EXT: Generic expression parser

Extension Key: expressions
Language: en
Keywords: forDevelopers, forAdvanced

Table of Contents

Introduction

What does it do?

In TypoScript templates there exists a function called "getText", which makes it possible to retrieve values from a great number of sources (GET/POST, TSFE, rootline, etc.). This extension provides a library which aims to reproduce this capability, so that it can be used by developers in their own extension.

This can be useful in cases where you have some kind of configuration but are not using TypoScript, for whatever reason.

So what is an expression?

Like for the TypoScript function, an expression – at a minimum – is comprised of a key, a colon (:) and a value. Example:

```
gp:no_cache
```

The above syntax will instruct the parser to look for a GET/POST variable called "no_cache". The key represents the "space" where to look for the value. With some special keys, the value takes a different meaning. This is documented below.

The parser library can either parse a single expression or parse expressions inside strings, provided expressions are placed between curly braces. Example:

```
This is the value of no_cache: {gp:no_cache}
```

Questions?

If you have any questions about this extension, please ask them in the TYPO3 English mailing list (typo3.english), so that others can benefit from the answers.

Keeping the developer happy

If you like this extension, use the social functions of the TER to make some buzz about it. Alternatively, if you want to give something back, you may consider my Amazon wish list: http://www.amazon.co.uk/registry/wishlist/G7DI2AN99Y4F

You may also take a step back and reflect about the beauty of sharing. Think about how much you are benefiting and how much yourself is giving back to the community.

Compatibility

As of version 1.5.0, TYPO3 4.5 or more is required.

User's Guide

Expression syntax

The general syntax of an expression – at its simplest – is the following:

```
key:value1|value2|value3...
```

The key defines the "domain" where to look for the value or sequence of values. For example, the key "page" means that values should be looked for in the current page record.

The values can be multiple, separated by pipe symbols (|). Each value normally represents a "dimension" of the "domain" where the value is being looked up. This can be either a dimension of an array or a member variable of an object. This is totally interchangeable, i.e. the expression parser will traverse the structure transparently whether any given "dimension" is an actual array dimension or some object property. Example:

```
tsfe:fe_user|user|uid
```

"fe_user" is an object and "user" is one of its member variables. But "user" is an array and "uid" is one its keys. As you can see it terms of expression syntax, this is totally transparent.

If no key is found inside the string to parse, it is returned as is.

Extended syntax

Additions to the base syntax make it possible to call functions for post-processing the result of the expression. The syntax becomes:

```
key:value1|value2|value3...->function:arg1,arg2,...
```

The function call is indicated by the "->" symbol. Then comes the function key, which the parser will match to an actual function or method call. The function will receive the value of the expression as an argument. It may also require additional arguments, which are defined after a colon (:) and separated by commas (,). Example:

```
gp:tx_myext_pi1|foo->fullQuoteStr:pages
```

In this example, we retrieve the value of the GET/POST variable `tx_myext_pi1[foo]` and put it through `t3lib_db::fullQuoteStr()` with the "pages" table name as a second argument (the first one being the value itself).

Functions can be called in a chain:

```
key:value1|value2|value3...->function1:arg1,arg2,...->function2:arg1,arg2,...->...
```

The call order is from left to right.

Available function keys are described below.

Subexpressions

An expression could itself contain expressions. These are called "subexpressions". The syntax might look something like:

```
tsfe:fe_user|user|{gp:fe_field}
```

In this case, the expression would first be parsed for subexpressions. Assuming the GET/POST var "fe_field" contained "name" as a value, the expression would become

```
tsfe:fe_user|user|name
```

which would then be parsed normally.

Every feature of expressions can be used inside subexpressions, except further subexpressions.

Alternate expressions

An expression can be made of several expressions, separated by a double slash (//):

```
key1:value1|value2|... // key2:value1|value2|... // …
```

These represent alternatives. Expressions are evaluated from left to right and the first one to return a value will end the process. Thus the expression with "key2" (in the above example) will be evaluated only if the first one didn't return anything.

This can be used in particular to define default values, by using an alternate that is **not** an expression. Example:

```
gp:year // 2010
```

In this case, the expression parser will look for a GET/POST variable called "year". If it does not exist, it will evaluate the next alternative. Since it is a simple value (it's not an expression, because it has not key – as marked by colon), it will be taken as is and the complete expression will evaluate to "2010".

Expression keys reference

The table below lists all the keys provided by the expression parser.

Key:	Examples:	Explanations:
tsfe	tsfe:id *Retrieve the uid of the current page* tsfe:fe_user|user|username *Retrieve the username of the currently logged in user*	Get a value from the $TSFE global variable
page	page:uid *Retrieve the uid of the current page*	Get a value related to the current page, as stored in $TSFE->page
config	config:language *Retrieve the current language*	Get a value from the "config" object, as stored in $TSFE->config['config']
fe_user	fe_user:name *Retrieve the current FE user's name*	Get a value for the current frontend user, as stored in $GLOBALS['TSFE'] ->fe_user->user
plugin	plugin:tx_vgetagcloud_pi1.|startPage.|data *Retrieve the TypoScript property plugin.tx_vgetagcloud_pi1.startPage.data*	Get a TypoScript property for a plugin as stored in $GLOBALS['TSFE']->tmpl ->setup['plugin.']. Remember that TS indices have an ending dot (.).
gp	gp:tx_ttnews|tt_news *Retrieve the uid of a single news record*	Get a value from either the $_GET or $_POST superglobal arrays. This makes it possible to retrieve any GET or POST variable passed to the page.
vars	vars:showUid *Retrieve the "showUid" variable from the piVars*	Get a value from "internal" variables. This depends on what is loaded here by the extension that users the parser.
extra	extra:foo *Retrieve the value of "foo" from the extra data array of the data filter*	Same as above, but for so-called "external" variables.
date	date:Y *Retrieve the current year (4 digits)*	Get values related to the current time, using formats from the PHP date() function.
strtotime	strtotime:2009-01-01 *Retrieve the timestamp corresponding to Jan 1, 2009* strtotime:tomorrow *Retrieve the timestamp corresponding to tomorrow, same time*	Get a timestamp by interpreting a human-readable date, as per the capacities of PHP's function strtotime().
session	session:dummy|bob *Retrieve index "bob" of array dummy stored into the session*	Get values from some structure stored in the temporary session (i.e. "ses" and **not** "user"). The first item (after the "session:" key) has a special meaning. It corresponds to the key that was used to store into the session. The following indices are used normally.

Key:	Examples:	Explanations:
env	env:http_host *Retrieve the host name*	Get values from the environment variables, via the use of t3lib_div::getIndpEnv(). See that method for available values. **Note:** t3lib_div::getIndpEnv() expects values to be uppercase. The expressions parser takes care of uppercasing any incoming value, so there's no need to worry about that.

Expressions can be used both in the the frontend and the backend, but as you can see in the above list, some keys will obviously not be available in the backend as they rely on frontend object (for example, `$TSFE`).

It is possible to create processors for custom expression keys (see "Developer's Guide") and thus to extend the expression parser.

Function keys reference

The table below lists all the function keys available in the expression parser. Arguments between square brackets are optional.

Key:	Examples:	Explanations:
fullQuoteStr:arg	gp:tx_myext_pi1|foo->fullQuoteStr:pages *Call t3lib_db::fullQuoteStr() on the value, with "pages" as argument*	Calls `t3lib_db::fullQuoteStr()`. Requires an additional argument, which corresponds to a table name (see the method's comments if in doubt). Will not be available if the `$TYPO3_DB` global variable is not set.
quoteStr:arg	gp:tx_myext_pi1|foo->quoteStr:pages *Call t3lib_db::quoteStr() on the value, with "pages" as argument*	Same as above, but using `t3lib_db::quoteStr()` instead.
strip_tags:[arg]	gp:tx_myext_pi1|foo->strip_tags:<p> *Call the PHP function strip_tags() on the value but preserve <p> tags*	Calls the PHP function `strip_tags()`. The additional argument is optional and corresponds to the list of tags to preserve (refer to the PHP manual for more details).
removeXSS	gp:tx_myext_pi1|foo->removeXSS *Call t3lib_div::removeXSS() on the value*	Calls `t3lib_div::removeXSS()`.
intval:[arg]	gp:tx_myext_pi1|foo->intval:8 *Call the PHP function intval() using base 8*	Calls the PHP function `intval()`. The additional argument is optional and corresponds to the base to use for conversion (refer to the PHP manual for more details).
floatval	gp:tx_myext_pi1|foo->floatval *Call the PHP function floatval()*	Calls the PHP function `floattval()`.
boolean	gp:tx_myext_pi1|foo->boolean *Return FALSE if value is empty, TRUE otherwise*	This is an internal function. It's actually the opposite of the PHP function `empty()`. This means that any value of 0 or "0", or an empty string will return FALSE. Any other value will return TRUE.
hsc:[arg1,arg2,arg3]	gp:tx_myext_pi1|foo->hsc *Call the PHP function htmlspecialchars()*	Calls the PHP function `htmlspecialchars()`. The additional arguments are all optional and corresponds respectively to the behavior to adopt regarding single and double quotes, the character set to use in the conversion and what to do about existing HTML entities (refer to the PHP manual for more details).

Key:	Examples:	Explanations:
strftime	gp:date->strftime:%d.%m.%Y *Call the PHP function strftime() to format the timestamp received*	Calls the PHP function `strftime()`. It expects a Unix timestamp as a value and takes a date format as an additional argument (refer to the PHP manual for more details).

It is possible to create processors for custom function keys (see "Developer's Guide") and thus be able to call pretty much any function inside an expression.

Developer's Guide

This chapter describes what can be achieved using the library provided by this extension. If you are using TYPO3 4.3, the class is registered with the autoloader, so there's nothing to do but use it. Otherwise – as a first step – you will have to include the parser class, e.g.

```
require_once(t3lib_extMgm::extPath('expressions', 'class.tx_expressions_parser.php'));
```

API

This section describes the API made available in the parser class.

Method:	Purpose:	Parameters:
evaluateExpression	This method evaluates a single expression and returns the value	$expression: the string to parse and evaluate. This string is not expected to contain subexpressions. Use evaluateString() instead.
evaluateString	This method parses a string and evaluates every expression that it contains. Furthermore the resulting string itself can be evaluated as an expression	$string: the string to parse. It may contain subexpressions, which will be parsed appropriately. $doEvaluation: boolean, true by default. If true the string itself is evaluated as an expression, after all subexpressions have been evaluated. If false, the string itself is not evaluated.
setVars	This method can be used to store so-called "internal" variables inside a static member variable of the parser class (see below).	$vars: an array of any size. It replaces any existing "internal" variables.
setExtraData	This method can be used to store so-called "external" variables inside a static member variable of the parser class (see below).	$data: an array of any size. $reset: boolean. If true, the existing "external" variables will be overridden by the content of $data. If false (which is the default), the content of $data is merged with existing "external" variables.

All the above methods are static, so there's no need to create an instance of the parser class. To evaluate a string, simply make a call like:

```
$parsedString = tx_expressions_parser::evaluateString($string, true);
```

Internal and external variables

The idea behind internal and external variables is to provide two different spaces where to store custom data, to be retrieved by the parser. What is available in these spaces depends on what you stored into them. This is really up to each extension that uses expressions.

The concept is that "vars" is for values that can be considered as internal to the extension. For example, for a FE plugin, you might want to load the piVars into the "vars" array, so that they can be used inside an expression. On the other hand, the "extra data" array might contain values that come from some other external source, for example some general TypoScript.

It is up to each extension developer to decide whether to use those arrays or not, or also to differentiate between the two or just use a single one. It is really just a conceptual difference, so it must essentially make sense with regard to the extension itself.

Example:

```
$data = array('foo' => 'bar');
tx_expressions_parser::setExtraData($data, false);
```

This will load the $data array into the internal variables. Values from this array can then be retrieved with the following expression:

```
extra:foo
```

which will return "bar".

Exceptions

The `evaluateExpression()` method may throw a number of exceptions. The `evaluateString()` method will catch them, but if you call `evaluateExpression()` directly you will have to take care of these exceptions.

Introducing new keys

Using hooks it is possible to create classes that can interpret new keys. The first step is to create a class that implements the `tx_expressions_keyProcessor` interface. This interface defines a single method called getValue() which is expected to return some value after having interpreted the string it receives. What the method receives is the part of the expression that comes after the colon. For example, if the expression is:

```
key:value1|value2|value3...
```

the method will received "value1|value2|value3" as argument. The method should handle that string and return whatever makes sense. It may throw exceptions to show that the input was not valid and didn't lead to a valid result.

The class must then be registered in the `localconf.php` file or some extension's `ext_localconf.php` file, using the following syntax:

```
$GLOBALS['TYPO3_CONF_VARS']['EXTCONF']['expressions']['keyProcessor'][key] =
'EXT:myext/class.tx_myext_keyprocessor.php:&tx_myext_keyprocessor';
```

Note that it differs slightly from usual hooks. In this case the expression key that can be interpreted by the class is used as a key in the hook array (in bold in the above example). The advantage is that all registered hooks are not called unnecessarily every time a custom key is found. The limitation is that there can be only one registered hook per custom key, but it wouldn't make sense anyway to have several classes to handle the same custom key.

Example

Assume the following expression must be parsed:

```
negative:yes
```

The class to handle it will have to be registered as:

```
$GLOBALS['TYPO3_CONF_VARS']['EXTCONF']['expressions']['keyProcessor']['negative'] =
'EXT:myext/class.tx_myext_keyprocessor.php:&tx_myext_keyprocessor';
```

It will receive "yes" as an input. The code of the class may be something like:

```
require_once(t3lib_extMgm::extPath('expressions',
'interfaces/interface.tx_expressions_keyprocessor.php'));

class tx_myext_keyprocessor implements tx_expressions_keyProcessor {
      public function getValue($indices) {
            $value = 'Yes';
            if ($indices == 'yes') {
                  $value = 'No';
            }
            return $value;
      }
}
```

In the above example, it will return "No".

Introducing new functions

Just like with keys, it is possible to use a hook to introduce custom functions, which makes it possible to do pretty much whatever one wants with expressions. An example class is provided in `samples/class.tx_expressions_functionProcessor.php`. It introduces a function called "offset". To make this function available, it must be registered as follows:

```
$GLOBALS['TYPO3_CONF_VARS']['EXTCONF']['expressions']['customFunction']['offset'] =
'EXT:test/class.tx_test_functionprocessor.php:&tx_test_functionProcessor->offset';
```

The function itself looks like this:

```
public function offset($parameters, $pObj) {
      $value = intval($parameters[0]);
      $offset = 0;
      if (isset($parameters[1])) {
            $offset = intval($parameters[1]);
      }
      return $value + $offset;
}
```

It receives an array containing the function call parameters and what would normally be a back-reference to the calling object, but is actually a dummy object, since the `tx_expressions_parser` class is purely static. The call parameters array is an indexed array containing:

- at index 0, the value that was calculated from the expression
- at index 1 and more, any additional parameters that were defined in the function call.

As an example, consider the following expression using this custom function:

```
gp:foo->offset:5
```

The `offset()` method will receive the value of "gp:foo" in `$parameters[0]` and "5" in `$parameters[1]`.

The function is expected to return the modified value, or the original value from the expression if no change happened.

Hooks

There's one hook available in the parser library. It allows the manipulation of the value resulting from an expression. It must be registered with something like:

```
$GLOBALS['TYPO3_CONF_VARS']['EXTCONF']['expressions']['postprocessReturnValue'][] =
'EXT:myext/class.tx_myext_keyprocessor.php:&tx_myext_valuePostProcessor';
```

It must implement the `tx_expressions_valuePostProcessor` interface. Here's a sample code:

```
require_once(t3lib_extMgm::extPath('expressions',
'interfaces/interface.tx_expressions_valuepostprocessor.php'));
class tx_myext_valuePostProcessor implements tx_expressions_valuePostProcessor {
      public function postprocessReturnValue($value) {
            if (is_numeric($value)) {
                  $value += 30;
            }
            return $value;
      }
}
```

This sample code will add 30 to the given value, if it is numeric.

As can be seen, the hook method receives the value itself and is expected to return it, even if unchanged.

Fluid View Helper

The extension comes with a view helper for Fluid. As curly braces ({ and }) are used as markers in Fluid, they must be escaped so as not to break Fluid rendering.

Declaration of the namespace:

```
{namespace expression = Tx_Expressions_ViewHelpers}
```

Usage:

```
<expression:evaluate>Current page id is \{tsfe:id\}</expression:evaluate>
```

Result:

```
Current page id is 1
```

Usage (with inline notation):

```
{expression:evaluate(expression:'Current user is \{fe_user:username\}')}
```

Result:

```
Current user is zaphod
```

Note: inline notation is available only since version 1.4.0.

Security concerns

This library makes it possible to tap GET and POST variables. The safety of these values can never be ascertained. The exact risks depend on where this library is used. Care must be taken – once values are retrieved using the expressions library – to sanitize those values in an appropriate manner depending on how they are used (e.g. XSS, SQL injection, etc.). The functions that can be called on each expression provide some safety against this.

When expressions are used inside an extension, it is also up to each developer to judge what additional security to implement.

Known problems

None to date. Please report any problem to the TYPO3 mailing lists (typo3.english or typo3.dev as appropriate) or submit a bug report to the extension's bug tracker (http://forge.typo3.org/projects/extension-expressions/issues).

To-Do list

Nothing much planned yet, feel free to make suggestions. Use the extension's bug tracker (http://forge.typo3.org/projects/extension-expressions/issues) to submit feature requests.