Create Page Template Introduction

by OSOE Project.

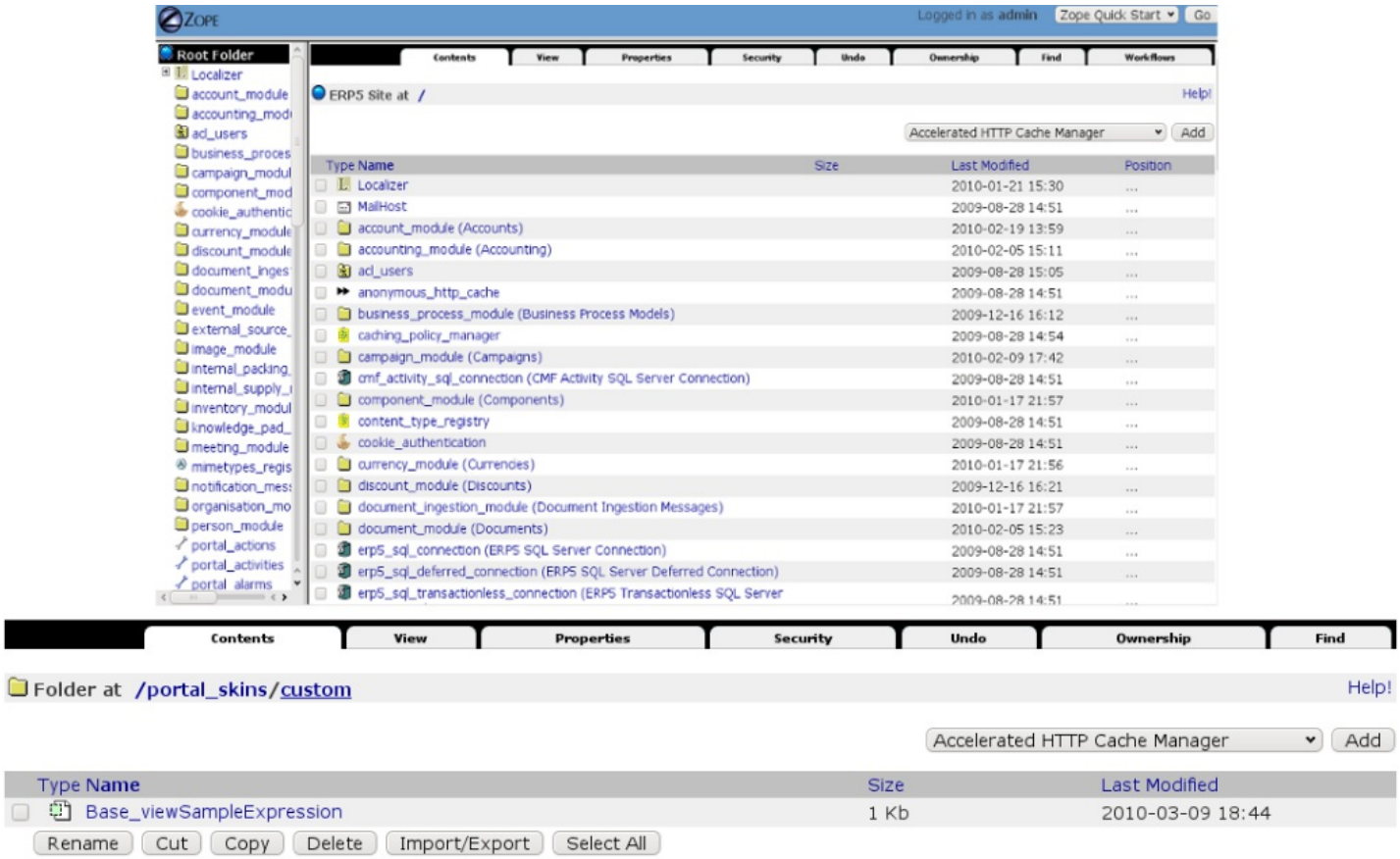
▶ Details

Agenda

- tal:content
- Tal:replace
- tal:block
- tal:repeat
- tal:define
- metal:use-macro

▶ Details

Create a Page Template



▼ Details

First of all you must login as a manager into your instance by using the URL www.tiolive.com/INSTANCCE_ID/manage.

Once you are in the first page of Zope, you can now go in the folder portal_skins/custom in which we are going to create our new page template.

In our example we will create the page template called “Base_viewSampleExpression”. If you want to develop in a proper way under ERP5, you must read the naming convention available at http://www.erp5.org/GuidelinesForNamingConvention

Call a Page Template

▼ Details

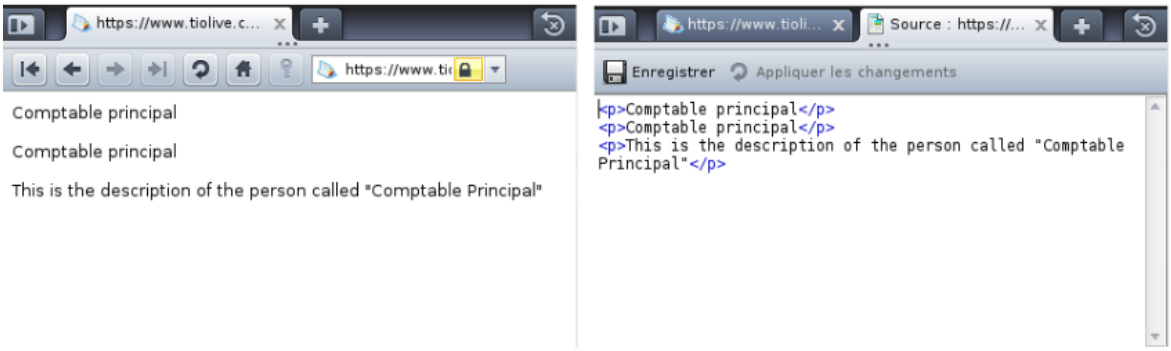
Once you created the new page template, you can already call it, for example on a person document. You can try to call the URL:

https://www.tiolive.com/INSTANCE_ID/person_module/1/Base_viewSampleExpression

You will then see the default rendering of an empty page template. We can now add Tal content to our page template so it will have more informations than only the title of the document.

`tal:content`

```
<p tal:content=" here/getTitle" >Whatever is written</p>
<p tal:content=" python:here.getTitle()" >Whatever is written</p
<p tal:content=" python:here.getDescription()" >Whatever is writ
```

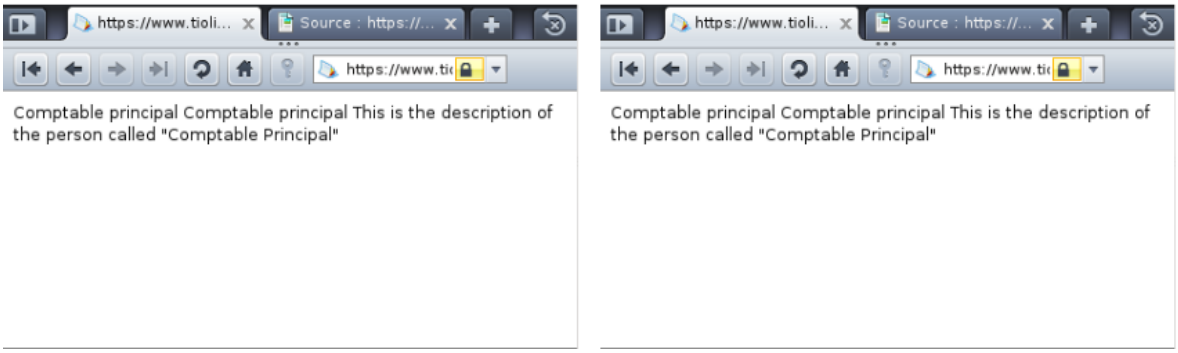


▼ Details

Rather than replacing an entire element, you can insert text or structure in place of its children with the content statement. The statement argument is exactly like that of replace, and is interpreted in the same fashion. If the expression evaluates to nothing, the statement element is left childless. If the action is canceled, then the element's contents are unchanged.

`tal:replace`

```
<p tal:replace=" here/getTitle" >Whatever is written</p>
<p tal:replace=" python:here.getTitle()" />
<p tal:replace=" python:here.getDescription()" />
```



▼ Details

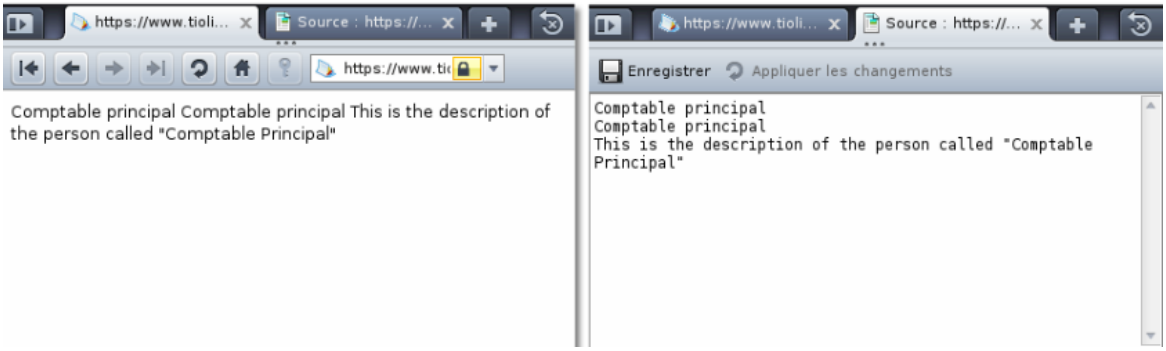
To replace an element with dynamic content, use the replace statement. This replaces the statement element with either text or a structure (unescaped markup). The body of the statement is an expression with an optional type prefix. The value of the expression is converted into an escaped string if you prefix the expression with text or omit the prefix, and is inserted unchanged if you prefix it with structure. Escaping consists of converting "&" to "&", "<" to "<", and ">" to ">".

If the value is nothing, then the element is simply removed. If the action is canceled, then the element is left unchanged (see the TALES default value).

Note: The default replacement behavior is text.

`tal:block`

```
<tal:block tal:replace=" here/getTitle" >Whatever is written</
<tal:block tal:replace=" python:here.getTitle()" />
<tal:block tal:replace=" python:here.getDescription()" />
```

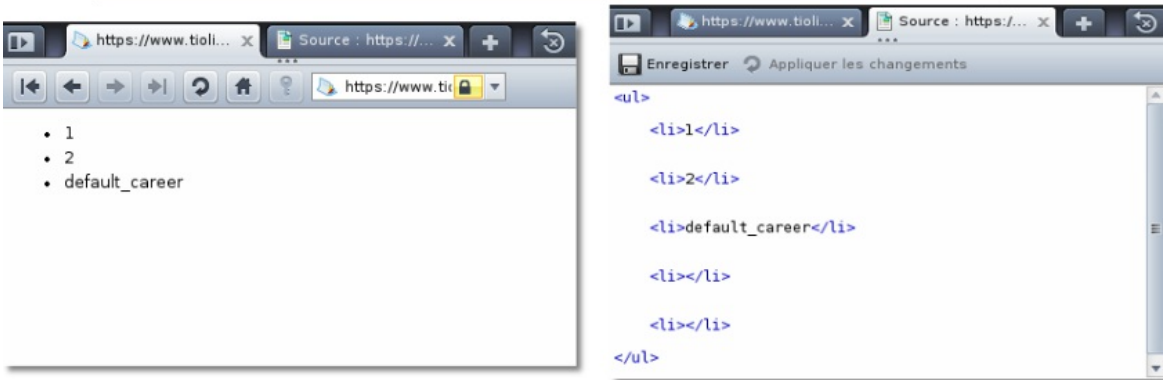


▼ Details

tal:block is a syntactic sugar for elements which contains many TAL attributes which are not to be echoed.

`tal:repeat`

```
<ul>
  <tal:block tal:repeat=" document here/contentValues" >
    <li tal:content=" document/getTitle" >Some Content of List Item</
  </tal:block>
</ul>
```



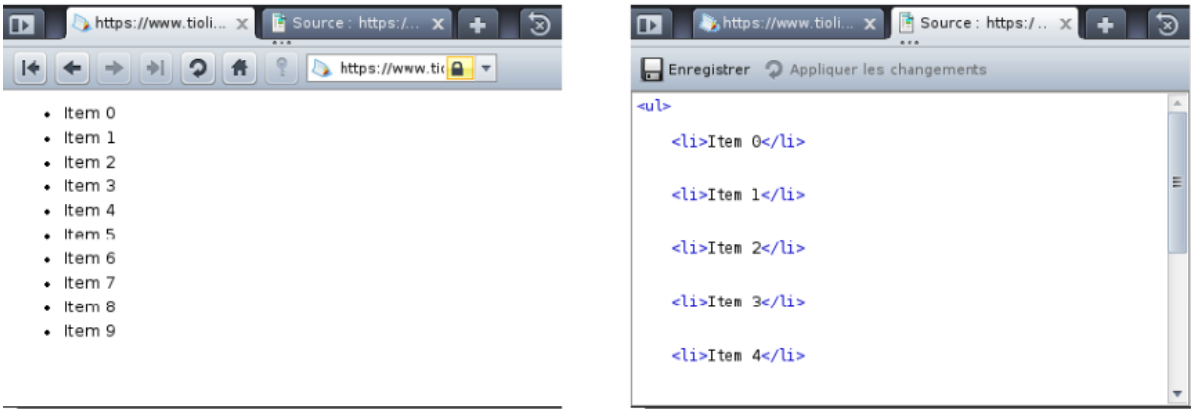
▼ Details

When you want to replicate a subtree of your document once for each item in a sequence, you use repeat. The expression should evaluate to a sequence. If the sequence is empty, then the statement element is deleted, otherwise it is repeated for each value in the sequence. If the action is cancelled, then the element is left unchanged, and no new variables are defined.

The variable_name is used to define a local variable and a repeat variable. For each repetition, the local variable is set to the current sequence element, and the repeat variable is set to an iteration object. You use iteration objects to access information about the current repetition (such as the repeat index). (Iteration objects are more properly the domain of TALES.) The repeat variable has the same name as the local variable, but is only accessible through the built in variable named repeat.

`tal:define`

```
<ul tal:define=" range_list python:range(0, 10);
                 range_name python:'Item'" >
  <tal:block tal:repeat=" list_item range_list" >
    <li><tal:block tal:replace=" range_name" /> <tal:block tal:replace=" list_iter
  </tal:block>
</ul>
```



▼ Details

You can define two different kinds of TAL variables: local and global. When you define a local variable in a statement element, you can only use that variable in that element and the elements it contains. If you redefine a local variable in a contained element, the new definition hides the outer element's definition within the inner element. When you define a global variables, you can use it in any element processed after the defining element. If you redefine a global variable, you replace its definition for the rest of the Template.

If the expression associated with a variable evaluates to nothing, then that variable has the value nothing, and may be used as such in further expressions. If the expression cancels the action, then that variable is not (re)defined. This means that if the variable exists in an enclosing scope, its value is unchanged, but if it does not, it is not created. Each variable definition is independent, so variables may be defined in the same statement in which some variable definitions are canceled.

`metal:use-macro`

```
<p metal:define-macro="foo">Some Text</p>
<h1 metal:use-macro="template/macros/foo">Some Other Text</h1>
```

▼ Details

To define a macro, choose a name for the macro and add a define-macro attribute to a document element with the name as the argument. The element's subtree is the macro body.

To use a macro, first choose the document element that you want to replace. Add a use-macro attribute to it, and set the value of the attribute to an expression that will return a macro definition. This will usually be some kind of path or template id and a macro name. It is important to remember that this expression will be evaluated separately from any TAL commands in the template, so it cannot depend on any such commands.

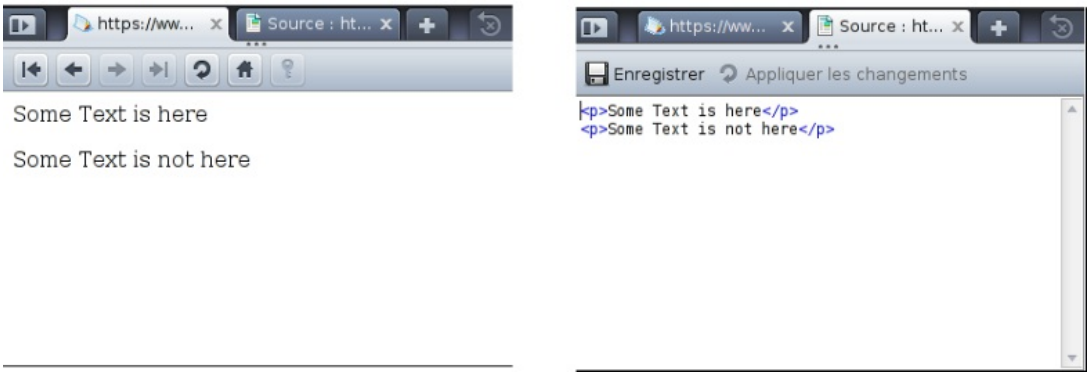
Macro Expansion

The effect of expanding a macro is to graft a subtree from another document (or from elsewhere in the current document) in place of the statement element, replacing the existing subtree. Parts of the original subtree may remain, grafted onto the new subtree, if the macro has slots. If the macro body uses any macros, they are expanded first.

When a macro is expanded, its define-macro attribute is replaced with the use-macro attribute from the statement element. This makes the root of the expanded macro a valid use-macro statement element.

metal:fill-slot

```
<p metal:define-macro="foo">Some Text <tal:block metal:define-slot="bar"
here</tal:block></p>
<h1 metal:use-macro="template/macros/foo">Some Other Text
<tal:block metal:fill-slot="bar">is not here</tal:block></h1>
```



▼ Details

Macros are much more useful if you can override parts of them when you use them. For example, you might want to reuse a complex table, but provide different contents for one of the table cells in every place that you use it. It would be possible to place the contents somewhere on each Template, define a variable for it, and insert that variable into the cell in the macro body. This, however, would violate the presentation structure, preventing you from seeing the table with the cell contents in place.

To make elements of the macro body overridable, add define-slot attributes with the value set to a slot name. Wherever the macro is used, choose corresponding sub-elements of the statement element and add fill-slot attributes with the value set to the slot name. When the macro is expanded, fill-slot elements will replace the define-slot elements in the macro body that use the same slot name.

Slot names must be unique within a single macro, and within a single macro use. It is legal, however, to define a slot in a macro and not fill it. This will simply cause the default contents of the slot definition to be copied into the expanded macro. If a fill-slot element names a slot that is not found in the macro body, it causes an error.