Admin interface

Create Widget

The widget creations have following conditions:

- Widgets code should be placed in the directory name 'widgets', where the name of sub-folder is same as the widget's name: ('widgets/[widget name]')
- place a [widget name].php file within the widget's path
- The widget must contains a .md file, through which the Mapi framework recognized. Eg: widgets/[widget name]/[widget name].md. This file should contain a title, version number, and a brief description of the widget.

```
#Menu
##0.0.6
Display menu items
```

If they are all available, the Mapi framework will recognize the widget and will appear in the admin panel. Here you can turn on and off, and install / uninstall if required.

Create popup popup window for the markers on the map

The contents of the popup window code added by javascript. The #mappiamo realize this with OpenStreetMap and javascript library located on http://leafletjs.com which implements and available on the path 'assets\js\leaflet\'.

Steps of widget creation

The content of [widget name].php

Within the [widget neve].php-ben have to be include function names "mwidget_[widget name]()". This is the first function, what read and run by the Mapi framework. This is the reason, why the function name bounded. The widget start with "mwidget_" and the widget name: "mwidget_[name]". The other classes or functions names are no matter.

Because this function will be read first by the widget, these rows will be executed first.

Eg:

```
function mwidget_mappiwidget() {
      echo '<h1>My cool widget</h1>';

      if ( 'view1' == $view ) {
            incude( 'includes/view1.php' );
```

```
        } elseif ( 'view2' == $view ) {
            include( 'includes/view2.php' );
        } else {
            include( 'includes/default.php' );
        }
}
```

HTML initialization code not required to the widget `(<head>, <body> …)` because this code read from the template.

To display the widget user interface, it must be register in the template:

Eg.: templates/template_name/home.php -> row 19:
`(<?php $this->widget( 'mapiwidget', array() ); ?>)`

The result: http://mappiamo.org/template_name/index.php
echo: "Hello a MapiWidget "

The template necessary to register the widget with the required parameters as follows:

```
<?php $this->widget( 'mapiwidget', array( 2, 3 ) ); ?>
```

The widget, which reads above two parameters, like this:

```
function mwidget_mapiwidget( $a, $b ) {
      echo $a; // -> 2
      echo $b; // -> 3
}
```

The method in the template for widget registration:

```
<?php $this->widget( 'mapiwidget', array( 2, 3 ) ); ?>
```

"$this->widget()" : global function call the widget
“mappiwidget” : the name of widget
“array(2, 3)” : send parameters from template to the widget’s function

---

Reading content from database

The framework read the content with this method:
http://www.mappiamo.org//index.php?module=[module name]&object=[content id]
so:
http://www.mappiamo.org//index.php?module=**content**&object=252
where the value of module called **'content'** is on the path '/modules/**content**/', reads row from the database where the id value is 252. The database table called 'Content' contains 21 columns, the content stored on column called 'text'.

The default content display by a module called 'home', which refer to a database table called 'content' ’s default row. The default module (called home) is selected from the database

table called 'modules'. It should be set to 1 the line containing the name of the module ('name' column) 'default_module' column.

So:
http://mappiamo.org/index.php?**module**=**home**
doing same as like this:
http://mappiamo.org/
because the default setting in the table '**module**' is the row where the module name is '**home**'.

SEO link by .htaccess URL rewrite method

If .htaccess rewite used on the server root, the link to access content can be possible with SEO friendly URL. The URL syntax is:
http://www.mappiamo.org/[controller name]/[content ID]/[parameters]

For example:
If the original URL is: http://www.mappiamo.org/index.php?module=content&object=931
The rewrited SEO friendly url: http://www.mappiamo.org/content/931

where the 'content' is the module (controller) name, the '931' is the ID of the content in the database table 'contents' column name 'id'.

Same method can be user to read category:
http://www.mappiamo.org/category/1
where the number '1' after 'category' is the category ID in the database table 'category'.

The rewrited and the regular URL can be used both to access content. If SEO friendly URL required within full site, this method have to be used on the MVC View (HTML) code.

Creating cutom modules

The custom modules displays a kind content that allow different displays as the default. These include for example different information, contact us. If you want to do this, you will need to create custom modules.
The path for these custom modules is the subdirectories of 'modules' directory. For example the path for the custom content is: 'modules/custom_content'.

Modules may be a bit more complicated than the widget, but greater benefits than the widget.

The developed module has to be made in the MVC system. Subdirectories under the MVC system should be set up accordingly:
modules/content/models
modules/content/view

The Controller of the MVC modules in the root directory of the module, the module has the same name as the controller:
modules/content/content.php

The models are in the folder ‘models’, the folder ‘view’ contains the MVC view, which defines the module display.

The MVC controller, which has the same name as the root directory of the module, the MVC controller file calls the view and the model functions as follows:
$This-> model("name_model", $parameters) -> call model from the controller
$This-> view("name_view", $data) -> call view from the controller

The functional modules available later via URL links. For example, if the name of the module 'accessbility', the following links to be called:
http://www.mappiamo.org/?Module=accessbility&parameter=value

The module content can be accessed by friendly URL if required:
http://www.mappiamo.org/[module_name]/[parameter_value]
http://www.mappiamo.org/accessbility/parametervalue

The other possibility is that the module will decide what will appear on the site.

If required an administrative interface for the module, then it must be created as a special module of the admin interface:

Example: ‘manager/modules/mcontent’

The HTML input fields can be inserted to the Mapiano content.
For example, a checkbox looks like this code in the view:

```
<input type="checkbox" name="mmap_category[]" value="4" />
```

Important data the name of input field 'mmap_category[]' and in this case, '4' is identified the category. The ‘[]’ required to the end of filed name if the input fields getting back arrays like checkboxes multiple selected lists.

In case you want to insert custom content on the map, you need a JS function, on the path assets/js/asset.map.js.
The online manual: http://leafletjs.com/

Example: (assets/js/asset.map.js) function:

```
this.add_marker = function ([latitude], [longitude], [category]) {

}
```

– displays the content of the marker within a popup window.
After that, you can add your own custom functions - if you need new content on the map markers - not only in the mapiamo category.

Because the assets starts earlier than the widgets, you get access to your own widget through this JS function: assets/js/asset.map.js
See, for example: widgets/folder/map.php

Meta-data inputs on content creation and modification process

The meta-data can be saved on the admin interface under the content menu. Meta data have two inputs fields, called 'input' and 'name'. The data stored on the 'content_meta' table on the database. This table have 4 columns, the 'input' and 'name' for admin data, 'id' is the auto increased identification number, and 'external_id' what is the row 'id' on the content table for database join. One more column required in the future for the property name.

The SQL to display content titles and metadata at one table using table contents and content_meta:

SELECT contents.**title**, content_meta.**name**, content_meta.**value** FROM contents INNER JOIN content_meta ON contents.**id** = content_meta.**external_id**

The modification request about medatada inputs

The metadata 'name' field must be selected from a list in the future, and can be input new value what not in the list. The list must be contains (and offer) the previously stored meta names, and meta names from http://schema.org. In this site, look 'type and property' on the link http://schema.org/docs/gs.html#schemaorg_types. Here is the paragraph where the meta 'names' can be collected. For example, the meta names for products:

http://schema.org/Product

- aggregateRating
- audience
- brand
- color
- …..
- the content of Property column.

The another request, that the 'name' field list must be contains the previously stored custom meta names read from the database 'content_meta' table 'name' column.

The solution process

The meta input fields on the admin interface is the part of module ‘content’. This is the reason why the path for this module is ‘manager/modules/mcontent’.

The MVC view for the meta inputs is on the path manager/modules/mcontent/views/edit.php on row 252. For this view required ajax solution for getting required contents for name input.

The meta name data structure have main property. For example: ‘event’, ‘organization’, ‘highway’ etc. The meta name is the list of possible values of property. On the admin interface, new list required for the possible main properties, and the name field can offer the valid values of selected main property. For this process, the ‘content_meta’ table on the database have to be modified and new column have to be inserted to store main property names. When the property name selected, the ajax search list of ‘name’ filed will be filtered by this selection. The property and name selection based on the ‘content_meta’ database table content.

The another requirement for the ‘property’ and ‘name’ fields have to be offered the standard values of ‘property’ and ‘property values’ based on the required data structure, mut these data maybe not stored yet on the CMS database (because no content yet). This data structure depend on the site content. For searchable list creation for ‘property’ and ‘name’ lists, JSON, XML, or database content required filled with the relevant data structure and values.

HTML Input field validations

Input fields (especially text fields) have to be validated before the input data stored on the database. The validation rules have to be stored on the lib/mlib.validate.php as static function. This function contains regex as rule or any expressions as rule. This static function have to be registered on the bin/classes/abs.class.record.php on the ‘setup_object’ function.

Example:
The rule on the file: lib/mlib.validate.php

```
static function email( $value ) {
	if ( ! MValidate::string( $value ) ) return false;
	if ( ! preg_match( '/^([a-z0-9\+_\-]+)(\.[a-z0-9\+_\-]+)*@([a-z0-9\-
]+\.)+[a-z]{2,6}$/ix', $value ) ) return false;
	return true;
}
```

This rule have to be registered on the file: bin/classes/abs.class.record.php

```
if (isset($this->email)) {
	if ( $this->email && MValidate::email( $this->email ) ) $record->email=
$this->email;
	else return mapi_report_message( 'Not a valid e-mail.' );
}
```

This method done server side validation before database insert.

Client side validation before this method would be much faster.
The example of client side validation for test is on the file assets/js/asset.form.js -> 324

The XML importer

The XML import module created for automated data insertions for new content, meta, and categories. This module can be found on the admin interface.

For XML import, two files required: XML for data, and INI for field names.
Some fields required for XML import: Title, Address, ZIP, City
These field names must be defined on the INI file. These rows minimally required for correct import:

category=[the category name]
description=[list of XML field names have to be inserted to the description separated by ‘,’]
Title=[XML field name of title]
Address=[XML field name of address]
ZIP=[XML field name of zip code]
City=[XML field name of city name]

The left side of equation cannot be changed, and this is case sensitive:
category, description, Title, Address, ZIP, City

The right side of equation is the XML filed name, case sensitive.

For example:

If the XML content is:
<**Ragione_Sociale**>VIVAI GARIGLIO SOCIETA' AGRICOLA SEMPLICE</**Ragione_Sociale>**
<**Address**>Borgata Tetti Rolle 4</**Address**>
<**ZIP**>10024</**ZIP**>
<**City**>Moncalieri</**City**>

The ini content must be contains:
Title=**Ragione_Sociale**
Address=**Address**
ZIP=**ZIP**
City=**City**

For the **title** only, possible to use XML @attributes like this:
<ufficio **nomeufficio**="Fiera del Parco di Stupinigi">

For this example xml content, the valid ini file must contains “Title” and XML @attributes name:
Title=**nomeufficio**

User access to the database from the template

If the regular visitor have to read or write the database content, the process can be done from the simple content. When the content page created, Ajax access possible from the front-end to back-end and vice versa.

Creating custom pages for Ajax

1. Create new content using Manager **Contents -> Add new**
2. Use source code mode on the HTML editor
3. Create raw HTML code with all required elements using ID for JS access
4. Check the HTML Purify settings if the HTML code not save to the content database. The config have to be reviewed and modified on the **lib/mlib.purifier.php** on the static function instance(), For example if you need <textarea>, , and <hr> tags for the content, you have to insert this row to the config settings:

```
$config->set( 'HTML.AllowedElements', array('textarea', 'div', 'hr') );
```

When the page created and the HTML code saved, user Page -> Add new on the manager to show menu for the new content.

Create Javascript for Ajax access

Javascript required for the Ajax access. The Javascript have to be saved to the template directory, for example **templates/mappiamo/js/ajax.test.js**. This JS have to be loaded to the template. The best way is the header file: **templates/[Template Name]/head.php**. Check the page source if the required javascript loaded with the custom content.

Use JQuery within this Javascript, and call your PHP with Ajax method. For example:

```
$(document).ready(function() {
	$.ajax({
	   type: 'POST|GET',
	   url: rootpath + 'ajax/AjaxDemo.php',
	   data: { dataname: datavalue },
	   success: function (data) {

	   ……. do something with return data …….

	   });
	});
});
```

Use “rootpath” variable to read access path to the PHP file. This variable can be read within JQuery, or use global variable on the content page and read by PHP.

Create PHP file for server side access

The PHP file for the back-end process have to be accessed from the client side. The best way to save this file to the template path for example: templates/mappiamo/ajax/AjaxDemo.php.

The PHP must be contains inserts for config and database handler:

```
define( 'APATH', dirname( __FILE__ ) );

$settings = APATH . '/../../../settings.php';
$idiorm_lib = APATH . '/../../../lib/idiorm/idiorm.php';
```

If these file paths are correct, use include(**$variable**).

After include, you can access to the global config, and you can setup and use the database handler for read, write, or update databases.

The example files for ajax demo:

- The root: http://test.mappiamo.org/mapi/
- The page: http://test.mappiamo.org/mapi/index.php?module=content&object=653
- The template header: templates/mappiamo/head.php
- The required javascript: templates/mappiamo/js/ajax.test.js
- The PHP: templates/mappiamo/ajax/AjaxDemo.php
- The config for HTML purify: lib/mlib.purifier.php