EXT: Templatedisplay

Extension Key: templatedisplay
Language: en
Keywords: forAdmins, forIntermediates

Table of Contents

Introduction

Templatedisplay can be viewed as "mappable" template engine for TYPO3. The extension is part of the Tesseract extensions family and deals with rendering content on the Frontend.

In short words, it enables to do a mapping between markers and databases fields. A marker is a pattern that will be replaced dynamically by a value coming from the database. This value can be formatted according typoscript configuration. It is also possible to incorporate user defined markers within a hook.

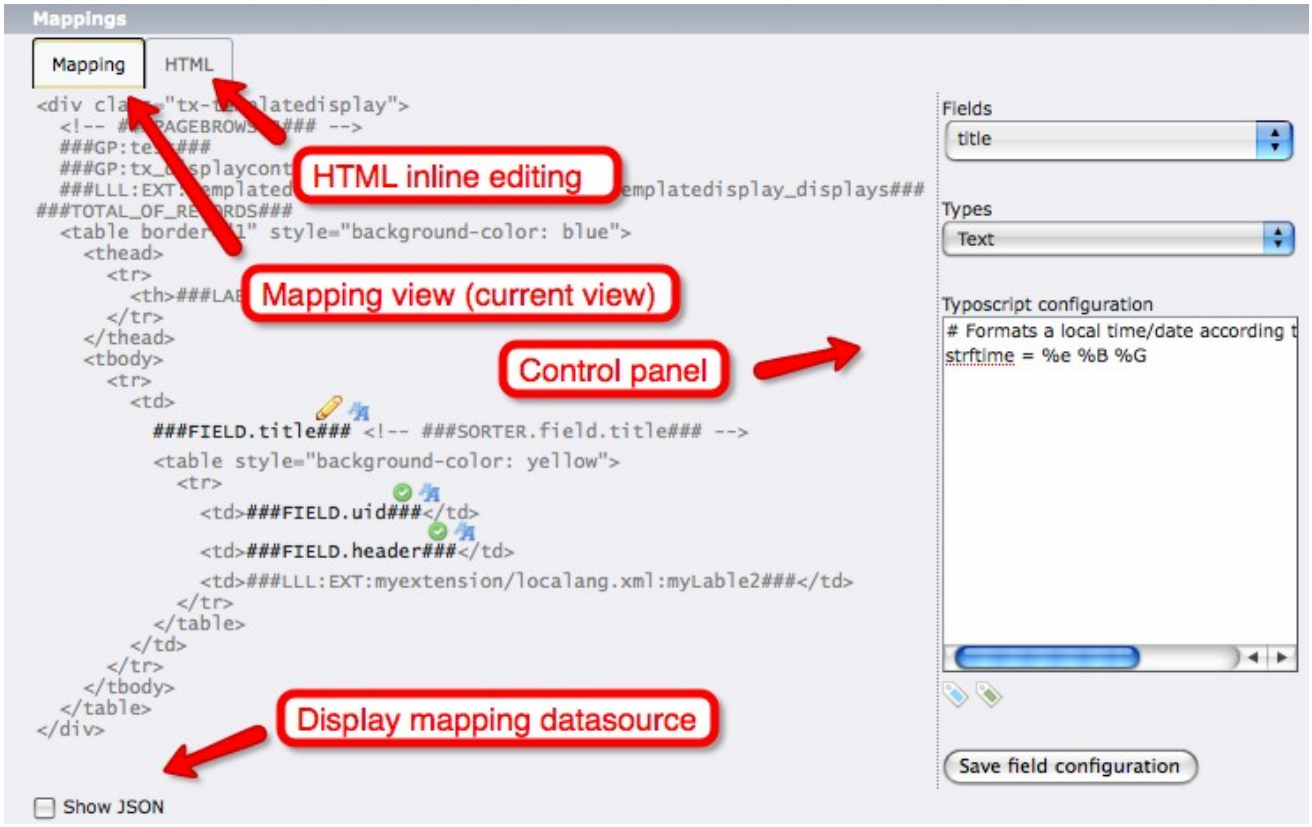
Templatedisplay is well designed for rendering lists with advanced features like sorting, filtering, page browsing. It offers a simple syntax for looping on record set, testing condition, counting records.

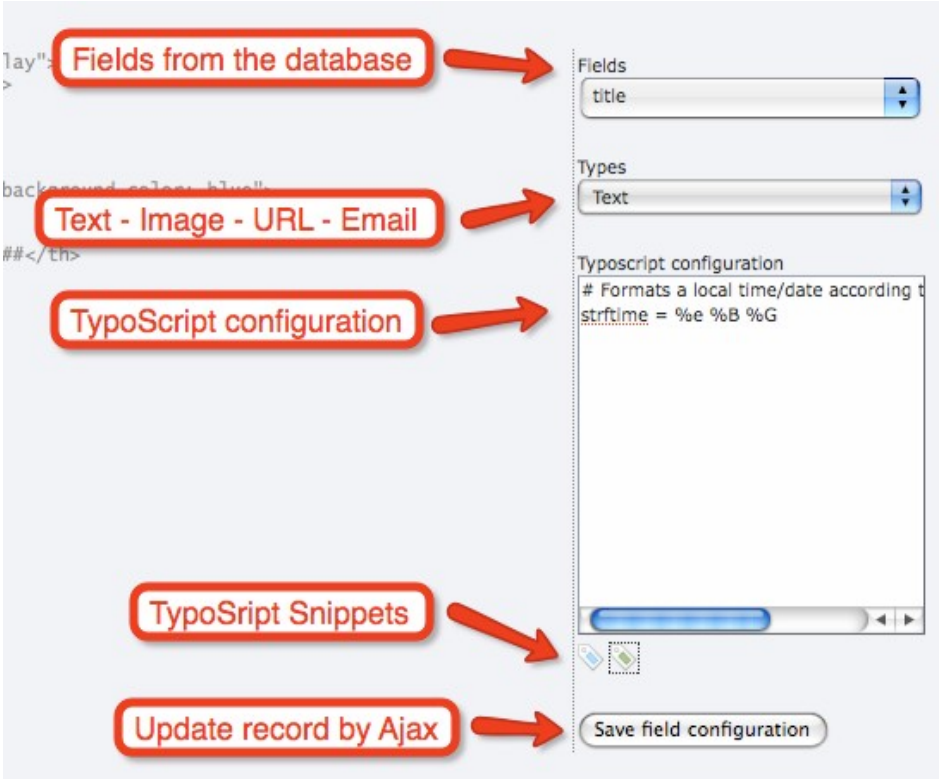
Screenshots

This is basically the default view where it is possible to map any markers to a database's field. The mapping is done by

1. clicking on a marker (e.g. ###FIELD.title###),
2. selecting a field in the drop-down menu,
3. selecting a type within the list,
4. adding some possible additional configuration,
5. clicking the "save field configuration" button.

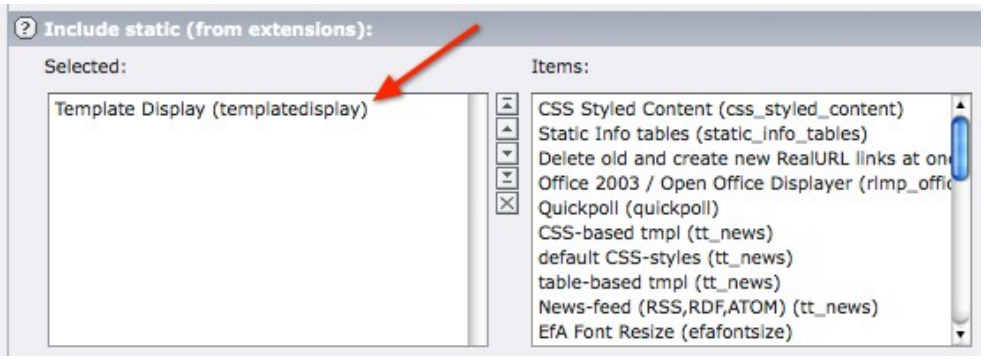
By using the name of the marker, Templatedisplay will tries to identify the field in the dropdown menu "Fields". If the correspondence fails, the filed must be selected manually in the drop-down menu.




Fields from the database
Text - Image - URL - Email
TypoScript configuration
TypoSript Snippets
Update record by Ajax
Fields
title
Types
Text
Typoscript configuration
# Formats a local time/date according t
strftime = %e %B %G
Save field configuration

Installation

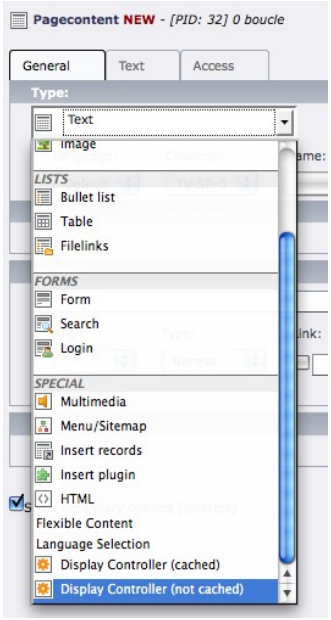
The extension must be installed as part of the Tesseract package. Once installed, the next step is to include the static template in your TypoScript, so as to benefit from default settings.



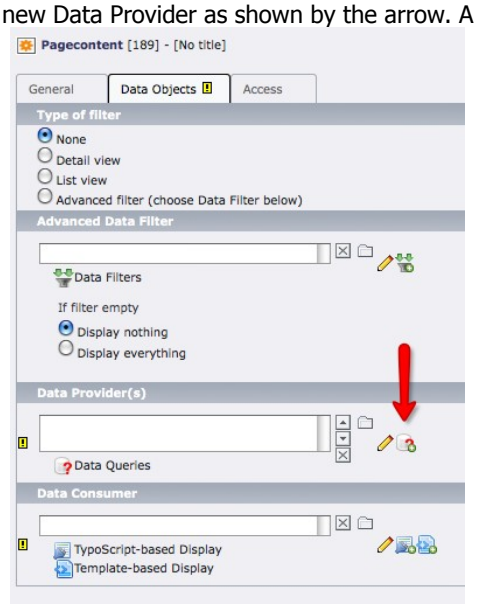
Use templatedisplay step by step

This is a step by step tutorial assuming that you have installed the whole set of Tesseract extensions.

Create a new tt_content record of type "controller".

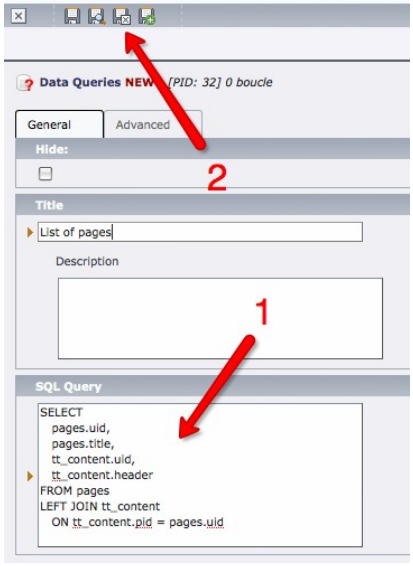


Click on tab "Data Objects" and create a new Data Provider as shown by the arrow. A new window will open.

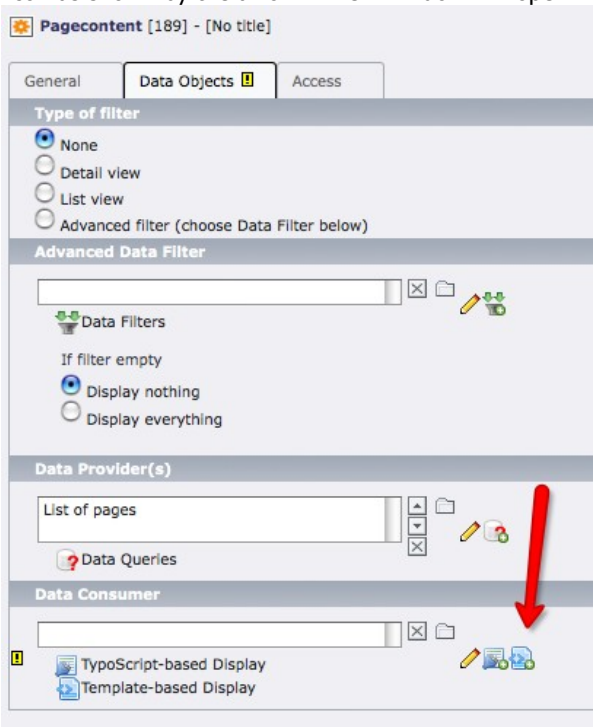


The "Data Provider" is used for fetching data in the database. Write down a request in field "SQL Query". Once it is done, "save and close" the record. For example, the request bellow is intended to display a list of page linked with their content elements.

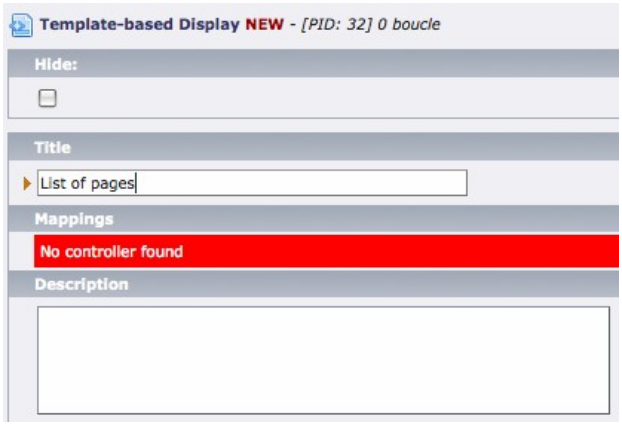
```
SELECT
   pages.uid,
   pages.title,
   tt_content.uid,
   tt_content.header
FROM pagesLEFT JOIN tt_content
   ON tt_content.pid = pages.uid
```



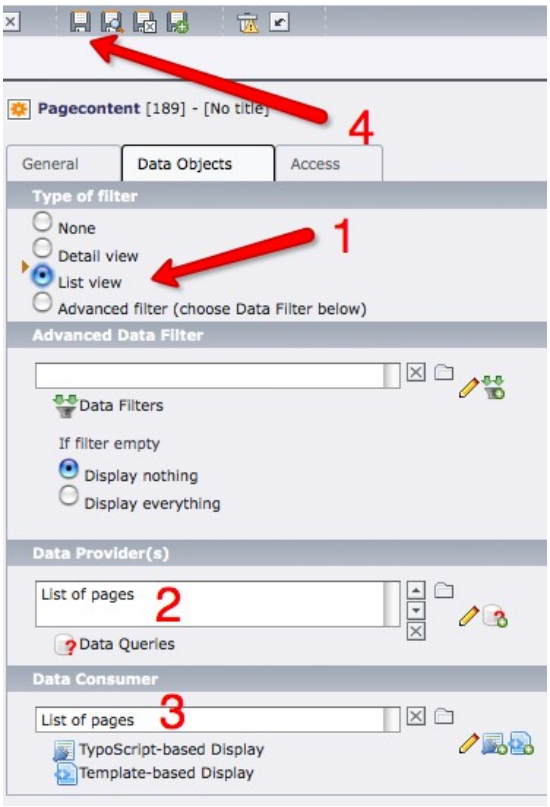
Then, we need to add a Consumer to the controller. Generally speaking in Tesseract, a "Consumer" is a service that copes with rendering a data structure. In the current case, Templatedisplay is a consumer that will render a data structure as HTML. Click on the "new Templatedisplay" icon as shown by the arrow. A new window will open.



Fill in the field "title" and click "save and close" the record. Notice the red bar which is **normal** in fact since the template does not know about it parents yet. Remember that the page content element has not been saved so far.

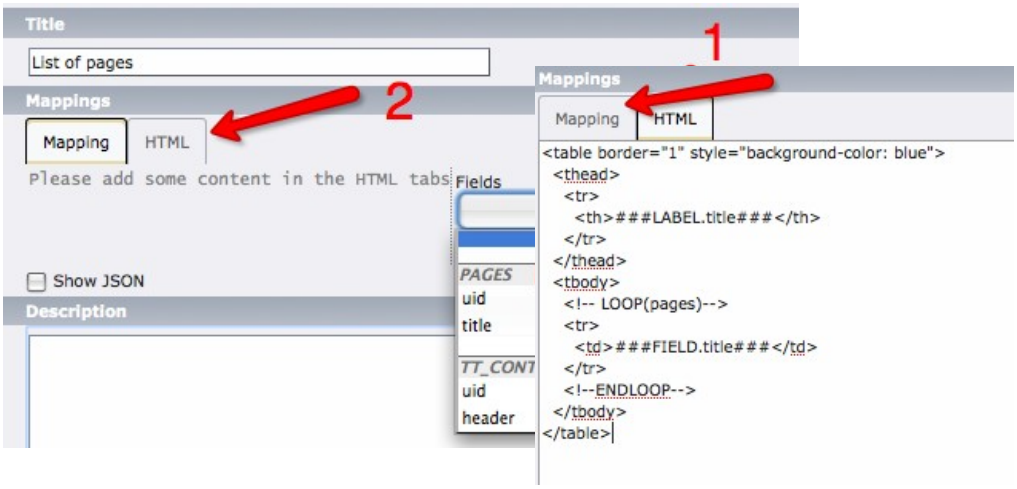


We are almost done. Before saving the controller, select "List view" (1) because we want to display a list of pages. Make sure point 2 + 3 are OK. Save the document.



Once these steps are done, it is possible to add HTML to the template. Select the consumer and hit the pencil icon. A popup window should appear.

By now, there is a drop down menu (1) containing all the fields appearing in the previously written request. Hit the HTML tab (2) and add some HTML. Copy / paste the example bellow for the purpose of the tutorial.

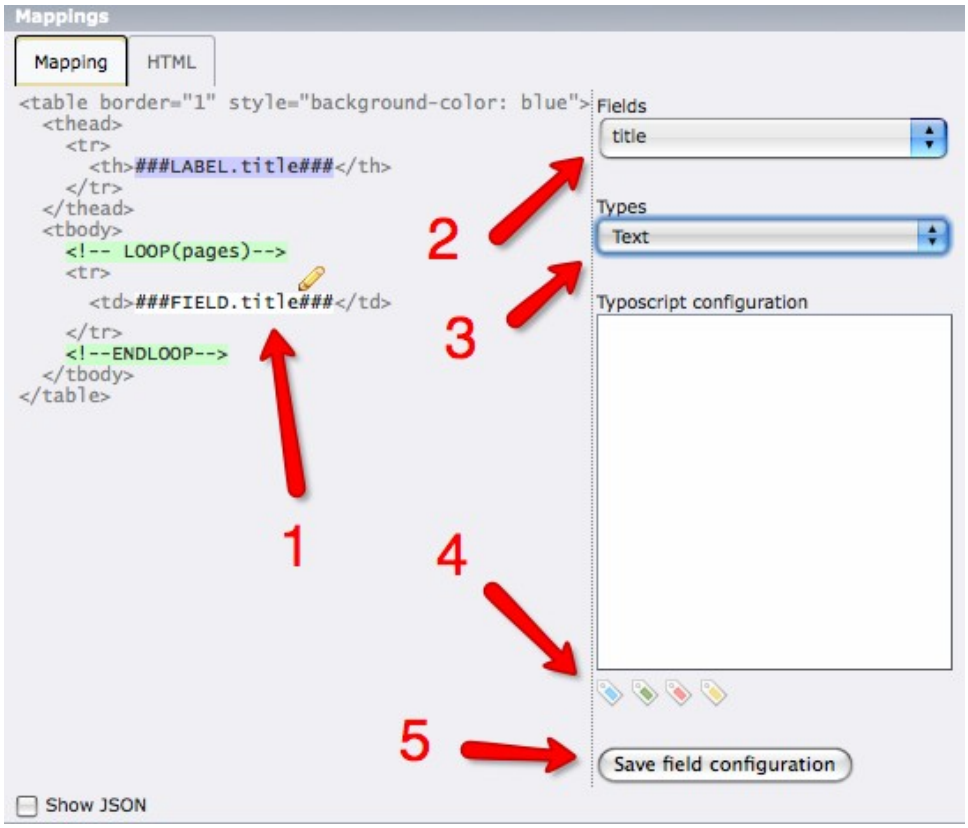


Notice that the HTML is saved to the database thanks to an Ajax call.

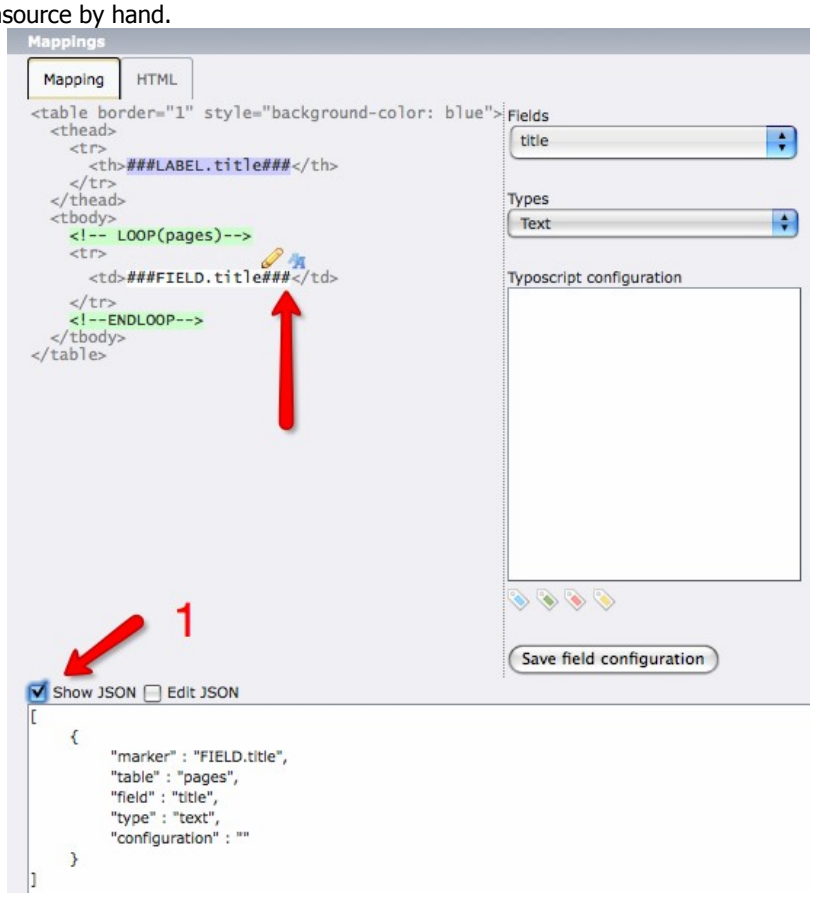
```
<table border="1" style="background-color: blue">
  <thead>
    <tr>
      <th>###LABEL.title###</th>
    </tr>
  </thead>
  <tbody>
    <!-- LOOP(pages)-->
    <tr>
      <td>###FIELD.title###</td>
    </tr>
    <!--ENDLOOP-->
  </tbody>
</table>
```

The Template is ready for mapping. Mapping is the process of associating a maker with a field of the database. Markers appear in white and are clickable. On the screenshot bellow, we can see a red  right to ###FIELD.title###. It indicates, the field is unmapped. Try to click on a marker and observe the behaviour of the right panel.

Click on the marker (1) and select the correct field in the drop down menu (2). Then choose a type for the field (3). Insert some TypoScript configuration if needed. Little snippet (4) icons are available. Finally, save everything by pressing the button (5).



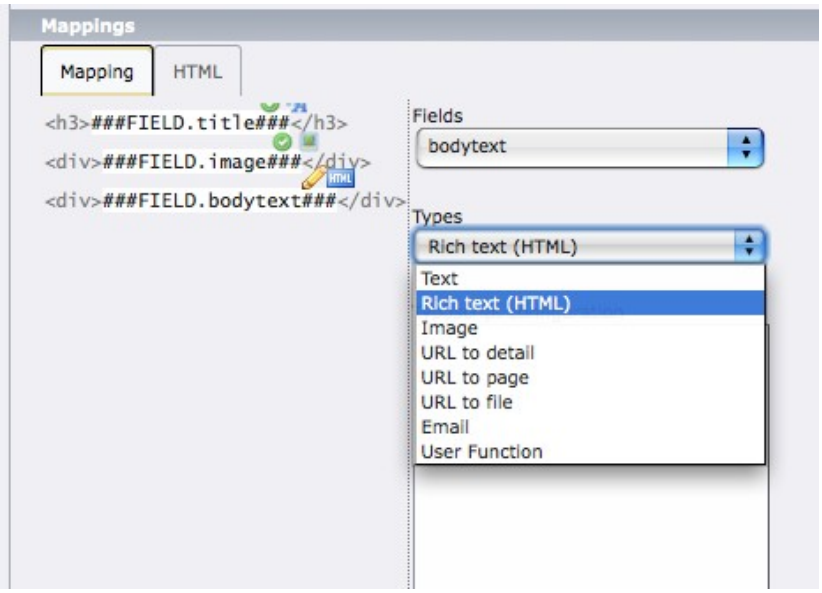
Once it is saved, a little icon appears right to the pencil showing the type of the marker: "text" – "richtext" – "image" – "link" – "email" - It is possible to access the mapping information by clicking on the "Show Json" checkbox (1) and modify if necessary the JSON datasource by hand.



Configuration

Element types

There are various elements types that can be chosen according to your needs.



- Text: corresponds to the TEXT content object. The local TypoScript is the same as for Object TEXT. The value of the TEXT object will be loaded with the value from the mapped database field.
- Rich text: this is actually the same as the "Text"-type element, but it is designed to handle fields that use the RTE. Such fields need special rendering, so that RTE data is interpreted before display in the front-end. As this is not necessary with every text field, a separate element type exists.
- Image: corresponds to the IMAGE content object. The value from the database field will be automatically stored in the "file" property of the object.
- URL to detail: this is designed to create a link and corresponds to the typolink TS function. It is automatically loaded with a configuration to create a link to a detail view as expected by the Display Controller (extension: displaycontroller). It is also configured to return only the URL, but this can be overriden in the local TS.
- URL to page: also a typolink. It should be used only for database fields who contain page id's, as it will create a link to said page.
- URL to file: also a typolink, but meant for links to files.
- Email: again the same, but for emails.
- User function: this corresponds to the USER content object. It is preconfigured with a property called "parameter" which will contain the value from the database field.

List of available markers

Content markers

Name	Description
###FIELD.myField###	This is the most common marker that deals with content of the database. When possible, try to make correspond the name of the marker with the name of the field. Templatedisplay will be able to guess automatically the mapping. Click on it to start the mapping process.
###OBJECT.myValue###	Attach some TypoScript to this marker. Same configuration options than FIELD markers but no field associated with.
###LABEL.myField###	The label of the field is translated according to the language of the website. To have a correct translation, the LABEL must have a proper TCA.

Name	Description
###LLL:EXT:myExtension/locallang.xml:myKey###	When no TCA is provided or an external string must be translated, use this syntax for translating a chain of character.
###EXPRESSION.key:var1|var2###	Calls on the expression parser of extension "expressions" to resolve any well-formed expression. **Example:** `###EXPRESSION.gp:clear_cache###` will retrieve the value of a GET/POST variable called "clear_cache".
###FILTER.myTable.myField###	Value of a filter. MyTable is optional and depend of the filter naming.
###SORT.sort###	Value of the sort. The most probably a field name.
###SORT.order###	Value of the order. Can be "ASC" or "DESC"
###SESSION.sessionName.order###	Access information stored in the session.
###COUNTER###	The counter is automatically incremented by 1. This syntax makes sense inside a LOOP and can be used for styling odd / even rows of a table for example. The syntax may looks like this: `<!--IF(###COUNTER### % 2 == 0)-->class="even"<!--ELSE--> >class="odd"<!--ENDIF-->` In the case of a LOOP in a LOOP the second COUNTER remains independent. `<!--LOOP(pages)--> <div>counter 1 : ###COUNTER###</div> <!--LOOP(tt_content)--> <div>counter 2 : ###COUNTER###</div> <!--ENDLOOP--> <!--ENDLOOP-->`
###COUNTER(loop_name)###	This kind of counter is handy in case of LOOP in a LOOP. Let's assume, we need to access the value of the parent COUNTER in a child's LOOP. `<!--LOOP(pages)--> <div>Some value</div> <!--LOOP(tt_content)--> <div>counter pages: ###COUNTER(pages)###</div> <!--ENDLOOP--> <!--ENDLOOP-->`
###PAGE_BROWSER###	If extension "pagebrowse" is installed and correctly loaded, displays a universal page browser. Other page browsers are possible but must be handled with a Hook.
###RECORD(tt_content, 12)###	Call in the template it self an external record. Very handy for including records in a records.
###HOOK.myHook###	See section Hooks
###TOTAL_RECORDS###	Returns the total number in the main of records **without considering a possible limit**. To have a glimpse on the data structure, add the parameter "debug[structure]" in the URL. The value ###TOTAL_RECORDS### corresponds to the cell "**totalCount**" of the main structure (level 1). Make sure you have a backend login to see the table.
###SUBTOTAL_RECORDS###	Returns the total of records in the main data structure **considering a possible limit**. To have a glimpse on the data structure, add the parameter "debug[structure]" in the URL. The value ###SUBTOTAL_RECORDS### corresponds to the cell "**count**" of the main structure (level 1). Make sure you have a backend login to see the table.
###TOTAL_RECORDS(tablename)###	Returns the total of records corresponding to a table name **without considering a possible limit**.
###SUBTOTAL_RECORDS(tablename)###	Returns the total of records corresponding to a table name **considering a possible limit**.

Name	Description
###RECORD_OFFSET###	Return the page offset. The page offset correspond to the current position inside a global record set. This marker is useful when displaying a page browser. See marker ###PAGE_BROWSER###. You can have something like this: ###RECORD_OFFSET### / ###TOTAL_RECORDS### which will display the current position among the total number of records.

Structure Markers

Name	Description
<!--LOOP(loop_name)--> <!--ENDLOOP-->	Where loop_name is a table name.
<!--IF(###FIELD.maker### == 'value')--> <!--ENDIF-->	Allows to display conditional content. Beware, at the moment it is not possible to overlap IF conditions

Functions

Name	Description
PRINTF(###FORMAT###, ###MARKER_1###, ###MARKER_2###, ###MARKER_n###)	Equivalent to the PHP function printf. Exepecint a %s, %d, etc... in the marker ###FORMAT### Example Let's define those 2 values LLL:EXT:myExtension/locallang.xml:myLabel = Hello %s ! FIELD.first_name = Fabien Expression... PRINTF(###LLL:EXT:myExtension/locallang.xml:myLabel###, ###FIELD.first_name###) ...will return Hello Fabien !
LIMIT(###MARKER###, 4)	Limit the number of words in a marker E.g. LIMIT(###FIELD.description###, 4) will return the first 4 words of field description
UPPERCASE(###MARKER###)	Uppercase a marker E.g. UPPERCASE(###LABEL.title###) will return "TITLE"
LOWERCASE(###MARKER###)	Uppercase a marker E.g. UPPERCASE(###EXPRESSION.gp:myExtension|myParameter###)
UPPERCASE_FIRST(###MARKER###)	Uppercase the first letter of the marker E.g.T UPPERCASE_FIRST(###LABEL.title###) will return "Title"
COUNT(tableName)	Return the number of records from the datastructure. Add parameter debug[structure] in the URL to see the datastructure. (Works with a BE login) E.g. COUNT(tt_content) will return the number of records in table tt_content
STRIPSLASHES(###MARKER###)	Stripslashes function
STR_REPLACE('search','replace',###FIELD###)	Search for a pattern and replaces by something else inside a marker.
PAGE_STATUS(404) PAGE_STATUS(404, page/404/) PAGE_STATUS(301, new/page/)	If the datastructure is empty, send the appropriate header and redirect link when needed.

Typoscript configuration

Default rendering

A default rendering can be defined for each element type. The static template provided with the extension contains the following:

```
plugin.tx_templatedisplay {
      defaultRendering {
            richtext.parseFunc < lib.parseFunc_RTE
      }
}
```

This configuration copies the RTE parseFunc into the parseFunc for the rich text-type element, making possible to render correctly RTE-enabled fields. Here's an example configuration:

```
plugin.tx_templatedisplay {
      defaultRendering {
            text.wrap = <span class="text">|</span>
      }
}
```

This would wrap a span tag with a "text" class around **every** text-type element rendered by Template Display.

Other examples

Exemple 1: defining the page title according to a field value, useful for a detail view. **Make sure, "Display Controller (cached)" is defined.** Otherwise, "substitutePageTitle" will have no effect.

```
plugin.tx_templatedisplay {
    substitutePageTitle = {title} - {field_custom}
}
```

Exemple 2: giving the pagebrowse parameters

```
plugin.tx_templatedisplay {
      pagebrowse {
            templateFile = fileadmin/templates/plugins/pagebrowse/template.html
            enableMorePages = 1
            enableLessPages = 1
            pagesBefore = 3
            pagesAfter = 3
      }
}
```

Reference

Name	Value	Description
defaultRendering	Rendering configuration	Default TS configuration for each element type
substitutePageTitle	dataWrap	Substitute page title with values from the datastructure. Instead of having the default page title, it is possible to set an other value
pagebrowse	pagebrowse configuration	See extension pagebrowse

[tsref:plugin.tx_templatedisplay]

Templatedisplay by examples

A good way to acquire a new skill is to observe and learn by examples.

TODO: What about having a T3 package containing a working Tesseract.

Developer's Guide

Hooks

Hooks offer an opportunity to step into the process at various points. They offer the possibility to manipulate data and influence the final output. Hooks can be used to replace personalized markers, introduced previously in the HTML template. There is a convention in templatedisplay to name Hook like ###HOOK.myHook###.

In templatedisplay, there are 2 available hooks:

- preProcessResult (for pre-processing the HTML template)
- postProcessResult (for post-processing the HTML content)

To facilitate the implementation of a hook, a skeleton file can be found in EXT:templatedisplay/samples/class.tx_templatedisplay_hook.php

Step 1

In the ext_localconf.php of your extension, register the Hook.

```
$GLOBALS['TYPO3_CONF_VARS']['EXTCONF']['templatedisplay']['postProcessResult']['myHook'][] =
'EXT:templatedisplay/class.tx_templatedisplay_hook.php:&tx_templatedisplay_hook';
```

Remarks:

- "postProcessResult" can be replaced by "preProcessResult".
- "myHook" can be something else but must correspond to the marker ###HOOK.myHook###.
- Make sure the path of the file is correct and suit your environment.
- Don't forget to clear the configuration cache!!

Step 2

Write the PHP method that will transform the content.

```
class tx_templatedisplay_hook {
	public function postProcessResult($content, $marker, &$pObj) {
		$controller = $pObj->getController();
		$data = $controller->cObj->data;
		if ($data['uid'] == 11399) {
			$_content = '';
			$content = str_replace('###HOOK.myHook###', $_content, $content);
		}
		return $content;
	}
}
```

Custom element types

It is possible to define custom element types. Such types will be added to the list of available types in the mapping interface, which makes them easier to use for users than the user-function type.

As for hooks this is a two-step process.

Step 1

Register the custom type in ext_localconf.php file of your extension. The syntax is as follows:

```
$GLOBALS['TYPO3_CONF_VARS']['EXTCONF']['templatedisplay']['types']['tx_test_mytype'] = array(
	'label'		=> 'LLL:EXT:' . $_EXTKEY . '/locallang.xml:mytype',
	'icon'		=> 'EXT:' . $_EXTKEY . '/mytype.png',
	'class'		=> 'tx_test_templatedisplay'
);
```

The custom type is registered with a specific key (e.g. "tx_test_mytype") and with the following information:

- a label that will appear in the drop-down list of available element types (as well as alt text for the icon)
- an icon that will appear in the mapping interface when that type has been selected
- a class to do the processing of that custom type. The class must implement the tx_templatedisplay_CustomType interface (more below).

You also need to include the class that will do the processing. As of TYPO3 4.3 you can register it with the autoloader instead (this is the preferred way).

Step 2

The method itself is expected to do the rendering. It receives the following parameters:

Parameter	Type	Description
$value	mixed	The current value of the field that was mapped.
$conf	array	TypoScript configuration for the rendering (this may be ignore if you don't need TypoScript).
$pObj	object	A reference to the calling tx_templatedisplay object.

A sample implementation is provided in the "samples/class.tx_templatedisplay_phonetype.php" file. The code looks like this (without comments):

```
class tx_templatedisplay_PhoneType implements tx_templatedisplay_CustomType {
	function render($value, $conf, tx_templatedisplay $pObj) {
		$rendering = '<a href="callto://' . rawurlencode($value) . '">' . $value . '</a>';
		return $rendering;
	}
}
```

In this simple example the class just does some minor processing with the value it receives and returns the result.

As of TYPO3 4.3, it is recommended that such classes also implement the t3lib_Singleton interface so that only one instance of it is created (otherwise one instance is created for each field using this custom type on each pass in the loop). This will save memory.

Debugging

Debugging is provided in form of parameters added in the URL. A backend login is mandatory in order to see the output.

Name	Description
debug[structure]	Display the current data structure. Useful to see which data are given to templatedisplay.
debug[template]	Display the structure of the template. The template is cut in small pieces for processing according the LOOP and SUBLOOP.
debug[filter]	Display the active filter.
debug[markers]	Display the list of markers and their values.
debug[display]	Display the untranslated markers. By default, the production mode is active. It means when a translation does not succeed, the marker is erased from the final output. For debug purpose, it might be useful to identify these markers.
debug[pagebrowse]	Display the parameter that are transmitted to the pagebrowser. Extension pagebrowse is expected here.

Whenever the extension "devlog" is installed, it is also possible to monitor some information. In a templatedisplay record, check the following options as desired. It may be convenient, in (pre) production mode to check some information produced by visitor.

Changelog

Version	Changes:
1.0	Initial version of the extension
1.0.1	Added page browse documentation Added Typoscript configuration
1.1	Added PRINTF FUNCTION
1.2	Added default rendering option Added rich text element type