MetaModels Documentation

Release 2.0.0

Team MetaModels

Sep 02, 2017

Contents

This is the official documentation of *MetaModels*, an extension for the Contao Content Management System.

This documentation is split into three sections:

In the *Manual* you find general documentation about MetaModels.

In the *Cookbook* you find specific solutions for specific needs.

In the *Reference* you find reference information like a list of events.

If you want to contribute, too, please click the „Edit on GitHub“ link in the upper right corner or visit our GitHub project page.

CHAPTER 1

Manual

Introduction to MetaModels

What are MetaModels?

MetaModels is an extension for the Contao CMS. This extension enables you to input a large variety of structured data and display it on your website following different criteria such as list and detail views, filtering, sorting, pagination, multilingualism and many more..

“Structured data” means content, which is usually stored in a database scheme with different tables and relations. MetaModels supports different types of field types (attributes) as for example text, selects, check boxes, radio buttons, integers/decimal, yes/no fields, file selection etc.

Possible applications for such data content are in the fields of product catalogues, events, menues, adress or employee lists, houses and rental properties, picture galleries or multilingual text/image content.

With MetaModels, the data models can be created completely in the Contao backend. There is no need for you to code, as for a specific extension. Both the creation of the input masks for the backend as well as the output for the frontend with optional filters belongs to the creation of a MetaModel.

The MetaModels extension features a high flexibility for data input and output and thereby covers a lot of specific needs. You can find more details in rst_features. Also have a look at some MetaModels show cases or check the Contao forum for further show cases introduced in the german forum.

History of MetaModels

Metamodels started out as the next generation of the famous Catalog extension.

Over time ‘Catalog’ developed into a complex extension and offered many possibilities combined with Contao. But unfortunately it became more and more difficult to maintain the code and to implement new functions.

From the experiences we gained with the development of Catalog 1 and Catalog 2, it became clear to us, that we needed to start from scratch.

That’s why we developed “MetaModels”: a totally new extension influenced by many modern programming paradigms. Our goal was to develop an extension with a flexible and extensible code base.

The current Metamodels version 2.0 is the result of many hours of discussion about what is “the best solution” and hard programming work.

MetaModels in comparison with other tools

MetaModels works well with the division of labour between administrator and editor which means: the administrator or developer creates one or multiple MetaModels with input masks and output functions and the editor(s) can add the content as they are already used to from other areas of the Contao backend.

The input masks allow you to accurately specify which data has to be entered (or can be entered) and how. The extensions "[dma_elementgenerator]" or "[rocksolid-custom-elements]" also provide similar functions. The difference is that MetaModels will allow you to even display complex data structures and additionally provides you with various functions for output and filtering.

Before starting a new project you might wonder whether it is better to develop your own extension instead of using MetaModels. But there is no general answer to this question, because both solutions will enable you to solve various problems. The following aspects might help you to make your decision:

Pro developing own extension: Is it required to develop a product which can be marketed, as for example a commercial extension which can be made available to other Contao users at the push of a button? Then you should consider developing your own specific extension. The basic requirements to do so are appropriate skills in PHP programming and knowledge of the Contao API.

Pro MetaModels: In case that you want to implement a very individual solution which can be quickly customised in the Contao backend, MetaModels is certainly a good choice. If you also need specific functions e.g. supporting multilingualism, MetaModels can play fully on its strengths. MetaModels supports users to develop a solution without programming. But it should be noted that only with some basic knowledge in PHP, HTML, and SQL databases, you will be able to make fully use of the opportunities provided by MetaModels.

Resources

- MetaModels project website
- MetaModels on Github
- MetaModels manual on Github
- MetaModels Contao Wiki
- MetaModels Contao community subforum
- MetaModels IRC Channel on freenode #contao.mm

Install and update MetaModels

You will need a Contao-LTS-version to be able to install MetaModels - the current version is Contao 3.5.x

Installation via Composer

MetaModels and all its dependencies can be installed with the Composer package manager in the Contao backend.

If your Contao installation is already using the new Composer package manager, you can easily install MetaModels by selecting, respectively typing in the package name into the search field, as follows:

- metamodels/bundle_all

Regarding the bundle, you will have to select version "2.0.x" at the moment - this bundle will automaticaly install the complete "MetaModels core" with it. While selecting restrictions you can choose between different stages, such as "bugfix release", "feature release" etc. - the current MetaModels functions will be activated with "feature release".

In case that you don't need all filters and attributes, you can also install them separately or you can select another Bundle package. The packages mentioned above are grouped together and should meet most requirements.

You can find an overview of your already installed packages in the display of the dependency graph (checkbox) in the Contao Composer client ("Package management").

Installaton via Nightly build

Alternatively to the installation via Composer, you can install MetaModels via FTP. To do this, you will have to download the current version of MetaModels from the project website http://now.metamodel.me/ unzip it and upload it via FTP to your server. Most of the folders have to be saved into the folder */system/module* - only two PHP files which are supporting the Ajax functions have to be saved into the Contao root directory.

Afterwards you will have to update the database in the "extension manager". If the following error message appears `Fatal error: Class 'MetaModels\Helper\UpgradeHandler' ....! metamodels-tng-branch/config/runonce_0.php` you should purge the internal cache. This option can be found in the menu item "Maintenance" of the Contao backend.

Testing of special packages via Composer

The bundle 'bundle_all' contains all currently available and released MetaModels packages. Additionally there are packages with bugfixes or brandnew functions that have to be tested. For the MetaModels core this could be e.g. a package called "dev-hotfix-xyz". You can see those packages inter alia on Github within the corresponding repository (e.g. MetaModels/core) in the 'branches' tab.

In case that you want to test a package like this, you'll have to separately select and install it in the package management. For the selection in the package management, check the checkbox "dependencies i nstalled" and then click on the corresponding package, e.g. 'metamodels/core' and aditionally in the following options click on e.g. 'dev-hotfix-xyz'.

After "Reserve package for installation" you'll have to make some small changes to Composer-JSON. To do this go to the package manager to "settings" and there click onto "expert mode". The displayed JSON file has to be extended with the entry "as 2.0.0" within the node "require". If you happen to have several extra packages you have to do this for every entry.

for example: `"metamodels/core": "dev-hotfix-xyz"` modify to `"metamodels/core": "dev-hotfix-xyz as 2.0.0"`

After the installation via "update packages" you should delete the Composer cache in the "settings" of the package management.

As MetaModels is closely interlinked with the DC_general (DCG), you will frequently need to update to a newer version here as well for testing. The procedure is the same as for MetaModels including the adjustment of the JSON entry with the "as 2.0.0".

To come back to the initial version , just delete the package in the package management.

Please never forget to provide the MetaModels developer team with your valuable feedback after your test on Github.

Update MetaModels

If you installed MetaModels via Composer you will also have to update it that way.

With the manual MetaModel installation you have to consider several aspects. The following procedure has shown itself to be the most efficient:

- delete ALL old MetaModel folders (you can check which folders these are by double-checking the previous download) - really **ALL**
- Clear the Contao cache -> /system/cache (everything there within this folder)
- **NEVER EVER** do a database update (otherwise all will be gone)
- download the new nightly build files, unzip and upload them (via FTP)

- update the database via /contao/install.php

You can find the most current information in the forum

Switch MetaModels from "Nightly build" to "Composer"

The procedure is similar to "Update MetaModels". By switching to Composer you should consider that Composer is a memory intensive application. Based on experience, you should have at least 100MB. The exact required memory size is dependent on other installed packages and also on the server configuration of your provider.

The following procedure has shown itself to be the most efficient:

- install Composer
- delete ALL old MetaModel folders (you can see which folders these are by double-checking your previous "nightly" download) - really **ALL**
- Clear the Contao cache -> /system/cache (everything there within this folder)
- **NEVER EVER** do a database update (otherwise all will be gone)
- in Composer select the desired MetaModel version, reserve for installation, then install
- the database update should then automatically being proposed and done

You can find the most current information in the forum .

The first MetaModel

Install with composer

You'll need the MetaModels core and some attributes / filter to get MetaModels running. In you composer search `metamodels/core` an `metamodels/bundle_all` to install the core and all bundles and filters. Don't forget to run composer install through „Update packages". When installed, run the database update and your MetaModels installation is done.

Note: If you know that you don't need all attributes and/or filter you can install every single package by it's own.

Create MetaModels

To get started with MetaModels we need at least one MetaModel, jai! We will build a small MetaModel, non translated, MetaModel for real estate references.

In our example we need two MetaModels:

reference (the MetaModel which contain the real estate objects)

category (the MetaModel to define categories for references)

Create reference and category metamodels.

Create attributes

An (empty) MetaModel is just a container for your data objects. But before you can store data in your MetaModel, you need to define some types of data which you like to store.

In MetaModels there are several „attributes" to store different kind of data. Most of the time you need at least a text attribute (e.g. to store a name).

mm_reference

Our reference will contain these attributes:

- Name (text)
- Alias (alias)
- Published (checkbox)
- Description (longtext)
- Keyfacts (tabletext)
- Category (multiple select)
- Highlight-Picture (file)
- Picture Gallery (file, multiselect)

Name

Attribute Type text

Column Name name

Name Name

Description Name of reference

Alias

The alias is an (optional) unique Name / identifier for the data record.

Attribute Type alias

Column Name alias

Name Alias

Unique Yes

Description Alias of reference

Alias-Fields Name [text]

Published

Attribute Type checkbox

Column Name published

Name Published

Published yes

Description

Attribute Type longtext

Column Name description

Name Description

Description Description of reference

Keyfacts

Attribute Type tabletext

Column Name keyfacts

Name Keyfacts

Label Entry

Width 500

Category

Attribute Type multi select

Column Name category

Name Category

Description Select a category for the reference

Database table mm_category

Currently, we haven't added attributes to our `mm_category` MetaModel. So for the moment leave the other selects blank, we'll get back later.

Highlight picture

Attribute type file

column name picture_highlight

Name Highlight picture

Customize filetree (optional) select a „content" folder where the reference pictures are stored

Gallery

Attribute type file

column name picture_gallery

Name Gallery

Customize filetree (optional) select a „content" folder where the reference gallery pictures are stored

multiselect yes

mm_category

For our category MetaModel we just need four attributes:

- name (text; „name")
- alias (alias; „alias")
- published (checkbox; „published")
- description (longtext; „description")

Create the attributes as you have just learned in the reference MetaModel.

Select configuration

Early, we introduced in our „reference" MetaModel a select attribute but leaved it's configuration nearly blank.

The real power of MetaModel now gets obvious here. With a simple select attribute you can easily connect MetaModels (or any other sql-table) and optional filter the objects. Filter...? We'll talk about this later.

Edit the „multi select" attribute in your „References".

Choose:

table mm_category

Name name - text

Alias alias - alias

Sorting sorting

Create Rendersettings

For now, we have two MetaModel with some attributes and a link between booth. But we didn't want just to store some data, we also like to display them.

A render setting contains some global settings, attributes you like to display and there settings. No matter if you like to display data in the backend or fronted you need at least one render setting. But we recommend to create at least one setting for the backend and one for the frontend.

Note: Prefix your render setting name with BE / FE for easy relocating*.

Basic-settings

Note: MetaModels provides a set of well organized, solid templates. There are templates for each render setting (e.g. metamodel_prerendered). You can create your own templates the contao why (Backend > Templates > Create > select the template you like to overwrite > Save (maybe with a new / name addition) > Edit > Choose)

-metamodel_prerendered All attributes are rendered with there template and settings (if available) -metamodel_unrendered All attributes are rendered in „raw“ to the frontend (the settings of the child attributes are

ignored)

Output Format:

-HTML 5 Renders as HTML5 content (This is the default format in Contao and therefore suggested). -XHTML Renders as xhtml (this format is deprecated in Contao and therefore not suggested). -Text Renders the „content“ as plaintext.

Jump-to-Page

The jump-to-page comes into the game when we like to display our references as list with a detail link to one item. So you need to define a jump-to-page where the user gets redirected if he clicks on a „detail“ link of one of our reference objects.

The filter setting define the rules for the target, your detail page.

Expert-settings

hide empty entries yes

hide labels yes

Create a rendersetting (backend)

Go to the „render settings“ of „reference“.

- Create a render setting called „BE: references“
- Add „all attributes“
- After adding, activate „name“ and „category“

Note: When you (later) add attributes to your MetaModel you need to add them also in your render setting.*

Create a rendersetting (frontend list)

Go to the „render settings“ of „reference“.

- Create a render setting called „FE: references list“

- Add „all attributes“
- After adding, activate „name“, „category“, „picture_highlight“

Create a rendersetting (frontend detail)

Go to the „render settings“ of „reference“.

- Create a render setting called „FE: reference detail“
- Add „all attributes“
- After adding, activate „name“, „description“, „category“, „picture_highlight“, „picture_gallery“

Input Screens

For now there are two MetaModels with some Attributes and Rendersetting. But how do we get data in our MetaModels? With input screens!

Input Screens could hold a collection of these attributes which are necessary to grep some data. Most times you just add all attributes in one Input Screen, but with the power of different input screen you can create different edit masks for different kind of user(groups).

But in our tutorial we just need one input screen for our users.

Basic-settings

So create a Input Screen with the following settings:

Name BE: References

Standard yes

Panel-Layout -leave this empty-

Integration standalone

Backend-Section Content

Render mode Flat

Data manipulation permission We want to allow editing, creating and deleting items - so choose all three.

Select configuration

Okay. Now we got the empty Input Screen container with a few settings. But to get things working, we need (remember the render setting!) some attributes in it.

Switch to the „settings“ of your currently created Input Screen and choose „add all“.

Define Attribute settings

Our input screen is ready. But we need tweak the attributes a little bit. For example we always want a name, description and Highlight Picture.

To get this done, we choose in these attribute settings the „mandatory“.

Grouping and sorting settings

In the grouping & sorting section you need to create at least one object to sort & maybe group your entries.

For example: “Enable manual sorting” without grouping.

View conditions

View conditions are the easy part in MetaModels. But, you might guess that you also need here at least one to get things work.

The view conditions define who could see and use which render setting and input screen.

Define a view condition

Define one view condition with following settings:

member-group -leave this empty-

user-group administrator

input screen BE: Referenz

Rendersetting BE: Referenz

We are ready to enter Data

Some time ago, we started with just a MetaModels package and already arrived to create data. Easy, hm?

Continue to the new „Referenz“ entry in your „content“ navigation and add a first item.

Filter Setting

(Todo)

CHAPTER 2

Cookbook

MetaModels “cookbook”

Our MetaModels “cookbook” provides you with numerous snippets, tips and tricks on how to best use MetaModels.

We are happy to include interesting and creative solutions to our list - please send your “receipts” or links to the forum or other websites to the following email:

MetaModels checklists

Short checklists for you to review if s.th. doesn’t work as expected.

Start with MetaModels

You should consider some basic things when you start with MetaModels.

The MetaModels project is running quite stable - nevertheless it is in constant development. In interaction with other components, such as the DC_general (DCG) or the Contao core, there may be a data loss. That’s why it is highly recommended to set up a regular backup.

Checklist:

- Did you install the current version of MetaModels and DCG (preferably via Composer)?
- In Contao “System settings” activate the checkboxes “Bypass the internal cache” in the section “Global configuration” and also “Display error messages” in the section “Security settings”. Subsequently purge all the caches.
- Set up a regular backup
- For known bugs and errors take a look on our forum or on Github

Filter is not displayed

A desired filter is not displayed on the website.

Checklist:

Did you create the filter setting?

Did you enable the filter setting?

Is the filter setting selected in your FE or CE module ("Filter settings to apply")?

Is the filter rule activated in your FE or CE module (attributes)?

Is the FE or CE module set to activated/visible?

An attribute isn't displayed following a modification

After the modification of an attribute (e.g. attribute type) it isn't displayed (anymore) on the website.

Checklist:

Check the attribute listings in the render settings and input screens

Delete the respective attribute in the render settings and add it as new again

If necessary, enter the values again into the input mask after the modification.

Check the debug output whether the attribute is output in the template

Input mask: populate fields with pre-defined values

The input fields in the input masks can be already filled in with pre-defined standard values. This can greatly facilitate data entry for the user, when creating new data records.

The metamodels input fields can be (almost) used in the same way as the input fields of the Contao core or common Contao extensions which have been created with a DCA array. There are some differences because MetaModels generates fields dynamically by the dc-general.

You can create pre-defined values with the key "default" added to the dc-array. The dc-array can be amended either by an entry in the file "dcaconfig.php" in the folder "/system/config/" or if there is an own module folder in the file "config.php".

Appropriate entries are already set up in the module "Metamodels-Boilerplate" in the file "config.php".

To enter a pre-defined value, you need to know the (internal) name of the MetaModel and the column name of the attribute. This informations may be given with an array entry in the following general form:

```
<?php
$GLOBALS['TL_DCA']['<MM-Table-Name>']['fields']['<Field-Column-Name>']['default']␣
↪= <Value>;
```

E.g. for an email field ([text]) from mm_first_index the default value could be set up like this:

```
<?php
$GLOBALS['TL_DCA']['mm_employeelist']['fields']['email']['default'] = '@mmtest.com
↪';
```

There are specifications for individual attribute types. Here is in which form the values are expected:

- **Text**: Text in inverted commas e.g. '@mmtest.com' `...['default'] = '@mmtest.com';`
- **Timestamp**: Integer for the timestamp e.g. 1463657005 or PHP function time() `...['default'] = 1463657005;` or `...['default'] = time();`
- **Select**: Integer of the ID of the value in inverted commas `...['default'] = '2';`
- **Multiple selection**: Array with alias values from the selected alias column `...['default'] = array('purchase', 'marketing');`
- **Checkbox**: true `...['default'] = true;`

As you can see from the attribute “Timestamp”, dynamic specifications are feasible. It would be possible to use existing values from MetaModels and to output them - if necessary with a calculation - as default. The methods of the API (ref_api_interf_mm) are available to you in order to access MetaModels.

Input screens: custom RegEx test

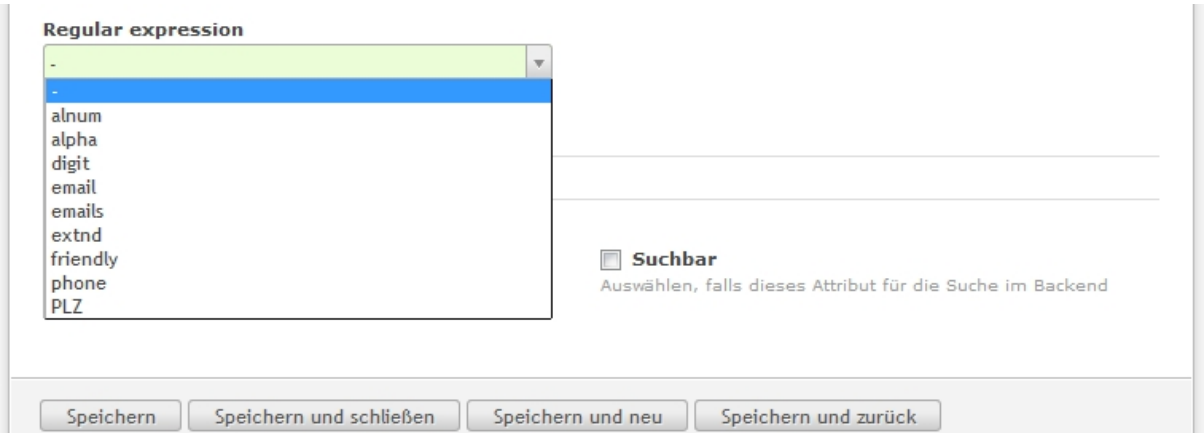
You can implement your own RegEx validation for a text input field in an input screen with the following event listener.

To implement it, respectively to activate it for the field in the input screen, this validation must be made available for Contao on-board functionality.

In order to do so, we’ll create the following hook “addCustomRegex” as follows - see API: addCustomRegex

- create a folder for your custom module in /system/modules - e.g. “/metamodels_mycustoms”
- in the folder metamodels_mycustoms add two more folders named “/config” and “/classes”
- in the folder /classes add the file “MyClass.php” as described in Contao API
- in the folder /config add the file “config.php” as described in Contao API
- additionally in the folder /config the file “event_listeners.php” - the key of the array $options must be the same as the value obtained from testing of $strRegexp in /MyClass (‘zip’)
- after you have created all the files and filled them with code, you can create the autoload.php by using “Autoload creator” under “developer tools” in the Contao back end.

The entry “ZIP” should now be available in the settings of an input field of an attribute of type “text” when the Regex test is selected. If not, purge all the caches in the back end and check the data if necessary.



Source codes

You’ll find the following source code in the files:

File /system/modules/metamodels_mycustoms/classes/MyClass.php

```
<?php
class MyClass
{
    public function myAddCustomRegexp($strRegexp, $varValue, Widget $objWidget)
    {
        if ($strRegexp == 'plz')
        {
            if (!preg_match('/^[0-9]{4,6}$/', $varValue))
            {
                $objWidget->addError('Feld ' . $objWidget->label . ' should␣
↪contain a valid ZIP postcode.');
```

```
            }

            return true;
        }

        return false;
    }
}
```

File /system/modules/metamodels_mycustoms/config/config.php

```
<?php
$GLOBALS['TL_HOOKS']['addCustomRegexp'][] = array('MyClass', 'myAddCustomRegexp');
```

File /system/modules/metamodels_mycustoms/config/event_listeners.php

```
<?php
use␣
↪ContaoCommunityAlliance\DcGeneral\Contao\View\Contao2BackendView\Event\GetPropertyOptionsEvent;
↪

// Event Listener with priority "-1"
return array
(
    GetPropertyOptionsEvent::NAME => array(
        array(
            function (GetPropertyOptionsEvent $event) {
                if (($event->getEnvironment()->getDataDefinition()->getName() !==
↪'tl_metamodel_dcasetting')
                    || ($event->getPropertyName() !== 'rgxp')) {
                    return;
                }

                $options = $event->getOptions();

                // Key "zip" equals $strRegexp test in myAddCustomRegexp
                $options['zip'] = 'ZIP';

                $event->setOptions($options);
            },
            -1
        )
    )
);
```

The file autoload.php in /system/modules/metamodels_mycustoms/config should look as follows after its generation:

```
<?php
ClassLoader::addClasses(array
(
    // Classes
    'MyClass' => 'system/modules/metamodels_mycustoms/classes/MyClass.php',
));
```

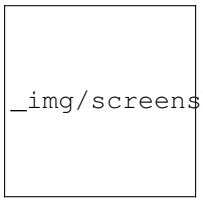
Notice: the RegEx validation was taken from the Contao manual und represents just a simple test method for german ZIP codes. You can find more accurate RegEx checks online or you could also implement a check against a list with assigned zip code numbers.

View condition: Display s.th., if the checkbox is not activated

If you want to create a view condition, which enables you to dispay a field, if a checkbox is **not** checked, this will be not possible with a trigger on the "inactive" value of the checkbox.

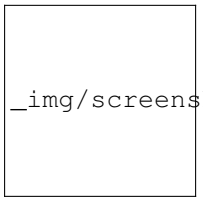
This is due to the fact that MetaModels treats the value "unchecked" differently from the Contao core - the Contao core will store nothing '' for "unchecked" instead of a null(0). This can not be processed by MetaModels or the DCG at the moment.

This problem can be fixed with a workaround: The visibility is triggered by "checked", but the test is inverted with NOT. To achieve that, a condition NOT has to be created in the view conditions and inside this condition the test whether the checkbox is "active" (see screenshot).



The following two screenshots show the hiding of an email input mask with the checkbox set.

Email shown



Email hidden

Debug templates

If you need a custom template - e.g for displaying a frontend list - or if you want to find out which attribute values are sent to the template, you can print those attribute values out to the HTML source code. An easy way to do this is the output of the item array with "print_r" .

The default template is "metamodel_prerendered" or respectively the template, which was selected in the output render settings.

In case that there is no custom template in use yet, you will have to create a copy of "metamodel_prerendered" within the Contao folder named "Templates".

The following code is added to the respective template:

```
<?php
echo "<!-- DEBUG START \n";
echo "<pre>\n";
print_r($this->items->parseAll($this->getFormat(), $this->view));
echo "</pre>\n";
echo "\n DEBUG END -->";
?>
```

Subsequently the template should start with the code as follows:

```
<?php
echo "<!-- DEBUG START \n";
echo "<pre>\n";
print_r($this->items->parseAll($this->getFormat(), $this->view));
echo "</pre>\n";
echo "\n DEBUG END -->";
?>

<?php $strRendersettings = isset($this->settings)? 'settings' : 'view'; ?>
<?php if (count($this->data)): ?>

<div class="layout_full">

<?php foreach ($this->data as $arrItem): ?>
<div class="item <?php echo $arrItem['class']; ?>">

<?php foreach ($arrItem['attributes'] as $field => $strName): ?>
//...
```

If the website with this listing is called in a browser, you should find the debug output in the source code.

Browser rendering can become very slow in case that the output is very extensive. It might then be helpful to output only one item node.

```
<?php
echo "<!-- DEBUG START \n";
echo "<pre>\n";
// only first node
print_r($this->items->parseAll($this->getFormat(), $this->view)[0]);
echo "</pre>\n";
echo "\n DEBUG END -->";
?>
```

You can remove the output by commenting out the output block, by deleting it or by switching to another template.

CHAPTER 3

Reference

MetaModels API

The MetaModels API consists of serveral interfaces which are the only API that should be considered immutable. Classes of the core and their private, protected and even public methods should generally NOT be considered immutable and may be changed over minor versions and patch releases.

During the alpha and beta phase of a new MetaModels major release, there may be changes to interfaces as well. Therefore the API should not be considered immutable during major development cycles.

An deprecation phase will be provided during minor cycles, denoting that a certain feature of the API will get dropped in the next major release. We will try to put the replacement already in place but for bigger breaks this will not be possible. The breaks will however be announced in an draft, along with an upgrade guide, prior to release as soon as the new interfaces are defined.

Core Interfaces

MetaModels reference

This reference is mainly intended for developers that want to enhance MetaModels with own attributes and/or filters etc.

CHAPTER 4

Indices and tables

- genindex