
```

Se pueden definir tantas variables como se quiera en un tal:define separándolas con un “;”. Cada variable puede tener su propio rango local o global. También se puede referenciar una variable definida previamente en la misma definición. For example:

```
<p tal:define="title template/title;
        global containerTitle here/title;
        tlen python:len(title);">
```

Manejo de Errores

Si ocurre un error en nuestro *page template*, podemos atrapar ese error y mostrar un mensaje útil al usuario. Supongamos que definimos una variable usando información de un *form:*

```
<span tal:define="global prefs request/form/prefs"  tal:omit-tag="" />
```

Si Zope encuentra un problema, como que la variable buscada no existe en el *form*, la página se rompería y se mostraría una página de error en su lugar. Se puede evitar ese comportamiento con la funcionalidad tal:on-error, que provee manejo de errores limitado:

```
<span tal:define="global prefs here/scriptToGetPreferences"
    tal:omit-tag=""
    tal:on-error="string:Ha ocurrido un error con las preferencias">
```

Manejo de Errores

Cuando ocurre un error Zope busca una declaración tal:on-error en el elemento actual, si no lo encuentra sigue con el inmediatamente superior y asi. Cuando encuentra un manejador de errores reemplaza el contenido de ese elemento con la expresión del manejador (en el ejemplo anterior, un mensaje de error).

Típicamente se define un manejador en un elemento que encierra una expresión lógica, por ejemplo una tabla.

Para un manejo de errores mas flexible se puede llamar a un *script*. Por ejemplo:

```
<div tal:on-error="structure here/capturarError">
...
</div>
```

Notar el uso de la palabra clave *structure*.

Manejo de Errores

El script puede examinar el error y tomar distíntas acciones dependiendo de su valor. El script tiene acceso al error a traves de la variable error definida predefinida. Por ejemplo (el script debe recibir los valores “errType”, ”errValue”):

```
if errType==ZeroDivisionError:
    return "<p>Can't divide by zero.</p>"
else:
    return """<p>An error occurred.</p>
            <p>Error type: %s</p>
            <p>Error value: %s</p>""" % (errType, errValue)
```

El *script* puede tomar varias acciones, por ejemplo, puede logear el error o mandarlo por *email.* La declaración tal:on-error no es de propósito general, si se desea hacer validación de *forms* se recomienda otras herramientas, como los “Controller Page Templates”.

Interacciones Entre Comandos TAL

Cuando hay una declaración TAL por elemento, el orden en el que se ejecutan es simple. Comenzando desde la raíz, la declaración de cada elemento es ejecutada y luego la de sus elementos hijos siguiendo el orden en que aparecen.

Sin embargo, es posible tener mas una declaración TAL en un mismo elemento. Cualquier combinación de declaraciones puede aparecer en un mismo elemento, con la excepción de tal:content y tal:replace, que nunca pueden estar juntos.

Interacciones Entre Comandos TAL

Cuando un elemento tiene múltiples declaaciones, se ejecutan en el siguiente orden:

1. define
2. condition
3. repeat
4. content *o* replace
5. attributes
6. omit-tag

Dado que la declaración tal:on-error sólo se invoca si ocurre un error, no aparece en la lista.

Interacciones Entre Comandos TAL

El razonamiento en este orden es algo así: a menudo se desea asignar variables para su uso en otras declaraciones, asi que *define* va primero. Lo próximo por hacer es decidir si el elemento se incluirá o no, así que sige el condicional; dado que la condición puede depender de las variables seteadas. Puede ser útil poder reemplazar las distíntas partes de un elemento con valores en cada iteración de un *repeat,* asi que éste viene antes de *content*, *replace* y *attributes*. *Content* y *replace* no pueden ser usados juntos así que ocupan el mismo lugar en la jerarquía. *Omit-tag* viene último dado que ningúna otra declaración depende de ella y que debe aparecer luego de *define* y de *repeat*.

Interacciones Entre Comandos TAL

Hay tres restricciones de las que hay que estar conciente al combinar declaraciones TAL en elementos:

- Solo un tipo de cada declaración puede ser usada en cada *tag*. Dado que HTML no permite atributos múltiples con el mismo nombre.
- Tanto tal:content como tal:replace no pueden ser usadas en el mismo *tag*, dado que sus funcionalidades conflictúan.
- El orden en el que se presentan los atributos TAL en un *tag* no afecta su orden de ejecución.

Expresiones

Las expresiones proveen valores a las declaraciones TAL, y solo pueden funcionar dentro de ellas.

Las variables son nombres que pueden ser usados en las expresiones. Los *Page Templates* proveen variables predefinidas. Estas son:

- nothing: es un valor falso, similar al *string* vacío. Se puede usar en un tal:replace o tal:content para borrar un elemento o su contenido. Si un atributo es definido como *nothing*, es removido del tag, mientras que un *string* vacío inserta el *tag* con valor vacío.

- default: es un valor especial que no cambia nada cuando es usado con tal:replace, tal:content o tal:attributes. Dejando el texto en su lugar.

Expresiones

- options: los argumentos que han sido pasados al template por teclado (si los hay). No se puede usar a traves de la *web.* Solo está disponible si se llama desde Python, por ejemplo, si el template “t” es llamado desde una expresión Python t(foo=1), el *path* “options/foo” equivale a 1.
- attrs: un diccionario de atributos del actual tag en el *template.* Las claves son los nombres de los atributos y los valores son los orginales de los atributos. No es una variable muy usada.
- root: el objeto raíz de Zope. Se usa para obtener objetos Zope desde distíntas ubicaciones, sin importar desde donde se este llamando al *template*.

Expresiones

- here: el objeto desde donde el template esta siendo llamado. A menudo esto es lo mismo que container, pero puede variar si se esta usando adquisición. Se usa para obtener objetos Zope que se espera econtrar en distíntos lugares dependiendo de donde se llame el *template*. Es igual a la variable *context* que se usa en scripts Python.

- container: el contenedor (usualmente una carpeta) en donde se encuentra el *template.* Se usa para obtener objetos Zope desde ubicaciones relativas a la posción del *template*. Las variables *container* y *here* se refieren al mismo objeto cuando un *template* es llamado desde su ubicación normal. Sin embargo cuando el *template* es aplicado a otro objeto (por ejemplo en un método ZSQL) ambas variables se referirán a distíntos objetos.

- modules: la colección de módulos Python disponibles en los *templates*.

Expresiones Path

Las expresiones *path* se refieren a objetos con un camino reflejado en un *path* de URL. Describen una caminata desde un objeto hacia otro. Todos los caminos comienzan con un objeto conocido (como una variable predefinida, incorporada en el *framework* o definida por el usuario) y parten de alli hacia el objeto buscado. Algunos ejemplos:

template/title

container/notas/objectValues

user/getUserName

container/master.html/macros/header

request/form/address

root/standard_look_and_feel.html

Las caminatas se pueden hacer desde un objeto hacia sus subojetos incluyendo propiedades y métodos. Se puede usar adquisición en las expresiones *path*.

Zope restringe las caminatas de la misma forma que restringe el acceso a objetos. Es necesario tener los permisos adecuados.

Path Alternativos

La existencia del camino “template/file” está garantizada cada vez que se usa el *template*, aunque su valor sea el *string* vacío. Algunos caminos, como “request/form”x”, pueden no existir durante la renderización del *template*. Normalmente esto causa un error cuando Zope evalúa la expresión *path*.

Ante la situación de que un camino no exista conviene tener un camino alternativo o valor para usar en su lugar. Por ejemplo, si “request/form/x” no existe, se podría usar “here/x” en su lugar. Esto se puede hacer listando los caminos por orden de preferencia y separándolos con una barra vertical “|”:

```
<h4 tal:content="request/form/x | here/x">Header</h4>
```

Dos variables que son útiles como alternativas para *path* son *nothing* y *default*.

Path Alternativos

Como parte final de de una expresión *path* alternativa tambien se puede usar una expresión que no sea de tipo *path*:

```
<p tal:content="request/form/edad|python:18">edad</p>
```

En este ejemplo, si el camino “request/form/age” no existe, el valor es el número 18. Esta *form* permite especificar valores por defecto que no pueden ser expresados como caminos. Notar que solo se puede usar una expresión de otro tipo en la última alternativa.

También se puede probar directamente la existencia de un camino prefijando con una expresión de tipo *exists*.

Expresiones String

Las expresiones *string* permiten combinar fácilmente expresiones *path* y texto. Todo el texto que aparece luego del prefijo “string:” es chequeado en busca de expresiones *path*. Cada expresión *path* debe estar precedida por un signo de dolar '$'. Algunos ejemplos:

"string:Solo texto. No hay un path aqui.."

"string:copyright $year por Gomez Bolaños."

Si la expresión path tiene mas de una parte (si contiene una barra '/'), o necesita ser separada del texto quie la sigue, debe ser encerrada por corchetes '{}'. Ejemplo:

"string:Tres ${manzana}s por favor."

"string:Su nombre es ${user/getUserName}."

Expresiones String

Dado que el texto esta dentro del valor de un atributo, solo se puede incluir una comilla doble usando la entidad sintáctica '"' . Como el símbolo dolar se usa para para marcar una expresión de tipo *path,* un símbolo de dolar literal debe ser escrito como dos símbolos de dolar '$$'. Unos ejemplos:

"string:Favor de pagar $$$pesos_debidos"
"string:Ella dijo, "Hola gente.""

Las operaciones de formateao de *string* complejas, como buscar y reemplazar o cambiar a mayúsculas, no puede ser hecha fácilmente con expresiones *string*. Para esos casos se deben usar expresiones Python o *scripts*.

Ejercicio: Probar distíntas combinaciones de expresiones *string* en *templates* de prueba.

Expresiones Not

Una expresión not permite negar el valor de otra expresión:

```
<p tal:condition="not:here/objectIds">
  No hay objetos contenidos.
</p>
```

Las expresiones *not* devuelven el valor verdadero cuando la expresión a la que están aplicados devuelve falso y viceversa. En Zope, cero, el *string* vaciío, las secuencias vacías, *nothing* y *None* se consideran falsas, mientras que todo lo demas se considera verdadero. Las expresiones *path* no existentes no tienen valor de verdad y la aplicación de un 'not:' sobre ellas fallará.

Expresiones Nocall

Una expresión *path* común intenta renderizar el objeto que invoca. Esto significa que si el objeto es una función, *script*, método, o alguna otra cosa ejecutable, la expresión va a evaluar al resultado del objeto llamado. Esto es lo que se desea usualmente, pero no siempre. Supongamos que se quiere definir una variable para contener una función o clase de un módulo (de Python), para ser usada en una expresión Python. Podemos usar la expresión *nocall* para esto:

```
<p tal:define="join nocall:modules/string/join">
```

Esta expresión define la variable *join* como una función (*string.join*), en vez de ser el resultado de llamar a la función.

Las expresiones pueden ser usadas tanto en objetos como en funciones.

Expresiones Exists

Una expresión *exists* es verdadera cuando su *path* asociado existe. La siguiente es una forma de mostrar un mensaje de error solo si éste es pasado en el *request*:

```
<h4 tal:define="err request/form/errmsg | nothing"
    tal:condition="err"
    tal:content="err">¡Error!</h4>
```

Se puede lograr lo mismo mas fácilmente con una expresión *exists*:

```
<h4 tal:condition="exists:request/form/errmsg"
      tal:content="request/form/errmsg">¡Error!</h4>
```

Expresiones Exists

Se pueden combinar expresiones exists con expresiones not, por ejemplo:

```
<p tal:condition="not:exists:request/form/number">
Por favor ingrese un número entre 0 y 5</p>
```

Notar que en este ejemplo no se puede usar la expresión "not:request/form/number", dado que esa expresión será verdadera si la variable “number' existe y es distínta de cero.

Ejercicio: probar la expresión *exists* con combinaciones con la expresión *not* en un *template* de prueba.

Expresiones Python

Una expresión Python puede contener cualquier cosa que el lenguaje Python considere una expresión. No se pueden usara declaraciones como *if* o *while.* Ademas, Zope impone algunas restricciones de seguridad para evitar el acceso a información protegida o crear problemas como *loops* infinitos.

Expresiones Python: Comparaciones

Un lugar donde las expresiones Python son necesarias es en las declaraciones tal:condiction. Usualmente se quiere comparar entre dos strings o dos números, y no hay soporte en las expresiones TAL para hacer esto sin expresiones Python. Se pueden usar los operadores de comparación < (menor que), > (mayor que), == (igual que), != (distítno que). Tambien se pueden usar los operadores booleanos *and*, *not* y *or*. Ejemplo:

```
<p tal:repeat="widget widgets">
  <span tal:condition="python:widget.type == 'gear'">
  Gear #<span tal:replace="repeat/widget/number>1</span>:
  <span tal:replace="widget/name">Name</span>
  </span>
</p>
```

Este ejemplo itera sobre una colección de objetos, imprimiendo información acerca de los *widgets* que son de tipo *gear*.

Expresiones Python: Comparaciones

A veces se quiere elegir entre diferentes valores dentro de una única declaración basándose en una o mas condiciones. Esto se puede hacer con la función *test* de la siguiente manera:

```
Usted <span tal:define="name user/getUserName"
     tal:replace="python:test(name=='Anonymous User',
                                          'necesita logearse', default)">
      está logeado como
      <span tal:replace="name">Nombre</span>
    </span>
```

Si el usuario es Anonymous, luego el elemento span es reemplazado con el texto “necesita logearse”. De otra forma el contenido por defecto es usado.

Expresiones Python: Comparaciones

La función test funciona como un “if/then/else”. Este es otro ejemplo de como usar la función test:

```
<tr tal:define="oddrow repeat/item/odd"
    tal:attributes="class python:test(oddrow, 'oddclass',
                                      'evenclass')">
```

Esto asigna al atributo class los valores 'oddclass' y 'evenclass' alternadamente a las filas de la tabla, permitiendo dar distíntos estilos a la salida HTML.

Sin la función *test* necesitaríamos escribir dos elementos *tr* con diferentes condiciones, una para columnas pares y otra para impares.

Expresiones Python: Otros Tipos De Expresiones

Se pueden usar otros tipos de expresiones dentro de una expresión Python. Cada tipo expresión tiene una función correspondiente con el mismo nombre, incluyendo: *path(), string(), exists(), and nocall()*. Esto permite escribir expresiones como:

```
"python:path('here/%s/thing' % foldername)"
"python:path(string('here/$foldername/thing'))"
"python:path('request/form/x') or default"
```

El último ejemplo tiene un significado levemente distínto a la expresión *path* "request/form/x | default", dado que usa el texto *default* si "request/form/x" no existe o si es falso.

Expresiones Python: Objetos Zope

Buena parte del potencial de Zope involucra lograr que interactúen objetos especializados. Nuestros *Page Templates* pueden usar *scripts*, métodos SQL, catálogos y objetos de contenido customizados. Para poder usar todos estos objetos tenemos que saber como accederlos dentro de los *Page Templates*.

Las propiedades de los objetos son atributos, luego podemos acceder el título de un template con la expresión "template.title". La mayoría de los objetos Zope soportan adquisición, lo que nos permite obtener atributos desde objetos “padre” (anteriores en la jerarquía). Esto significa que la expresión “here.Control_Panel” traerá el objeto “Control Panel” desde la carpeta raíz. Los métodos de los objetos son atributos, como “here.objectIds" y "request.set".

Expresiones Python: Objetos Zope

Los objetos contenidos en una carpeta pueden ser accedidos como atributos de la carpeta, pero como a menudo tienen Ids que no son identificadores Python válidos, no se puede usar la notación habitual.

Por ejemplo, no se puede acceder el objeto “penguin.gif” con la siguiente expresión:

"python:here.penguin.gif"

en su lugar se debe usar:

"python:getattr(here, 'penguin.gif')"

dado que Python no reconoce nombres de atributos con puntos '.' en ellos.

Expresiones Python: Objetos Zope

Algunos objetos, como request, módulos, y carpetas Zope proveen soporte para acceso de items a la Python, por ejemplo:

```
request['URL']
modules['math']
here['thing']
```

Cuando se usa el acceso de items en una carpeta, no intenta usar adquisición sobre el nombre, luego solo tendrá éxito en la búsqueda si hay un objeto con ese identificador en la carpeta.

Usando Scripts

Los objetos *script* se usan para encapsular lógica de negocios y manejo de datos complejos. Cada vez que nos encontremos escribiendo muchas declaraciones TAL con expresiones complicadas en ellas, debemos considerar usar un *script*. De esta forma se pueden mantener páginas mas simples y prolijas.

Cada *script* tiene una lista de parámetros que espera que se le de cuando es llamado. Si esta lista es vacía, entonces se puede llamar al *script* usando una expresión *path*. De lo contrario, se deberá usar una expresión Python para poder pasarle los parámetros:

```
"python:here.myscript(1, 2)"
"python:here.myscript('arg', foo=request.form['x'])"
```

Usando Scripts

Si se desea devolver mas de un dato desde un *script* a un *Page Template*, es buena idea hacerlo con un diccionario. De esa forma, se puede definir un avariable para contener todos los datos, y usar expresiones *path* para referirse a cada uno de ellos. Por ejemplo, supongamos que el *script* getPersona devuelve un diccionario con las claves nombre y edad:

```
<span tal:define="persona here/getPersona"
      tal:replace="string:${persona/nombre} tiene ${person/edad}">
</span> años..
```

Por supuesto, tambien se puede devolver un obleto Zope o una lista Python.

Ejercicio: Hacer dos *script* simples (que devuelvan un string por ejemplo), uno que reciba parámetros y otro que no.

Módulos Python

El lenguaje de programación Python tiene una gran cantidad de módulos, que proveen mucha variedad de funcionalidades para los programas Python. Cada módulo es una colección de funciones Python, datos, y clases relacionadas con un mismo propósito, como cálculos matemáticos o expresiones regulares.

Varios módulos, incluyendo “math” y “string”, están disponibles para expresiones Python por defecto. Por ejemplo, se puede optener el valor de pi desde el módulo “math” escribiendo “python:math.pi”. Para accederlo desde una expresión path se usa la variable *modules*, “modules/math/pi”.

Módulos Python

El módulo “string” está escondido en las expresiones Python por la función del tipo de expresión “string”, asi que para acceder al módulo se debe usar la variable *modules.* Esto se puede hacer directamente en la expresión en la que se use, o defieniéndo una variable global para ello, como en:

```
tal:define="global mstring modules/string"
tal:replace="python:mstring.join(slist, ':')"
```

Macros

Hasta ahora hemos visto como los p*age* t*emplates* pueden ser usados para agregar contenido dinámico en páginas individuales. Otra característica de los *page templates* es la habilidad para reusar elementos a traves de varias páginas.

Por ejemplo, con *page templates* se puede conseguir un "look and feel" estándar para un sitio. Sin importar el contenido de la página, se puede conseguir que tenga un encabezado, barra lateral, pié de página, y otros elementos estándares. Este es un requerimiento muy común para sitios *web*.

Se pueden reusar elementos de presentación en distíntas páginas usando macros. Los macros definen una sección de la página que puede ser reusada en otras páginas. Un macro puede ser una página completa, o solo un tramo de ella, como el encabezado. Luego de definidos uno o mas macros en un *page template*, se los puede usar desde otro *page template*.

Usando Macros

Se pueden definir macros con atributos en *tags* de forma similar a las declaraciones TAL. Estos atributos son llamados declaraciones METAL, por “Macro Expansion Tag Attribute Language”. Este es un ejemplo de una definición de un macro:

```
<p metal:define-macro="copyright">
  Copyright 2001, <em>Foo, Bar, y Asociados</em> S.A..
</p>
```

Esta declaración metal:define-macro define un macro llamado "copyright". El macro consiste de un elemento p (incluídos los elementos contenidos).

Usando Macros

Los Macros definidos en un *page template* son almacenados en los atibutos macro del *template*. Se puede usar esos macros desde otros *page templates* refiriéndose a ellos a traves de los atributos macro que están en la *page template* donde fueron definidos. Por ejemplo, supongamos que el macro “copyright” está en un *page template* llamado “master_page”. Así es como se usaría ese macro desde otro pa*ge template*:

```
<b metal:use-macro="container/master_page/macros/copyright">
  Aca va el macro
</b>
```

En el siguiente page template, el elemento b será reemplazado completamente por el macro cuando Zope renderice la página:

```
<p>
  Copyright 2001, <em>Foo, Bar, y Asociados</em> S.A.
</p>
```

Usando Macros

Si se cambia el macro (por ejemplo, si el contenedor del copyright cambia) entonces todos los *page templates* que usan el macro reflejarán automáticamente el cambio.

Notar como el macro está identificado por una expresión *path* usando la declaración "metal:use-macro". La declaración "metal:use-macro" reemplaza el macro de la invocador por el macro invocado.

Ejercicio: crear un *template* declarando un macro y otro *template* que use ese macro.

Detalles de los Macros

Las declaraciones metal:define-macro y metal:use-macro son bastante simples. Sin embargo, hay algunas sutilezas que cabe mencionar.

El nombre de un macro debe ser único dentro del *page template* en el que es definido. Se pueden definir varios macros en un mismo documento pero todos deben tener distíntos nombres.

Normalemente nos referiremos a un macro en una declaración metal:use-macro con una expresión *path*. Sin embargo, se puede usar una expresión de cualquier tipo mientras que ésta devuelva un macro. Por ejemplo:

```
<p metal:use-macro="python:here.getMacro()">
  Reemplazado por un macro determinado dinámicamente,
  que está ubicado en el script getMacro.
</p>
```

Detalles de los Macros

Se puede usar la variable *default* con la declaración metal:use-macro:

```
<p metal:use-macro="default">
  Este contenido permanece – ninguna macro es usada
</p>
```

El resultado es el mismo que usar *default* con tal:content o tal:replace. El conteido por defecto del *tag* no cambiará cuando sea renderizado. Esto puede ser útil si se necesita usar un macro condicionalmente o respaldarse en el contenido por defecto si el macro no existe.

Detalles de los macros

Si se intenta usa la variable *nothing* con metal:use-macro se obtendrá un error, dado que *nothing* no es un macro. Si se desea usar *nothing* para incluir condicionalemente un macro, en su lugar se debe enmarcar la declaración metal:use-macro con una declaración tal:condition.

Zope gestiona los macros primero cuando renderiza los *templates*. Luego Zope evalúa las expresiones TAL. Por ejemplo, considérese este macro:

```
<p metal:define-macro="title"
   tal:content="template/title">
  Título del template
</p>
```

Detalles de los Macros

Cuando se use este macro insertará el título del *template* en el que el macro está siendo usado, no el título del *template* en el que el macro está definido. En otras palabras, cuando se usa un macro es como copiar el texto del macro dentro del nuevo *template* y renderizarlo en ese *template*.

Usando Slots

Los macros son mucho mas útiles si se puede sobreescribir parte ellos cuando se los usa. Esto se puede hacer definiendo *slots* en el macro que ser llenados cuando se usa el *template*. Por ejemplo:

```
<div metal:define-macro="sidebar">
  Links
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/products">Productos</a></li>
    <li><a href="/support">Soporte</a></li>
    <li><a href="/contact">Contactenos</a></li>
  </ul>
</div>
```

Usando Slots

Ahora supongamos que queremos incluir alguna información adicional en la barra lateral para algunas páginas. Una forma de lograr esto es con *slots*:

```
<div metal:define-macro="sidebar">
  Links
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/products">Productos</a></li>
    <li><a href="/support">Soporte</a></li>
    <li><a href="/contact">Contactenos</a></li>
  </ul>
  <span metal:define-slot="info_adicional"></span>
</div>
```

Usando Slots

Cuando se usa este macro se puede elegir llenar este *slot* como en:

```
<p metal:use-macro="container/master.html/macros/sidebar">
  <b metal:fill-slot="additional_info">
    No se olvide de mirar en <a href="/specials">especiales</a>.
  </b>
</p>
```

Usando Slots

Cuando este *template* se renderice mostrará la información extra provista en el *slot*:

```
<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/products">Productos</a></li>
  <li><a href="/support">Soporte</a></li>
  <li><a href="/contact">Contactenos</a></li>
</ul>
<b>
  No se olvide de mirar en <a href="/specials">especiales</a>.
</b>
```

Notar como el elemento *span* que define el *slot* es reemplazado por el elemento b que define el *slot*.

Personalizando la Presentación por Defecto

Un uso común de *slots* es proveer la presentación por defecto que se puede personalizar. En el ejemplo anterior la definición del *slot* era un elemento vacío. Sin embargo se puede poner contenido dentro de la definicón. Por ejemplo, consideremos el macro de barra lateral revisado:

```
<div metal:define-macro="sidebar">
  <div metal:define-slot="links">
  Enlaces
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/products">Productos</a></li>
    <li><a href="/support">Soporte</a></li>
    <li><a href="/contact">Contactenos</a></li>
  </ul>
  </div>
  <span metal:define-slot="info_adicional"></span>
</div>
```

Personalizando la Presentación por Defecto

Ahora la barra lateral es completamente personalizable. Se puede llenar el *slot* de “links” para redefinir los enlaces. Si se elige no ocupar el *slot* se devolverá el contenido por defecto.

Combinando TAL y METAL

Se pueden usar declaraciones TAL y METAL en un mismo elemento:

```
<ul metal:define-macro="links"
    tal:repeat="link here/getLinks">
  <li>
    <a href="link url"
      tal:attributes="href link/url"
      tal:content="link/name">nombre del link</a>
  </li>
</ul>
```

En este caso, getLinks es un *script* que arma una lista de objetos (podría usar una consulta a Catalog).

Combinando TAL y METAL

Como las declaraciones METAL son evaluadas antes que las TAL, no hay conflctos. Este ejemplo también es interesante dado que personaliza un macro sin usar *slots*. El macro llama al script getLinks para determinar cuales son los enlaces. Luego se puede cambiar el resultado redefiniendo getLinks para distíntas ubicaciones del sitio.

No siempre es fácil determinar la mejor forma de personalizar un el aspecto de las diferentes partes de un sitio. En general se usan *slots* para sobreescribir elementos de presentación, y se usan *scripts* para proveer contenido dinámicamente. Para los casos donde es difícil determinar la situación, los *script* proveen una solución mas flexible.

Macros de Página Completa

En lugar de usar macros para porciones de presentación compartidas entre páginas, se pueden usar macros para definir páginas completas. Esto es posible con *slots*. Un ejemplo:

```
<html metal:define-macro="mipagina">
  <head>
    <title tal:content="here/title">título</title>
  </head>
  <body>
    <h1 metal:define-slot="headline"
        tal:content="here/title">title</h1>
    <p metal:define-slot="body">
      body
    </p>
    <span metal:define-slot="footer">
      <p>Copyright 2001</p>
    </span>
  </body>
</html>
```

Macros de Página Completa

Luego se puede usar ese macro en templates para distíntos tipos de contenido, o diferentes partes del sitio.

Por ejemplo, este es un template para noticias que puede usar el macro:

```
<html metal:use-macro="container/master/macros/mipagina">

  <h1 metal:fill-slot="headline">
    Prensa:
    <span tal:replace="here/getHeadline">Titular</span>
  </h1>

  <p metal:fill-slot="body"
     tal:content="here/getBody">
    Cuerpo de la noticia aca.
  </p>

</html>
```

Macros de Página Completa

Este template redefine el *slot* “headline” para incluir las palabras "Prensa" y llama al método getHeadline en el objeto actual. Tambien redefine el *slot* “body” llamando al método getBody.

El poder de este enfoque es que ahora se puede cambiar el macro “page” y la página de noticias será automáticamente actualizada. Por ejemplo se podría poner el cuerpo de la página en una tabla y agregar una barra lateral a la izquierda y estos elementos pasarían a formar parte de todo el sitio donde se use este macro.

Ejercicio: crear un *page template* que use el macro “mipagina”, reemplazando contenido en los *slots*.

Ejercicio: Plone provee un template principal con el cual formar las páginas identificar cual es el archivo donde está, partiendo desde “portal_skins”.

Cacheando Templates

Aunque la renderización de *page templates* normalmente es rápìda, a veces no es lo suficientemente rápida. Para páginas accedidas con frecuencia, o páginas que toman mucho tiempo renderizarm, tal vez se prefiera cambiar algo de comportamiento dinámico por velociadad. “Caching” nos deja hacer esto.

Se pueden cachear *page templates* usando un administrador cache de la misma forma que se pueden cachear otros objetos, asociándolos con el administrador de cache. Esto se puede hacer llendo al “cache view” del page template y eligiendo allí un administrador de cache (debe haber uno en el camino de adquisición del template para que se vea en el view, la instalación de Plone agrega un RAM cache y un HTTP cache por defecto). Otra forma es llendo a la vista de asociación (“associate view”) del administrador de cache elegido y localizar allí el *template* deseado.

Cacheando Templates

Este es un ejemplo de como cachear un *page template*. Crearemos un *script* llamado “long” con el siguiente contenido:

```
for i in range(500):
  for j in range(500):
    for k in range(5):
      pass
return 'Done'
```

El propósito de este *script* es hacer notar el tiempo de ejecución. Crearemos un *page template* que use el script, por ejemplo:

```
<html>
  <body>
    <p tal:content="here/long">resultado</p>
  </body>
</html>
```

Cacheando Templates

Cargemos la página, nótese el tiempo que toma en renderizarse. Ahora mejoremos ese tiempo usando cache. Podemos crear un objeto “RAM Cache manager” o usar uno que ya exista, debe estar en la carpeta raíz, en la del *template* o alguna en el camino de adquisición. Comprobemos que se ve en la vista de cache de nuestro *page template* y seleccionémoslo.

Seleccionemos el objeto de administrador cache que hemos creado y hagamos click en “Save Changes”. Click en el link de configuración (settings), por defecto almacenará los objetos durante una hora (3600 segundos). Conviene elegir este número de acuerdo a la aplicación.

Regresemos a la página de veámosla una vez mas. Tomará un tiempo en cargarse pero se guardará en cache. Ahora recargemos la página y notemos el cambio en la velocidad. Se pueden intentar varias recargas comprobando que el objeto está en cache.

Cacheando Templates

Si el page template es modificado será removido del cache. Luego la próxima vez que se renderice volverá a tomar un tiempo, luego de lo cual volverá a estar en cache.

Cachear es una forma simple y poderosa para mejorar el rendimiento. No hace falta saber mucho de ella para lograr resultados. Igualmente puede convenir investigar mas sobre el tema, familiarizándose con los objetos “RAM Cache manager” y “Accellerated HTTP Cache manager”.

Obteniendo Grandes Cantidades de Información por Tandas

Cuando un usuario consulta una base de datos y obtiene un resultados con muchos de items, a menudo es mejor mostrarlo en varias páginas de, digamos, veinte items en lugar de mostrar todo en una sola. Al proceso de separar una lista por tandas de le llama “batching”. Page templates soporta la separación por tandas usando un objeto especial Batch, provisto por el módulo ZTUtils. Un ejemplo:

```
<ul tal:define="lots python:range(100);
                     batch python:modules['ZTUtils'].Batch(lots, size=10, start=0)">
  <li tal:repeat="num batch"
      tal:content="num">0
  </li>
</ul>
```

Obteniendo Grandes Cantidades de Información por Tandas

Este ejemplo muestra una lista con diez elementos (desde el 0 al 9). El objeto Batch corta una lista en tandas. Este ejemplo parte una lista de cien objetos en diez tandas de diez.

Se puede mostrar una tanda distínta de diez items pasando como parámetro un número distínto en *start*:

```
<ul tal:define="lots python:range(100);
            batch python:modules['ZTUtils'].Batch(lots,
                                size=10, start=13)">
```

Esta tanda comenzará desde el item catorce y seguira hasta el item veintitres. Es decir, muestra desde el númerpo 13 hasta lel 22. Notemos entonces que el argumento de inicio de la tanda (start) es el index del primer item. El index comienza desde cero. Python usa índices para referirse a los elementos de una lista.

Obteniendo Grandes Cantidades de Información por Tandas

Normalmente cuando usamos tandas queremos incluir elementos de navegación que nos permitan movernos de una tanda hacia otra. Este es un ejemplo:

```
<html>
 <head>
  <title tal:content="template/title">The title</title>
 </head>
 <body tal:define="empleados here/getEmpleados;
     start python:int(path('request/start | nothing') or 0);
     batch python:modules['ZTUtils'].Batch(empleados,
                                           size=3,
                                           start=start);
     previous python:batch.previous;
     next python:batch.next">
```

Obteniendo Grandes Cantidades de Información por Tandas

```
<p>
  <a tal:condition="previous"
     tal:attributes="href string:${request/URL0}?start:int=${previous/first}"
     href="previous_url">previo</a>
  <a tal:condition="next"
     tal:attributes="href string:${request/URL0}?start:int=${next/first}"
     href="next_url">proximo</a>
</p>

<ul tal:repeat="empleado batch" >
  <li>
    <span tal:replace="empleado/name">Juan</span>
    recibe $<span tal:replace="empleado/salary">100,000</span>
    al año.
  </li>
</ul>

</body>
</html>
```

Obteniendo Grandes Cantidades de Información por Tandas

Copiar este código en un *script* “getEmpleados”:

```
return [  {'name': 'Chris McDonough', 'salary':'5'},
          {'name': 'Guido van Rossum', 'salary': '10'},
          {'name': 'Casey Duncan', 'salary':'20' },
          {'name': 'Andrew Sawyers', 'salary':'30' },
          {'name': 'Evan Simpson', 'salary':'35' },
          {'name': 'Stephanie Hand', 'salary':'40' }, ]
```

Obteniendo Grandes Cantidades de Información por Tandas

Si vemos la declaración tal:define en el elemento *body,* define un conjunto de variables para la tanda. La variable “empleados” es una lista de objetos devuelta por el *script*.

La segunda variable, “start”, es o bien configurada con el valor de “request/start” o a cero si no hay una variable *start* en el *request*. La variable start lleva la cuenta de la posición actual en la lista de empleados. La tanda comienza en la posición especificada por la variable “start”.

Las variables “previous” y “next” se refieren a la pasada y a la próxima tanda respectivamente si existen. Todas estas variables quedan disponibles dentro del elemento *body*.

Obteniendo Grandes Cantidades de Información por Tandas

Veamos los enlaces de navegación. Se crean enlaces para navegar hacia atras y hacia adelante en la sucesión de tandas. La declaración tal:condition primero prueba si existen las respectivas tandas, si es asi, se renderiza el enlace (no aparece de lo contrario).

La declaración tel:attributes crea un enlace a las respectivas tandas. Elenlace es el URL de la página actual (request/URL0) junto con un string de consulta indicando el índice “start” de la tanda.

API de Plone/Zope

A menudo se desean manipular los objetos Plone por medio de scripts de Python, veremos ejemplos de cómo usar la API de Plone/Zope para realizar las siguientes tareas:

* Manipular objetos de contenido.
* Usar el catálogo de Plone.
* Manipular usuarios y grupos.

Manipular objetos de contenido

Llamamos objetos de contenido a los objetos que agregamos a través de la interfaz de Plone para manter el contenido del portal: documentos, imágenes, enlaces, carpetas, eventos, noticias y archivos.

Los objetos de contenido pueden ser manipulados por medio de scripts escritos en Python. Es decir, se pueden crear, modificar y borrar objetos de contenido; puede modificar el estado de un objeto de contenido; y se puede acceder a la información que cada objeto de contenido contiene.

Acceder Objetos De Contenido Desde Un Script

Vamos a crear un objeto de contenido usando la interfaz de Plone y luego lo vamos a acceder desde un script siguiendo los siguientes pasos:

- Agregamos un documento (página) a nuestro sitio Plone cuyo identificador sea mi-documento. En Plone 2.1.1 sólo escribimos el título “Mi documento” y el identificador se genera automáticamente.

Acceder Objetos De Contenido Desde Un Script

- A través del ZMI vamos a portal_skins/custom y agregamos un objeto *script* (Python) cuyo contenido sea:

```
from Products.CMFCore.utils import getToolByName

urltool = getToolByName(context, 'portal_url')

portal = urltool.getPortalObject()

document = getattr(portal, 'mi-documento')

print document.Title()
print document.CookedBody()

return printed
```

Acceder Objetos De Contenido Desde Un Script

En el *script* se observa el uso de las funciones getToolByName y getattr, y de los métodos getPortalObject, Title y CookedBody. Veamos cómo se usa cada uno.

- getToolByName(obj, name, default=[]) Se importa desde un módulo CMF. Devuelve la herramienta (tool) dada por name (string) dentro del objeto obj.

- getattr(object, name) es una función de Zope. Devuelve el valor del atributo nombrado de object. name tiene que ser string. Si name es uno de los atributos de object, el resultado es el valor de ese atributo. Por ejemplo, getattr(x, foobar) es equivalente a x.foobar. Si el atributo no existe, se devuelve default, de otra forma se lanza AttributeError.

Acceder Objetos De Contenido Desde Un Script

- hasattr(object, name) es una función de Zope. Los argumentos son un objeto y un *string*. El resultado es 1 si el *string* es el nombre de alguno de los atributos del objeto, y es 0 en caso contrario. (Esto se implementa llamando a gettattr(object, name) y checando si lanza una excepción o no).

- getPortalObject(self) es un método de la clase Products.CMFCore.URLTool.URLTool que devuelve el objeto Portal.

Modificar El Contenido De Un Documento Por Medio De Un Script

Antes de las sentencias *print* en el *script* del ejemplo anterior, agregamos:

```
document.edit(text_format='html', text='<div>
<b>Cambiamos el texto...</b></div>')
```

Se utiliza el método *edit*, que describimos a continuación:

- edit(self, text_format, text, file='', safety_belt='') es un método de la clase CMFDefaults.Document.Document. text_format puede valer html, structured-text o 'plain'; text es el cuerpo del documento.

Copiar Y Pegar Objetos De Contenido

- Crear una carpeta con identificador mi-carpeta.
- El *script* que se muestra abajo, copia el documento creado en el ejemplo y lo pega dentro de la carpeta recientemente creada:

```
from Products.CMFCore.utils import getToolByName

urltool = getToolByName(context, 'portal_url')

portal = urltool.getPortalObject()

cb_copy_data = portal.manage_copyObjects(['mi-documento'])

folder = getattr(portal, 'mi-carpeta')

folder.manage_pasteObjects(cb_copy_data)
```

Copiar Y Pegar Objetos De Contenido

Aquí vemos la utilización de dos métodos: manage_copyObjects y manage_pasteObjects, ambos se explican a continuación junto con manage_cutObjects.

* manage_copyObjects(self, ids=None, REQUEST=None, RESPONSE=None) es un método de la clase OFS.CopySupport.CopyContainer. Pone una referencia a los objetos nombrados en ids en el portapapeles.

Copiar Y Pegar Objetos De Contenido

- manage_cutObjects(self, ids=None, REQUEST=None) es un método de la clase OFS.CopySupport.CopyContainer.Pone una referencia a los objetos nombrados en ids en el portapapeles.

- manage_pasteObjects(self, cb_copy_data=None, REQUEST=None) es un método de la clase OFS.CopySupport.CopyContainer. Pega los objetos previamente copiados en el objeto actual. Si es llamado desde código Python, se debe pasar el resultado de una llamada previa a manage_cutObjects o manage_copyObjects como primer argumento.

Borrar Objetos De Contenido

El siguiente *script* borra el documento que creamos en el ejemplo:

```
from Products.CMFCore.utils import getToolByName

urltool = getToolByName(context, 'portal_url')

portal = urltool.getPortalObject()

portal.manage_delObjects(['mi-documento'])
```

Aquí usamos el método manage_delObjects:

- manage_delObjects(self, ids=[], REQUEST=None) es un método de la clase CMFPlone.PloneFolder.BasePloneFolder. Elimina todos aquellos elementos cuyos identificadores aparezcan en ids.

Crear Documentos, Carpetas Y Eventos Desde Un Script

```
from Products.CMFCore.utils import getToolByName

urltool = getToolByName(context, 'portal_url')
catalogtool = getToolByName(context, 'portal_catalog')

portal = urltool.getPortalObject()

# Creando un documento
doc = portal.invokeFactory('Document', 'test-doc')
document = getattr(portal, 'test-doc')
document.editMetadata(title='Documento de prueba', description='Esta es una descripción',
subject='')
document.edit(text_format='html', text='<b>Documento de prueba</b>')
```

Crear Documentos, Carpetas Y Eventos Desde Un Script

```
# Creando una carpeta
portal.invokeFactory('Folder', 'test-folder')
folder = getattr(portal, 'test-folder')
folder.setTitle('Mi Carpeta')
folder.setDescription('Esta es la descripción de la carpeta.')
catalogtool.refreshCatalog()

# Creando un evento
folder.invokeFactory('Event', id='event')
event = getattr(folder, 'event')
event.edit(start_date='2003-09-18', end_date='2003-09-19', location='home', description='Este es un evento de prueba.')
event.editMetadata(title = 'Foo', subject='Appointment')
```

Crear Documentos, Carpetas Y Eventos Desde Un Script

- editMetadata(self, obj, allowDiscussion=None, title=None, subject=None, description=None, contributors=None, effective_date=None, expiration_date=None, format=None, language=None, rights=None, **kwargs) definido en la clase CMFPlone.PloneTool.PloneTool

- invokeFactory(self, type_name, id, RESPONSE=None, args, *kw): *type_name* es el nombre del tipo de contenido que se quiere crear. Notar que este es el id del objeto que está en la herramienta *portal_types*, el cual puede no ser el mismo que aparece en la interfaz de usuario de Plone, la cual usa el título del tipo, si está disponible. id es el identificador (short name) del nuevo objeto, el que aparecerá en la URL. invokeFactory también acepta otros parámetros, los cuales se pasan al constructor del objeto, se pueden usar si se conoce el constructor; uno muy común es title.

- refreshCatalog(self, clear=0) reindexa todos los objetos que encuentra.

Cambiar El Estado De Un Documento A Privado

```
from Products.CMFCore.utils import getToolByName

urltool = getToolByName(context, 'portal_url')
portal = urltool.getPortalObject()

document = getattr(portal, 'test-doc')

rev_state = document.portal_workflow.getInfoFor(document, 'review_state', '')

print 'El estado inicial era: ' + rev_state + '\n'

if not review_state in ('rejected', 'retracted', 'private'):
    document.portal_workflow.doActionFor(document, 'hide', comment='')

rev_state = document.portal_workflow.getInfoFor(document, 'review_state', '')

print 'El estado final es: ' + rev_state + '\n'

return printed
```

Cambiar El Estado De Un Documento A Privado

Ejercicio: buscar en el ZMI el portal_workflow, identificar allí plone_workflow y verificar que la acción "hide" va de visible a private.

En este *script* se observa el uso de dos métodos: getInfoFor y doActionFor, veamos en más detalle cada uno de ellos:

•getInfoFor(self, ob, name, default=[], wf_id=None, args, *kw) es un método de la clase CMFCore.WorkflowTool.WorkflowTool. Devuelve una propiedad específica (dada por name), relativa al workflow para un objeto ob.

- doActionFor(self, ob, action, wf_id=None, args, *kw) es un método de la clase CMFCore.WorkflowTool.WorkflowTool. Ejecuta la acción de workflow dada por action para el objeto ob.

Usando El Catálogo De Plone

Plone provee una herramienta, el catálogo, a la cuál se le puede consultar sobre los objetos de contenido dentro del portal.

En esta sección veremos el uso del método searchResults:

```
searchResults(self, REQUEST=None, **kw)
```

Este método está definido en CMFCore.CatalogTool.CatalogTool, hace una llamada ZCatalog.searchResults con argumentos extra que limitan el resultado de lo que al usuario le está permitido ver.

Usando El Catálogo De Plone

Los términos de búsqueda pueden ser pasados en el REQUEST o como argumentos de palabras clave (keywords).

Una consulta de búsqueda consiste en un mapeo de nombres de índices (como aparecen en “portal_catalog”) a parámetros de búsqueda.

Se puede pasar un diccionario a searchResults como la variable REQUEST o se pueden usar palabras clave como argumentos para el método. Ejemplo:

searchResults(title='Uso del Catalogo', creator='Juancito')

es lo mismo que:

searchResults({'title': 'Uso del Catalogo', 'creator': 'Juancito'})

Usando El Catálogo De Plone

Ejercicio: localizar en el “portal_catalog” todos los “portal_type” que pueden ser utilizados como criterios de búsqueda.

Existen algunos índices especiales que pueden ser usados para cambair el comportamiento de la consulta:

- sort_on: especifica sobre cual índice ordernar los resultados.
- sort_order: se puede especificar si se desea orden reverso o descendiente. El comportamiento por defecto es ordenar ascendentemente.

Usando El Catálogo De Plone

El siguiente *script* lista todos los documentos del portal inversamente ordenados por id, notar que portal_type restringe el tipo de los objetos de contenido, y sort_on y sort_order hacen la ordenación propuesta.
Si se quisiera ordenar por título, en Plone 2.1.1, deberíamos pasar el valor sortable_title en el parámetro sort_on:

```
results = context.portal_catalog.searchResults(sort_on='id',
                    portal_type='Document', sort_order='reverse')

print [i.getObject().id for i in results]

return printed
```

Notar que usamos el método getObject puesto que el resultado arrojado por searchResults no son los objetos de contenido en sí, sino los identificadores de los objetos. El método getObject devuelve el objeto de contenido asociado a un item del resultado de una búsqueda.

Usando El Catálogo De Plone

Cada objeto de contenido define cuál de su información es *searchable* (recordar en *Archetypes* esto lo definimos en un campo del *schema*). El parámetro SearchableText busca sobre esa información en cada objeto. El parámetro review_state filtra objetos de acuerdo a su estado:

```
results = context.portal_catalog.searchResults(SearchableText='texto',
                                               review_state='private')

print [i.getObject().Title() for i in results]

return printed
```

Usando El Catálogo De Plone

Los índices que serán usados compo parámetros de búsqueda también pueden ser pasados como atributos de un record object. Estos son:

- query: una secuencia de objetos o un único valor a ser pasado como consulta al índice (obligatorio).
- operator: especifica la combinación de resultados de búsqueda cuando query es una secuencia de valores (opcional, por defecto: “or”).

Usando El Catálogo De Plone

- range: define un rango de búsqueda sobre un índice (opcional, por defecto: no activado). Valores permitidos:
  - min, para buscar los objetos con valores más grandes que el mínimo de los valores pasados en el parámetro query;
  - max, para buscar los objetos con valores más pequeños que el máximo de los valores pasados en el parámetro query;
  - minmax, para buscar los objetos con valores más grandes que el mínimo de los valores pasados en el parámetro query y con valores más pequeños que el máximo de los valores pasados en el parámetro query.

- level: se aplica sólo al índice *Path*. Especifica el nivel de directorio para comenzar la búsqueda (opcional, por defecto: 0)

Usando El Catálogo De Plone

En este *script* se muestra cómo se puede pasar un *record* object al método searchResults. Al parámetro *Date* se le pasa un objeto *record* para consultar por aquellos objetos que fueron creados en una fecha posterior al 30 de noviembre de 2005. El parámetro *Creator* filtra aquellos objetos que no fueron creados por el usuario admin:

```
results =  context.portal_catalog.searchResults(Date={'query':DateTime('2005/11/30'),
                                                'range':'min'}, Creator='admin')

print [i.getObject().Title() for i in results]

return printed
```

Manipulando Usuarios Y Grupos

```
id = 'usuario'
fullname = 'Emanuel Sartor'
password = 'facil'
email = 'emanuel@menttes.com'
roles = ('Manager',)

status=''

props = {'username': id,
         'fullname': fullname,
         'password': password,
         'email': email}
```

Manipulando Usuarios Y Grupos

```
#Agregando un nuevo miembro al portal
try:
     context.portal_registration.addMember(id, password, roles,
                                           domains='',properties=props)
    status+='El usuario '+fullname+' fue agregado exitosamente.\n'
except:
     status+='El usuario '+fullname+' no pudo serr agregado.\n'

print status

return printed
```

Hemos usado la función:

- addMember(self, id, password, roles, domains, properties) Crea un PortalMember y lo devuelve.

Manipulando Usuarios Y Grupos

```
groupname = 'Grupo0'

status=''

#Agregando un nuevo grupo al portal
try:
    context.portal_groups.addGroup(groupname,)
    status+='El grupo '+groupname+' fue agregado exitosamente.\n'
except:
    status+="El grupo no pudo ser agregado\n"

print status

return printed
```

Utilizamos la función:

* addGroup(self, id, roles, groups, args, *kw)

Manipulando Usuarios Y Grupos

Agregaremos el usuario creado al grupo:

```
id = 'usuario'
groupname = 'Grupo0'

status=''

# Asignando el usuario al grupo
try:
    group = context.portal_groups.getGroupById(groupname)
    group.addMember(id)
    status+='El usuario '+id+' fue agregado exitosamente al grupo '+groupname+'.\n'

except:
    status+='El usuario '+id+' no pudo ser agregado al grupo '+groupname+'.\n'

print status
return printed
```

Bibliografía

- The Zope Book (2.6 Edition)
- http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/ZPT.stx
- http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/AdvZPT.stx
- http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/AppendixC.stx
- http://docs.neuroinf.de/PloneBook/ch5.rst
- http://docs.neuroinf.de/PloneBook/ch6.rst
- http://www.ifpeople.net/fairsource/courses/material/apiPlone