MetaPortal Module API

Thorsten Kunz

December 9, 2003

Contents

1 Introduction

This is supposed to be a documentation for MetaPortal Module Developers. It describes the basic interfaces provided by the framework and contains informations about how to use them. With the informations provided here you should be able to write you first modules within a short amount of time. If you want to use more complex modules and access more of the provided libs you shoudl read the source! The framework and thus the module API heavily uses Smarty templates and the pear database abstraction. You can find dokumentation for these additional packages at http://smarty.php.net for the smarty template engine and at http://pear.php.net for the pear docs. If you don't know them yet we highly recommend you take a brief look at them before you continue here. Else you might have a few problems to understand some points of the api.

1.1 Modules

A module can be considered as the part of a page that provides the interesting functionality. The framwork takes care about users/groups/permissions ans all the stuff you need in almost every functionality. So a module programmer can take advantage og the API and just use them in ever module without the need to worry about it every time and time again. Also over time there will be various public modules provided by other programmers who had to face the same problem as you do. So you can just drop the module in and use it without the need to write it from the scratch.

1.2 Roles

Roles are used to assign users to a group with in most cases higher permissions than the default user possesses (eg. admins, etc.). MetaPortal does not use something like numerical userlevels but role *tags* to look if a user has access to special functions. There are some roles that are needed by MetaPortal itself and thus are available right from the very first minute. But they are almost all used to assign administrative tasks concerning MetaPortals settings and maybe not of any need by module programmers. If you're adding modules to MetaPortal you will need to add some new roles to give users access to the functions within your module (eg. a news module might need roles like *editor*, *reporter*, *editor_in_chief*, etc.). So you have to tell MetaPortal in your *roles.conf* file which role tags are needed by your module. As soon as a module is loaded into an organization MetaPortal will take care about installing the roles by making them available in the user/group administraion for assignment. The useradmins or superadmins have to take care then which users/groups should be assigned to which role. In the next section of this manual you can read how you can check if the

current user/session has a special role tag assigned.

1.3 User Attributes

Like within every major application it is not enough to just know a users id or loginname. There are a bunch of other informations that maybe of interest for your module. This can be the users EMail address or the users surname. MetaPortal calls these informations attributes. There is an infinite mass of different attributes that might be needed. Every single module might need an attribute that is only needed by itself and no other module. But there are some common attribuites that are *shared* amog some modules. So we decided to store all user attributes in a central place so all modules can share them without the need for the module programmer to store them seperated for themselfs. This does not only make it more easy for the programmer but also for the user who then only needs to enter his/her surename, etc. in one place and not inside every module again. To make it as easy as possible for the module programmer to access these attributes there is only one API call to get the needed attribute value for a specific user. They are accessible via attribute tags. (NOTE: there is no system implementet for the module programmer to specify wich attribute tags are required, optional or used at all. This will follow soon!)

2 Directory Structure

The name of your module is also the name of the directory inside the modules directory. You should prefix the modules name with you initials or some random letters of you choice. You don't want to create a module called *news* or *poll* because a module name has to be unique inside of a Metaportal installation.

2.1 General

To provide a valid module which is loadable into MetaPortal you have to take care about the directory structure and some files inside your modules directory which are parsed and checked for valid syntax. If the directorys ore files are not in the described format you won't be able to load it into MetaPortal.

2.2 root

The root of your modules directory should contain only a few files and the directorys describes below. Of course you can place as many files here as you want (eg. license files...) but you need at least the ones describes in the *Files* section below.

2.3 share/

This folder is somehow special and only a few module will need it. It can contain things that need to be directly accessible and are not dependent by the theme or container a module runs in. It will be symlinked into every container of every organization wich uses the module. An example on what might be in there are user uploaded content for eg. a gallery. Use it with care and don't confuse it with the *graphics/* dir which DOES depend and thus cachanges togeth with themes!

2.4 themes/

Everything the depends on a theme is located inside this directory. It contains one dir for every theme that is explicitly supported by the module. It the framework is using a theme that is not supported by the module it will use the default theme defined in your *module.conf* file (described below).

2.4.1 themes/<*THEMENAME*>/

Inside a directory named after a theme(eg. *metaportal* or *coldice*) resides data with may change depending on the used theme. This includes of course the templates files and the language definitions for the templates. Also the

graphics dir wich contains all the funky buttons and images that are part of a theme.

2.4.2 themes/<*THEMENAME*>/graphics/

The directory contains all the images, buttons,etc, your module is using during runtime. This includes everything that needs to be accessed directly. Images are a good example of something that can not be included into a template but have to be in a directory that is accessible by the clients browser. When a module is activated within an organization the framework takes care about symlinking the content of this folder into the apropriate directories in all the containers used by the organization. So you don't have to care about placing the links to your graphics by yourself.

2.4.3 themes/<*THEMENAME*>/lang/

Here you can store the language files to support different languages inside your module. The format of them is Smarty Config File style. The naming convention for them is [*LANGUAGE*].conf(eg. en.conf or de.conf).

2.5 includes/

Here you should place your sourcecode!

2.6 sql/

Here you should place .sql files wich are required to initialize the tables your module requires to run.

2.7 libs/

In this directory you should place all class files. If other modules want to use your classes and they specify this in their *module.conf* file, every file in your libs/ dir will be included when the other module is loaded. So there should be no self running code but classes and plain functions you want to share with other module coders. If you have sourcecode that produces output or classes you don't want to share for some reasons place them into the *includes/* directory. As soon as your libs are loaded into the environment of a different module, thelibs of your module listed in the *deps* line of your config file will also be loaded to satisfy your own requirements. The framework also takes care about dependency loops and cross dependecys.

3 Files

MetaPortal uses some files inside a modules root directory to interact with the frameworke itself. In this chapter the mandatory and optional files are explained by describing their functionality, content and syntax.

3.1 module.php

This is your main sourcecode file. It is included by the framework if you module is called and should contain the very basic runtime code for your module.

3.2 module.conf

In this file some variables are defined to provide the framwork with a few basic informations about how to deal with the module and its themes/languages.

3.2.1 mod_version

Just your module version. (e.g. mod_version="1.0") Currently not used by the framework.

3.2.2 default_lang

Defines a default language with should allways work. (e.g. default_lang="de") Is used if the language requested by the user is not in your *languages* list. Has to be in 2 char ISO notation (e.g. de or en).

3.2.3 languages

A comma seperated list of all languages supported by your module. The has to be in 2 letters ISO. (e.g. languages ="de, en, no")

3.2.4 default_theme

Choose a theme wich should be used as default if the desired theme is not available in your module and thus not present in your themes/ directory. (e.g. default_theme="metaportal")

3.2.5 deps

This is a comma seperated list of module names. All files from the modules *libs/* directorys will be included. So if you need classes from other modules, place the name of the module here and all dependencies will be (recoursively) included. (see about description of the *libs/* directory for details) (e.g. deps="zx_foo, xy_bar")

3.3 roles.conf

In this file you should list all roles needed by your module. The format is like a regualar config file with the role tag as the variable name and a human readable description as its content with one role per line (eg. ADMIN=The Admin Role).

4 Framework Objects and Classes

The framework itself provides you with instanciated objects wich provide you most of the basic features you need to rapidly develop and deploy a new module.

4.1 $GLOBALS['mp_log']

This object is an global instance of the *Log* class. The mp_log objects provides a convenient way to log all the stuff you want to to without the need for you to take care about the loggin media. We now describe how to use the essential methods of the object.

4.1.1 $GLOBALS['mp_log']→log(*$level*, *$source*, *$text*)

This is the primary interface method for the log object. It will print a log sting to the specified log facility. At the moment this could be a plain textfile, syslog or a mysql table. The first argument ist the log level. MetaPortal uses the same log levels as a normal syslog and provides symbolic constants for the levels which are explained later in this guide. This provides you with the capability to decide if a log entry is made only for debugging purposes or for a mission critical condition. As the second argument you should provide the source of the log entry. In most cases this is something like php_filename/classname/methodname to identify where the log entry originated from. The third argument is the plain text you want to write into the log file.

Sample:

```
$GLOBALS['mp_log']->log(MP_LOG_CRIT, 'MPUser::createNew()', 'Unable to create new user');
```

4.2 $GLOBALS['mp_sql']

This global SQL object is used to access the administrative tables of MetaPortal. If your tables reside in the same databes you should think about using it to process your queries. The object takes care of load balancing the queries if there are more databases to be read from. If you run a query with this object it also keeps internal statistics and does basic profiling of your querys. Currently it only counting all qerys issued with seperated read-only and read-write querys and it traks how many unique querystrings you issued. So you maybe want to look if you have redundant querys and tune them with a little bit more intelligence insight your classes to avoid issueing the same query more than once if it could be avoided with eg. caching.

4.2.1 $GLOBALS['mp_sql']→readQuery(*$query*, *$retval*)

This method should be used to issue read_onyl querys. The first argument is the query, the second is passed by reference and contains the result. It is not a regular mysql_result but a *DB_Result* set from the pear databased framework. (see http://pear.php.net for further informations on how to deal with this kind of result). If the call fails for some critical reason(eg. if the tables/fields do not exist) the application dies and produces a log entry as well as debug output on screen. The default fetchmode is set to be an associative array and if you want to change it please refer to the pear dokumentation.

Sample:

```
$sql = 'SELECT `login` FROM `mp_users`';
$GLOBALS['mp_sql']->readQuery($sql, $result);
while($row = $result->fetchRow()) {
   echo 'Login: '.$row['login'].'<br/>';
}
```

4.2.2 $GLOBALS['mp_sql']→writeQuery(*$query*)

The method for queries who do writing operations on the database. The only argument to it is your query string. It returns *true* if the query was issued without mayor problems and dies if the query goes wrong. (In most cases you can also issue read querys through this link but you realy shouldn't!) Like the read query the write query lets the whole applikations die on a critical error and produces an entry in the error log and on screen. The default fetchmode is set to be an assocciative array and if you want to change it please refer to the pear dokumentation.

Sample:

```
$sql = 'UPDATE `mp_users` SET `login` = 'joe' WHER `id` = '3' LIMIT 1';
$GLOBALS['mp_sql']->writeQuery($sql);
```

4.2.3 $GLOBALS['mp_sql']→countQuery(*$query*, *$retval*)

This call is only for *COUNT()* queries. The first argument is the query, the second one is your result. the result should in most cases be a interger $\geq$ 0. If a cirical failure happens the whole application dies like the read and write queries.

Sample:

```
$sql = 'SELECT COUNT(`id`) FROM `mp_users` WHERE `login` LIKE 'a%'';
$GLOBALS['mp_sql']->countQuery($sql, $result);
echo 'Logins starting with the letter 'a': '.$result.'<br/>';
```

4.3 $GLOBALS['mp_session']

This is a very important object, because it provides all kinds of information related to the current session as organisation id, user id or the current session state.

4.3.1 $GLOBALS['mp_session']→getOrgId()

Returns the Organization ID of the current active organization. Should always be an integer.

Sample:

```
$orgId = $GLOBALS['mp_session']->getOrgId();
echo 'Current Organization ID is: '.$orgId;
```

4.3.2 $GLOBALS['mp_session']→getUserId()

Returns the User ID if the session is authenticated and the user is logged in. If not it returns false. This is the case when a session is in the states *unauthed* or *authing* cuz at this time the user is not logged in.

Sample:

```
$userId = $GLOBALS['mp_session']->getUserId();
if($userId == false) {
    echo 'No User logged in';
} else {
    echo 'Current UserId is: '.$userId;
}
```

4.3.3 $GLOBALS['mp_session']→getStatus()

Returns a string with the current session state. The value should be *authed*, *unauthed* or *authing*. If it returns *false* there is something very wrong with the framework itself.

Sample:

```
$sessStatus = $GLOBALS['mp_session']->getStatus();
switch($sessStatus) {
    case 'authed':
        echo 'Session is authed!';
        break;
    case 'unauthed':
        echo 'Session is unauthed!';
        break;
    case 'authing':
        echo 'Session is currently authing';
        break;
    default:
        echo 'Something fishy is going on here!';
}
```

4.3.4 $GLOBALS['mp_session']→hasRoleTag(*$tags*, [*$modname*])

This method returns *true* if the current session has access to the role tag or at least on of the roletags specified in the first argument. If not *false* is returned. The first argument may be a singe roletag ar an array of roletags. The second argument is optional. If you leave it empty the roletags are assumed to be existend in the current selected module context. With providing the name of a different module as the second argument you have to oppotunity to check roles for different modules other than the current active one. The second argument is mostly used by plugins who don't have own roletags but it also works inside of modules.

Sample:

```
if( $GLOBALS['mp_session']->hasRoleTag(array('ADMIN'), 'usr_admin') ) {
    echo 'Current user has roletag 'ADMIN' for module 'usr_admin';
} else {
    echo 'Current user does not have roletag 'ADMIN' for module 'usr_admin';
}
```

4.3.5 $GLOBALS['mp_session']→getAction()

This method returns the last value of a post or get for *data[action]*. At the moment it is persisten to the session until 1. the module is changed 2. another *data[action]* is posted or getted. But this maybe subject to changed in the neaf futue. So depend to much on its behavior...

Sample:

```
$action = $GLOBALS['mp_session']->getAction();
echo 'Last value of data[action] since last change is: '.$action;
```

4.4 $GLOBALS['mp_user']→getAttribute(*$tag*)

This returns the users values saved for the specified tag. If the user hasn't entered a value for the tag yet but it is present inside the organisation the organisations default value for the tag is returned. If the requested tag is not present at all in the organisation *false* is returned.

Sample:

```
$email = $GLOBALS['mp_session']->getAttribute('EMAIL');
echo 'The current users EMail address is: '.$email;
```

4.5 $this

Your module.php is included into a *Module*-object. So you can use all methods of it with just using the *$this* object.

4.5.1 $this→getIncPath()

This functions returns the path to your *includes/*-folder inside your module dir WITH a trailing slash. So use it if you want to include more files into your module which reside inside the *includes/*-dir.

Sample:

```
$incPath = $this->getIncPath();
require($incPath.'my_include.inc.php');
```

4.5.2 $this→getGfxUrl()

This method returns a path to your graphics directory relative to your webservers root. You can use this if you want to generate links to graphics from within your sourcecode. Due to the fact that the framworks generates dynamic containers als thus compleately different URLs to the graphics are needed to access them you can not hardcode any paths in your code. The framework also takes care about modifiying the paths in accodance to the theme used with placing symlinks to the right placed in you module dir.

Sample:

```
$gfxUrl = $this->getGfxUrl();
$content = '<img scr='.$gfxUrl.'my_image.jpg' />';
```

4.5.3 $this→getGfxPath()

This return the absolute path to your graphics dir on the machine. The path returned is also dependedn on the current selected theme!

Sample:

```
$gfxPath = $this->getGfxPath();
unlink($gfxPath.'my_image.jpg');
```

4.5.4 $this→getShareUrl()

Like the *getGfxUrl()* method it returns an URL path but this time to the share director.

4.5.5 $this→getSharePath()

Like the *getGfxPath()* method it returns the absolute filesystempath to the *share/* directory for your module.

4.5.6 $this→getLang()

Returns the lang used for the module. This might be one of the values defined in the *module.conf* file in the *languages* variable. Useful to decide inside of php code which language to use if you have dynamic but language

depended sourcecode which you can't place directly inside the templates. Language values allways have to be formatted in 2 character ISO codes!

4.6 $smarty

The *$smarty*-object is in local scope to your modules.php file. It is an instance of the *ModulesSmarty*-class, an extension to the regular *Smarty*-class from the smarty temaplte engine. You can access all the methods provided by the engine like *assign*, etc. See http://smarty.php.net for further documentaion.

4.7 Misc class

Inside the *Misc* class you can find some useful functions for your module which are not part of any other class. All of them are public which means you don't have to instanciate the class. You can and should call every function with the the syntax *Misc::function()*.

4.7.1 Misc::varEval(*$variable*, *$pattern*)

This is a function to evaluate a variable against a regular expression. First argument is the variable and the second is the regex pattern to evaluate against. Its a nice thing because it takes care about logging every evaluation and also if an evauation fails. If the patterns match the function returns *true* else *false*. MetaPortal provides you with a couple of predefined symbolic constants for variable evaluation. You should use them where it makes sense because it'll make your source more readable and if they are updated you don't need to update your sourcecode. The constants are describes later on in this document.

Sample:

```
if( Misc:varEval($userId, MP_VAR_INTEGER) === false) {
    echo 'The userId has to be an integer!';
} else {
    echo 'OK, userId seeme to be an integer.';
}
```

5 Variables

There are some special variables which provide you with content or just should be avoided because they carry critical information through the MetPortal framework.

5.1 MetaPortal

Variables described here have special meanings an thus should not be used unless you intend to change them for some good reason. They are have valid scope inside your php code.

5.1.1 $data[module]

If you want to switch to another module you can *post* or *get* the $data[module] variable and fill it with the name of the module you want to load. If the users have not the required permissons for the module, a message will be displayed.

Sample:

```
<a href='{$PHP_SELF}?data[module]=my_module'>Load MyModule</a>
```

5.2 Smarty

Here are some variables wich are present within your smarty templates for your convenience. Use them but don't change them unles you know what you are doing!

5.2.1 {$PHP_SELF}

This is pretty much the same like in PHP itself. In fact is a copy of its contents.

Sample:

```
<form action='{$PHP_SELF}' method=post>
[...]
</form>
```

5.2.2 {$GFX_DIR}

Points to the graphics dir URL within your module. It is allways relative to your webroot. So you don't have to worry about the location(s) of your graphics. Just use this dynamic path to them.

Sample:

```
<form action='{$PHP_SELF}' method=post>
[...]
</form>
```

5.2.3 {$LANG}

Usefull for multi language support. Contains one of the languages you have defined in your *modules.conf* file.

Sample:

```
{include file='template_`$LANG`.tpl'}
```

5.2.4 {$SHARE_DIR}

Points to the shared dir URL within your module. It is allways relative to your webroot. So you don't have to worry about the location(s) of your shared files with a container. Just use this dynamic path to them.

Sample:

```
<script src="{$SHARE_DIR}/javascriptstuff.js" type="text/javascript">
```

5.2.5 {$LANG}

Usefull for multi language support. Contains one of the languages you have defined in your *modules.conf* file.

Sample:

```
{include file='template_`$LANG`.tpl'}
```

5.3 Forbidden Variables

Here is a list of Variables wich should not be used by you. Most of them are in global scope and can safely be used in your local module.php scope

5.3.1 mp_*

You should not use the prefix *mp_* for any of your variables because it is used by metaportal itself!

6 JavaScript

There are a few very minimalistic js functions available. They provide some functionality that may be helpfull in daily use in your templates. All provided functions and variables are prefixed with *mp*.

6.1 Functions

Functions provided by the MetaPortal theme.

6.1.1 mpAskLink(*link*, *question*)

This is just a little protective helper if you have critical links and you don't want to write an extra page like "Are you sure you want to click this Link?" for it. It pops up a confirmation question and only if the user then presses the 'OK' button the link will be followed. Very usefull for "delete" links or stuff like that. First argument is the link you want to follow if user clicks 'OK'. Seconf argument is the question you want to ask the user.

Sample:

```
<a href='javascript: mpAskLink('{$PHP_SELF}?data[module]=mp_test', 'Realy?');'>delete</a>
```

6.1.2 mpProtectSubmit(*button*, [*caption*])

This function intends to prevent dubble submission of forms that occure when a user dubbleclicks on form submit buttons. This can be very annoying. So what the function does is it replaces itself with the *mpBlockSubmit* function and disables the button. So every browser should at least understand one of the two methods of disableing the button. You just have to add the function to the *onClick* event of you submit button to make it work. The first argument is the button object you want to protect (usualy if placed inside the onClick this just the *this* object). Second argument is optional and may contain the value (caption) that is placed on the button after the click (e.g. 'please wait' or 'processing...').

Sample:

```
<input type='submit' onClick='return mpProtectSubmit(this, 'please wait...')' />
```

6.1.3 mpBlockSubmit

This function should not be used anywhere. Its only purpose is to replace *mpProtectSubmit* as soon as it is called to produce an error if it is called again.

6.2 Variables

Variables provided by the MetaPortal theme.

6.2.1 mpFormErrorMsg

This is the default errormessage that will be outputted if *mpBlockSubmit* is called.

7 Constants

Here are defined constants which can be used if you think they are of any use. If not... well they are defined anyway... so use them or not... but don't try to redefine them!

7.1 Eval constants

Here are a few constants to use with the Misc::varEval() function. They provide you with some default regular expressions so if you want to change them you just need to change the constants.

7.1.1 MP_VAR_USERNAME

Per default a username has to be alphanumeric with at least 2 and a maximum of 25 charakters.

7.1.2 MP_VAR_ORGNAME

Per default an organization name can be alphnumeric plus the . and has to be at least 3 and at maximum 25 charakters long.

7.1.3 MP_VAR_PASSWORD

Per default password can be any character but have to be at least 4 characters in size.

7.1.4 MP_VAR_INTEGER

Just checks if a variable contains only and at leaset one digit.

7.1.5 MP_VAR_MODNAME

A module name can per default only contain alphanumeric characters with at leaset 3 and a maximum of 25 characters.

7.1.6 MP_VAR_ROLETAG

A roletag has to contain just alphanumeric characters. Its size is between 5 and 50 characters long.

7.2 Log constants

These symbolic constants should be used in context with the logging facility of MetaPortal do decide how important an event you want to log is.

7.2.1 MP_LOG_DEBUG

Debug-level message

7.2.2 MP_LOG_INFO

Informational message

7.2.3 MP_LOG_NOTICE

Normal, but significant, condition

7.2.4 MP_LOG_WARNING

Warning conditions

7.2.5 MP_LOG_ERR

Error conditions

7.2.6 MP_LOG_CRIT

Critical conditions

7.2.7 MP_LOG_ALERT

Action must be taken immediately

7.2.8 MP_LOG_EMERG

System is unusable

8 Appendix

8.1 Sample module.php

```
<?php
/**
 * sample module
 *
 * little module that does nothing but printing out a few things
 *
 * @author Thorsten Kunz
 * @copyright Copyright &copy; 2003, Thorsten Kunz
 * @license http://www.bytecrash.net/metaportal/LICENSE BSD style license
 * @package some_package
 * @subpackage some_subpackage
 * @version v0.1
 */

//put this protective code sniplet at the begining of EVERY
//.php file you have in your module! It prevents the possibility
//to include the files directly wich might have a serious security
//impact in some cases!
if( !isset($GLOBALS['mp_config']['caller_name']) ||
    !eregi($GLOBALS['mp_config']['caller_name'], $_SERVER['PHP_SELF']) ) {

    die(``This file may not be accessed directly!'');
}

//at this point you should check things like userpermissions and in case
//your module should only be available to registered userd do a check
//if the session is authed here, too!
if($GLOBALS['mp_session']->getStatus() != 'authed') die(``holy shit!'');

//get the path for the include directory. not needed but makes the code
//a little more readable if you have many includes
$myIncludePath = $this->getIncPath();

//switch for the last action submitted via data[action]
switch($GLOBALS['mp_session']->getAction()) {
    case 'hello':
        print 'Hello World!<br/>This is sample module speaking...';
        break;
    case 'complex_hello':
        //require a file that has some functions and code in it
        require($myIncludePath.'file_with_complex_hello_code_from_include_dir.inc.php');

        //now display a smarty template to generate output from
        $smarty->display('template_file_to_display_complex_hello_code.tpl');
        break;
    default:
        print 'Well, you did not specify a valid action... but hello anyway!';
}

?>
```