EXT: Data Filter

Extension Key: datafilter
Language: en
Keywords: forEditors, forIntermediates

Table of Contents

Introduction

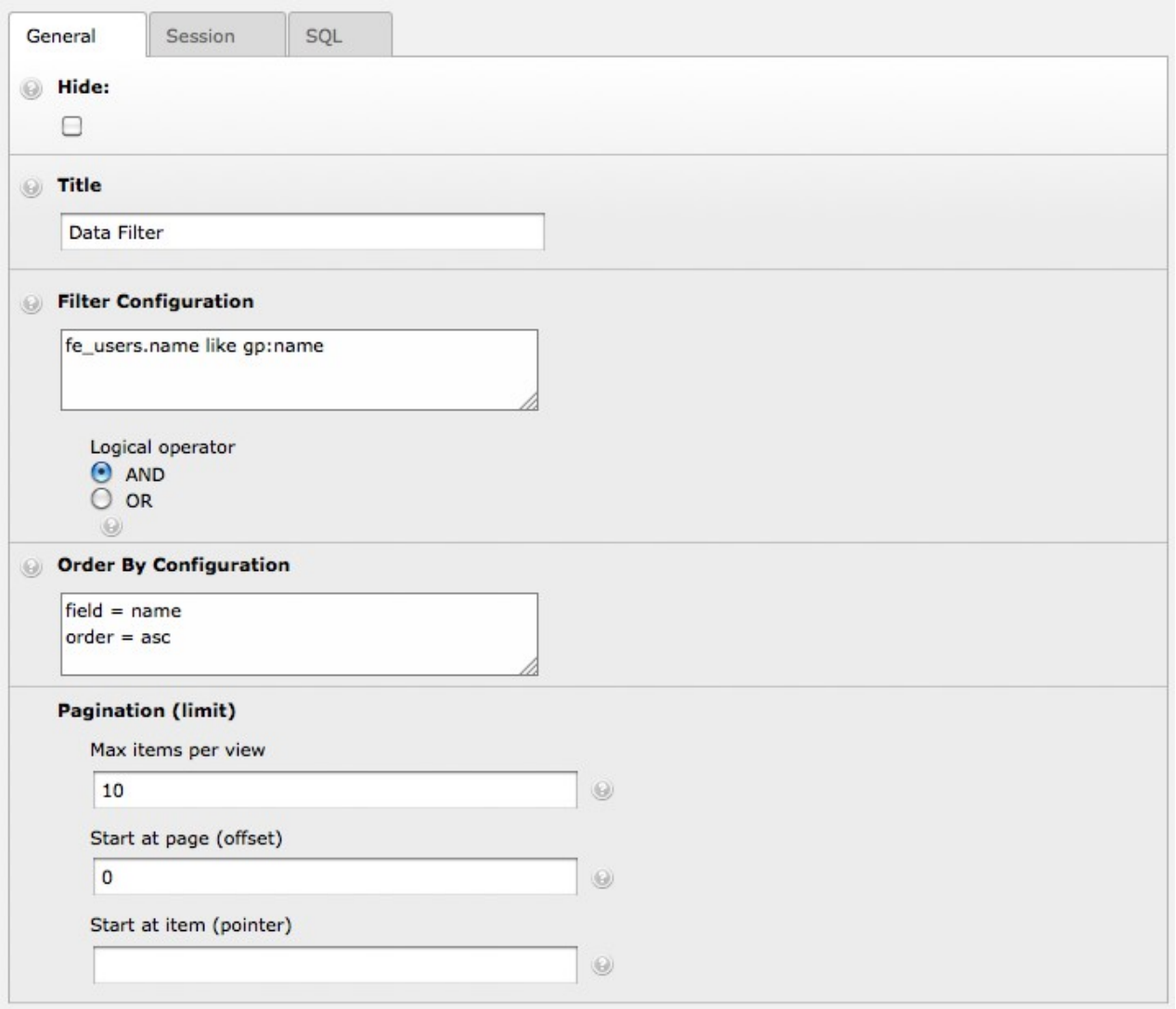
What does it do?

This extension provides a Data Filter (see extension "tesseract" for a general explanation) which is capable of retrieving data from many sources and format them into a Data Filter Structure. This structure can then be used by a Data Provider to filter the data it will return.

On top of strictly filtering the data, the Data Filter can also return information about capping the results (corresponding to the LIMIT/OFFSET keywords of SQL) and about sorting the data.

Screenshots

This shows the typical input screen of a Data Filter (note the "SQL" tab is added by extension "dataquery"):



Questions?

If you have any questions about this extension, you may want to refer to the Tesseract Project web site (http://www.typo3-tesseract.com/) for support and tutorials. You may also ask questions in the TYPO3 English mailing list (typo3.english).

Compatibility

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

Keeping the developer happy

If you like this extension, do not hesitate to rate it. Go the Extension Repository, search for this extension, click on its title to go to the details view, then click on the "Ratings" tab and vote (you need to be logged in). Every new vote keeps the developer ticking. So just do it!

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.

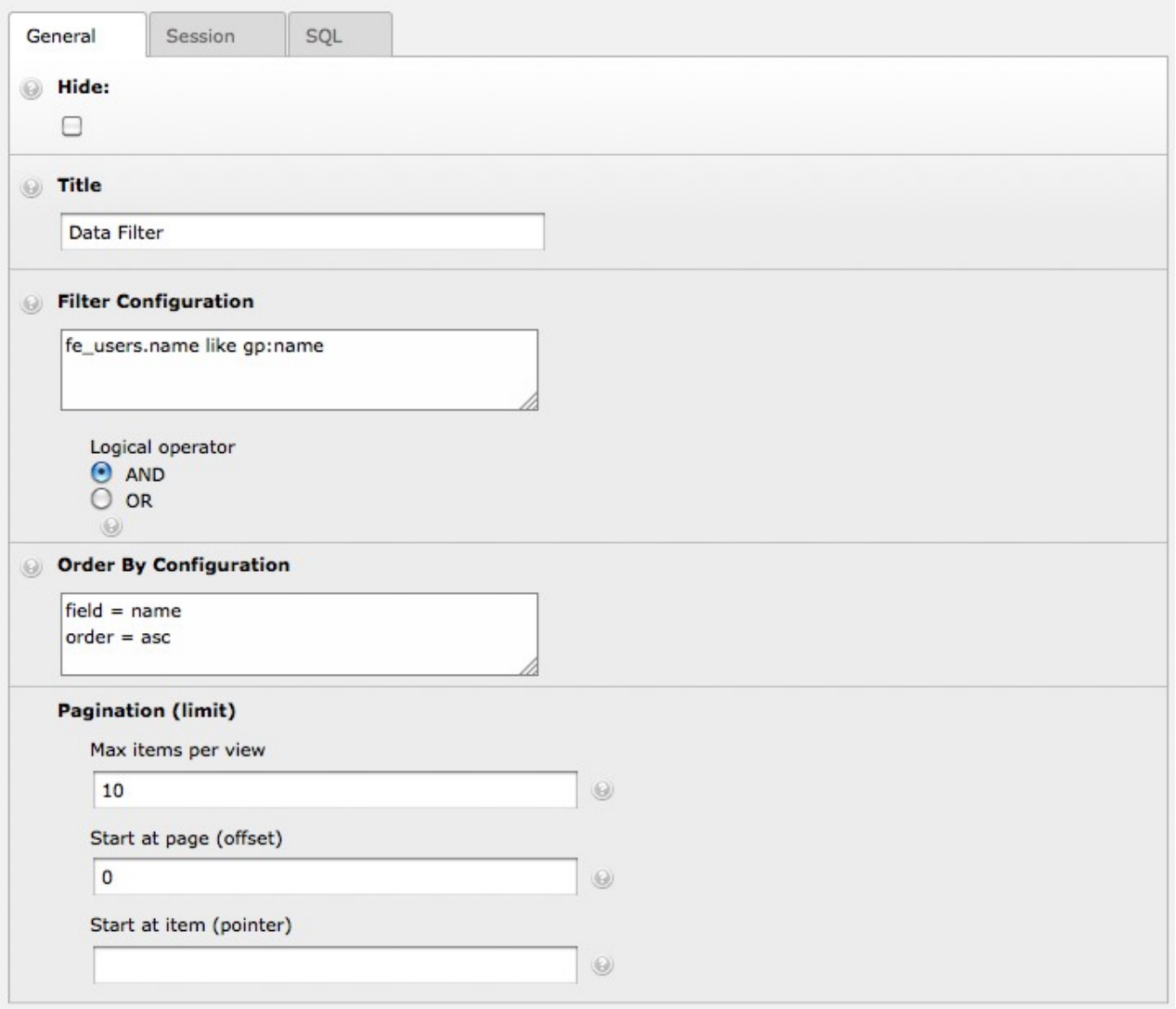
Users manual

Defining a Data Filter requires mostly to know the syntax for configuring the filter. This is loosely inspired by the syntax of the TypoScript getText function.

Input screen

This section describes the main input screen. The actual filter syntax is described below in "Filter Configuration".

NOTE: the "SQL" tab is added by the "dataquery" extension, so you may not see it if you don't have this extension installed.



Here are the various input fields:

- **Hide**: this flag is not currently used. It could be in the future. In the meantime it can still be used to indicate an unused filter.
- **Title**: a name for the filter, should be explicit enough to know what the filter is about.
- **Filter Configuration**: the filter conditions themselves. See "Filter Configuration" below.
- **Logical operator**: the logical operator that will be applied between each filter condition. The default is "AND".
- **Order By Configuration**: see "Ordering Configuration" below.
- **Pagination**: see "Limits" below.

Filter Configuration

You may define as many filters as you want, each on a line. All filters will be linked to each other using the selected logical operator (AND or OR). The general syntax of a filter is the following:

```
[extra keyword.][[table name].][field name] [operator] [value or expression]
```

Example:

```
uid = 12
pages.uid = page:uid
main.tt_content.header like foo
```

In both lines above, we are defining that the "uid" field of the "pages" table must be strictly equal to some value. In the first case, it is a simple number (12), that will be used as is. In the second case, the test value will be retrieved as the uid of the current page.

It is also possible to omit the table name. It is up to the Data Provider to know what to do in such a case. In the case of Data Query, the condition will be considered to apply to the main table (the one in the FROM clause).

Extra keywords

There are two extra keywords that can be used before the table name: **main** and **void**. Usage of a special keyword (and its dot) is optional.

The "main" keyword means that the condition should be applied to the "main condition" by the Data Provider. What this exactly means depends on the Data Provider. Let's consider Data Query as an example. This Provider will transform the conditions into SQL statements. The query in the provider will have a main table (the one in the FROM clause). It may also have secondary tables (linked using JOINs). By default filter conditions that relate to the main table are applied in the WHERE clause and those that relate to secondary tables apply to the ON clause of the relevant JOIN statement. Using the "main" keyword will force a condition that would normally be applied inside an ON clause to be moved to the WHERE clause.

The "void" keyword means that the condition should be evaluated but not applied. The Data Provider receiving this condition should simply ignore. Such "void" filters are useful to pass some filter configuration down the component chain. Indeed since Data Providers include the filter information inside the data structure, it will be passed on to the Data Consumers. This makes it possible to get filter information in the Data Consumer, in order to display it or to react to it.

Using a void filter

The usefulness of "void" filters may not be obvious at first. An example may help. Imagine a list of blog entries, which can be filtered on month and year. Both the list of entries and the list of possible years and months to select from are displayed using Tesseract. When a month/year combination is selected, the list of blog entries is filtered accordingly.

We would like the list of months and years to know about the selection too, so that we can highlight the current selection. But we don't want the month/year selection to apply to the months/years selector itself. By defining conditions on the month and year with the "void" flag, the month/year selector can know about the current selection without being affected by it.

Operators

The following operators are can be used:

Symbol:	Usage:
=	Strictly equals to
<=	Smaller than or equals to
>=	Bigger than or equals to
<	Strictly smaller than
>	Strictly bigger than
!=	Not equals *(since version 1.3.0 the ! sign is actually used to negate all operators)*
in	In a group of values (think of the SQL IN operator)
andgroup, orgroup	In a group of comma-separated values (AND or OR for each value depending on choice of operator)
like	Similar to the SQL LIKE operator. The Data Provider is expected to wrap the value in wildcards.
start	Should be similar to the SQL LIKE operator, but testing the start of a word. This means the Data Provider is expected to append a wildcard to the value.
end	Should be similar to the SQL LIKE operator, but testing the end of a word. This means the Data Provider is expected to prepend a wildcard to the value.

Since version 1.3.0 all operators can be negated by prepending them with an exclamation mark (!). Example:

```
tt_content.title !like foo
```

Expressions are managed by the "expressions" extension. Any expression described in that extension's manual can be used here.

On top of that some special values can be used in Data Filter. Special values all start with a backslash (\).

Key:	Explanations:

\empty	This special value is equivalent to the empty string (''). The following will fail:
	
	`pages.title !=`
	
	because an empty value will cause the entire line to be ignored. Use the following instead:
	
	`pages.title != \empty`

\null	This special value is meant to be matched to an undefined or unset variable. It only makes sense with the "=" and "!=" operators. Any other operator than "=" should be interpreted as "!=" in this case.
\all	This special value indicates that the condition should actually not apply, because we want all values. This is sometimes necessary when all values should be explicitly retrieved, ignoring any default value for the filter. **Example:** `tt_content.CType = gp:content_type // text` In the above example, all content elements from a given type will be selected, according to the value found in the GET/POST variable called "content_type". If this variable is empty, the type defaults to "text". In such a case it is not possible to see all content elements, except by setting the value of the GET/POST variable to "\all".
\clear_cache	This special value can be used to clear the filter cache. See "Filter cache" below for explanations.

The result of an expression can be a simple value (a number or a string, for example), but also a more complex type:

Type:	Examples:	Explanations:
interval	[10,20] *A value between 10 and 20, inclusive*]0,1000] *A value strictly bigger than 0 and smaller or equals to 1000* [5,*] *A value greater or equals to 5, with no upper boundary*	When an interval is defined, the original operator will be ignored and replaced with operators based on the type of square brackets used in the interval. "*" can be used for not setting a limit.
comma-separated strings of values	1,5,6,12 *The value should be either 1, 5, 6 or 12*	Such a value is used to defined a group of values among which at least one must be matched. It is up to the Data Provider to set up proper tests for this. This kind of value will normally be used in conjunction with the "in", "andgroup" and "orgroup" operators.
array	-	Data Providers are expected to know how to handle array values. The expected behavior is to loop on each value and handle them with the same operator, with all resulting conditions "added" together.

Comments

It is possible to use comment markers inside the filter configuration. Any line starting with "#" or "//" will be ignored.

Multiple values

An expression can also contain multiple values. This is useful for selecting alternate values if some are not defined, and in particular for setting default values.

Examples:

```
pages.title = page:nav_title // page:title
```

This expression will return the nav_title of the current page or its title, if no nav_title is defined

```
sys_language_uid = tsfe:sys_language_content // 0
```

This will return the id of the current language or 0 if we are using the default language

The order is always left to right. If the first value is not defined, we pass on to the next one to the right, etc.

Interpolated values

There can be expressions within expressions. These are called "subexpressions". An example application is to select a table or field name dynamically.

Example:

```
pages.{vars:field_name} = [5,10]
```

This expression will match the field (from the "pages" table) defined in the variable "field_name" to a value between 5 and 10.

Named configuration lines

Inside a filter, configuration lines are numbered according to their position in the text field, i.e. the first line is numbered 0, the second line 1, etc. This internal numbering is useful when a filter needs to be updated, that is when it is called again without the cache being cleared.

However it is also possible to explicitly give a name to a configuration line so that it is easier to target. This is particularly useful in a filter which tries to retrieve information from another filter stored into the session.

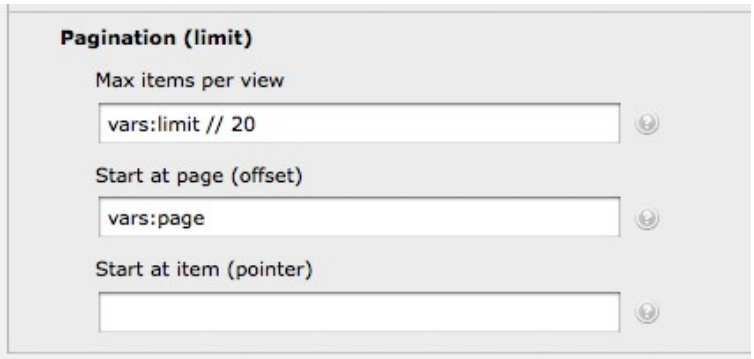
Naming a configuration line uses the following syntax:

```
interval5-10 :: pages.{vars:field_name} = [5,10]
```

In this case the line will be named "interval5-10".

Limits

A limit is comprised of two parts: the maximum number of elements to display and an optional offset, which will indicated how many elements from the start the Data Structure should really start. Both maximum and offset can use expressions, as defined above for filters, but do not need to match a field. So a typical configuration might look like that:



Note that the offset is not a number of elements, but a multiplier of max. Example: say the above configuration returns 20 for limit and 3 for offset, the actual number of records to shift in the Data Structure in 3 * 20 = 60.

Ordering configuration

The sorting of records can also be defined inside the Data Filter. There again the syntax for expressions can be used. The general syntax, however, is a bit different, since both a field and a sort order must be defined for each sorting criterion. So an actual ordering definition might look like:

```
field = tx_specialsearch_pi1|sort // items.name
order = tx_specialsearch_pi1|order
```

In this case the name of the field to use for ordering will be fetched from variable `tx_specialsearch_pi1[sort]`. If not defined, the default search will use the "name" field from the "items" table. The sorting order will be fetched from variable `tx_specialsearch_pi1[order]`. The order may be omitted and the Data Provider is expected to use some default sorting order.

Random ordering

For random ordering, the simple keyword "\rand" can be used. There's no need to define any "field" or "order" (as per the above configuration). Just a single line with the keyword:

```
\rand
```

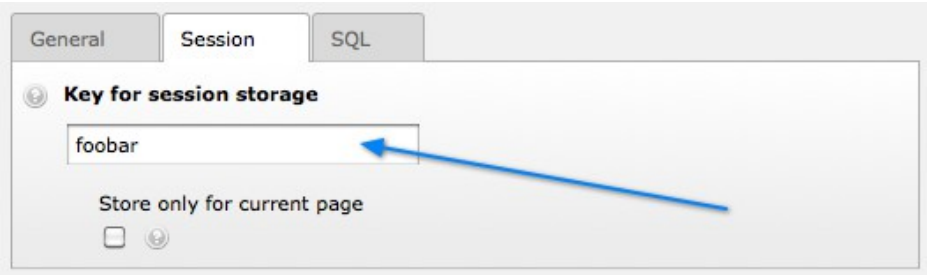
Any appearance of the "\rand" keyword will trigger random ordering. Thus the following lines have the same effect:

```
field = \rand
order = \rand
```

Note that all other ordering configurations will be ignore if the keyword "\rand" is found on any line.

Session storage

A filter can be stored into the session with the purpose of being accessed by another filter. To force a filter to be stored into the session, one must simply give it a session key (in the "Session" tab of the input screen).



The next time this filter is called up, its result will be stored into the session with the key "foobar". Let's assume the following configuration:

```
interval5-10 :: pages.{vars:field_name} = [5,10]
```

The information will be stored in the following format:

filters	pages.uid	interval5-10	condition	= [5,10]
			operator	=
			value	[5,10]

Operator and value are stored separately, but as a convenience they are also stored concatenated together. The advantage is that you can use a single subexpression to get both the operator and the value.

Note: if the value is an array, it will be transformed into a comma-separated string. This will obviously fail for multidimensional arrays, but it was decided to be an acceptable limitation. Indeed parsed filter values are meant to be retrieved using expressions (as per example below), so they must be of a simple type. Multidimensional could have been serialized, but this value would have been unusable afterwards. Anyway it is not clear how Data Providers would interpret multidimensional arrays.

Example:

```
pages.uid {session:foobar|filters|pages.uid|interval5-10|condition}
```

which will be interpreted as:

```
pages.uid = [5,10]
```

and then evaluated as a filter configuration line.

Filter cache

Once parsed a filter is stored into session. This is not related to the session storage mechanism mentioned above. It is used automatically so that values from a previous parsing are preserved. This is useful if you want to change just one value in a filter and preserve the others.

The drawback – as with any caching mechanism – is that this cache sometimes needs to be cleared. For example, if the filter relates to a search form, the filter cache must be cleared when a new search is performed. This can be achieved by sending "clear_cache = 1" either as a GET or POST parameter. In the example of a search form, this could be a hidden field in the search form.

There's also a possibility to clear the cache more selectively, for a single value, using a configuration like:

```
tt_news_cat.uid = vars:showUid // \clear_cache
```

In this example, if there's no showUid to be found in the vars, the value returned by the parser will be "\clear_cache". The Data Filter will act upon this and remove that value from the session cache.

Developers manual

Hooks

This extension provides two hooks that can be used by developers:

- **postprocessReturnValue**: this hook comes at the end of the filter's evaluation. It receives a reference to the `tx_datafilter` object. This makes it possible to access (using `getFilter()`) and modify (using `setFilter()`) the current values of the filter.
  Any class meaning to use this hook must implement the `tx_datafilter_postprocessFilter` interface.
- **postprocessEmptyFilterCheck**: this hook comes at the end of the empty filter check. It can be used to modify the result of the `isFilterEmpty()` method. It receives as parameters the current value of the empty filter check (a boolean) and a reference to the calling `tx_datafilter` object. It is expected to return a boolean value (true if the filter can be considered to be empty, false otherwise).
  Any class meaning to use this hook must implement the `tx_datafilter_postprocessEmptyFilterCheck` interface.

Known problems

If you have any issues, please refer to the Tesseract Project web site (http://www.typo3-tesseract.com/). You may also post your problems to the TYPO3 English mailing list (typo3.english), so that others may benefit from the answers too. For bugs or feature requests, please open an entry in the extension's bug tracker on Forge (http://forge.typo3.org/projects/extension-datafilter/issues).

To-Do list

The roadmap for the evolution of this extension can be found on Forge: http://forge.typo3.org/projects/roadmap/extension-datafilter

For feature requests, please open a report on Forge issue tracker: http://forge.typo3.org/projects/extension-datafilter/issues