Gila CMS Documentation

Release 2.0.3

Vasileios Zoumpourlis

Sep 28, 2022

User Manual

Contents:

CHAPTER 1

About

Gila CMS is an open-source and free content management system built with php7. Built with MVC architecture, is very easy to develop on it any customized solution. It is licensed under BSD 3-Clause License. The website is gilacms.com

1.1 Why choose Gila?

Gila CMS is a good option for self-hosted blogs or startup websites

- Installs by default a blogging system.
- No coding skills are required to install or maintain the website
- It is lightweight and fast.
- Easy to build on it.

1.2 Spreading the word!

You can help us spread the word about Gila CMS! We would surely appreciate it!

- Follow our Facebook Page
- Retweet us!
- Give a star on Github

1.3 Where you can get help

- Ask on stackoverflow using the tag **gilacms** (response will be fast)
- Join the Developers Newsletter

CHAPTER 2

Installation

2.1 Preparation

Before beginning with installation make sure that your web host or local server meets these requirements:

- Apache 2/ Nginx server
- MySQL / MariaDB server
- PHP 7.4+ with the following extensions *mysqli, mbstring, mysqlnd, json, gd, xml* and *mod_rewrite* enabled

First unzip gila in a public html folder e.g */var/www/html/gila* and make sure that the folder is writable from the application.

On **nginx** server you will need to configure the redirects in */etc/nginx/sites-enabled/default* (issue #1)

```
location / {
    index index.php index.html index.htm;
    rewrite gila/(?!assets)(?!tmp)(?!robots.txt)(.*)$ /gila/index.php?p=$1 last;
}
```

On **apache 2** server you may need to edit default VirtualHost file in order let .htaccess work. On ubuntu/debian you run

```
sudo nano /etc/apache2/sites-available/000-default.conf
```

And add these lines after *DocumentRoot /var/www/html*

```
<Directory "/var/www/html">
    AllowOverride All
</Directory>
```

If you need to activate mod_rewrite in Linux:

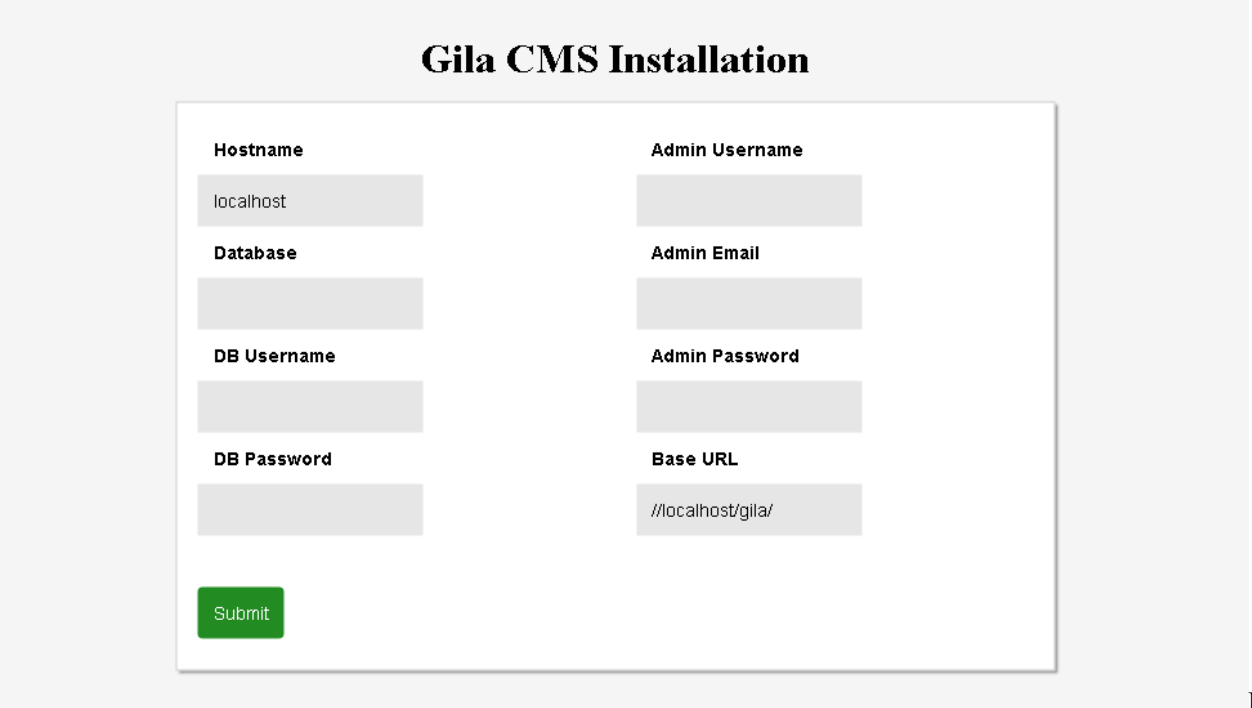
```
sudo a2enmod rewrite
```

Don't forget to restart your server when you make changes.

In order to proceed with the installation, you will need your **database settings**. If you do not know your database settings, please contact your host and ask for them. You will not be able to continue without them. More precisely you need the database hostname, the database name, the database username and password.

2.2 Installer

We access in installation page with the browser e.g *http://localhost/gila/install*

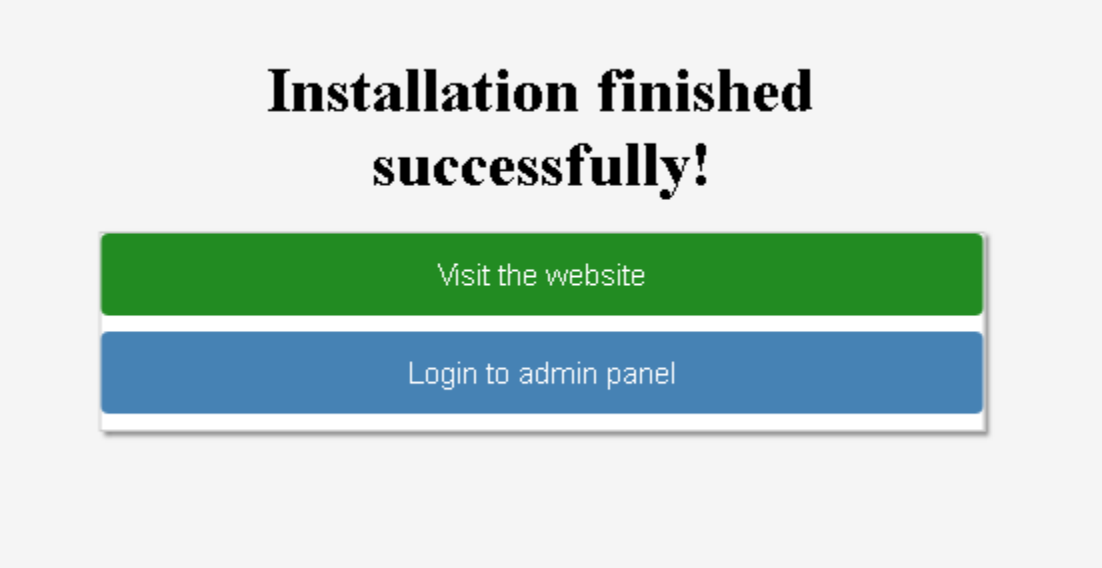


Install

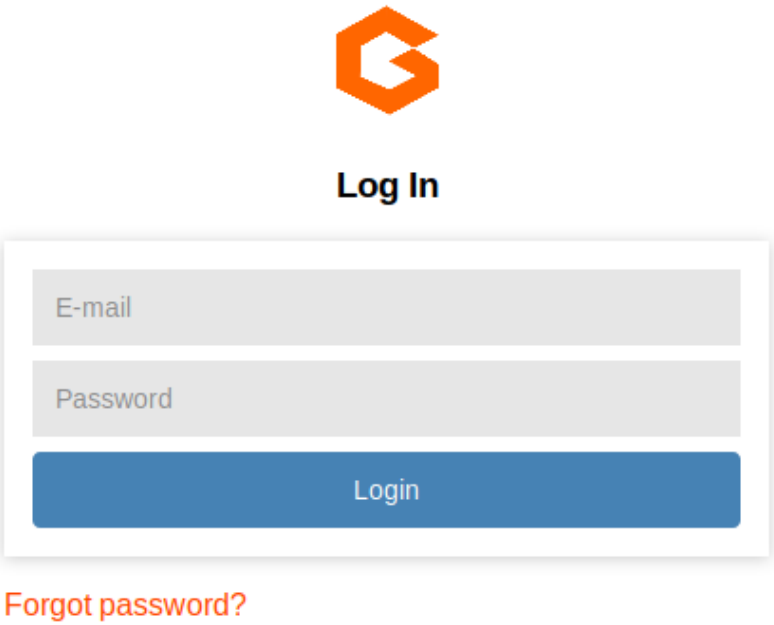
In the installation page we must fill all the fields

- **Hostname:** the hostname of the database, usually it is *localhost*
- **Database:** name of the database
- **DB Username, DB Password:** the username and the password in order to connect to the mysql
- **Admin Username, Admin Email, Admin Password:** a user will be created for the website as administrator with these data
- **Base Url:** the web address of the website e.g. *https://mywebsite.com/*

After filling the data and submit them, we wait a few seconds untill the installation is finished.

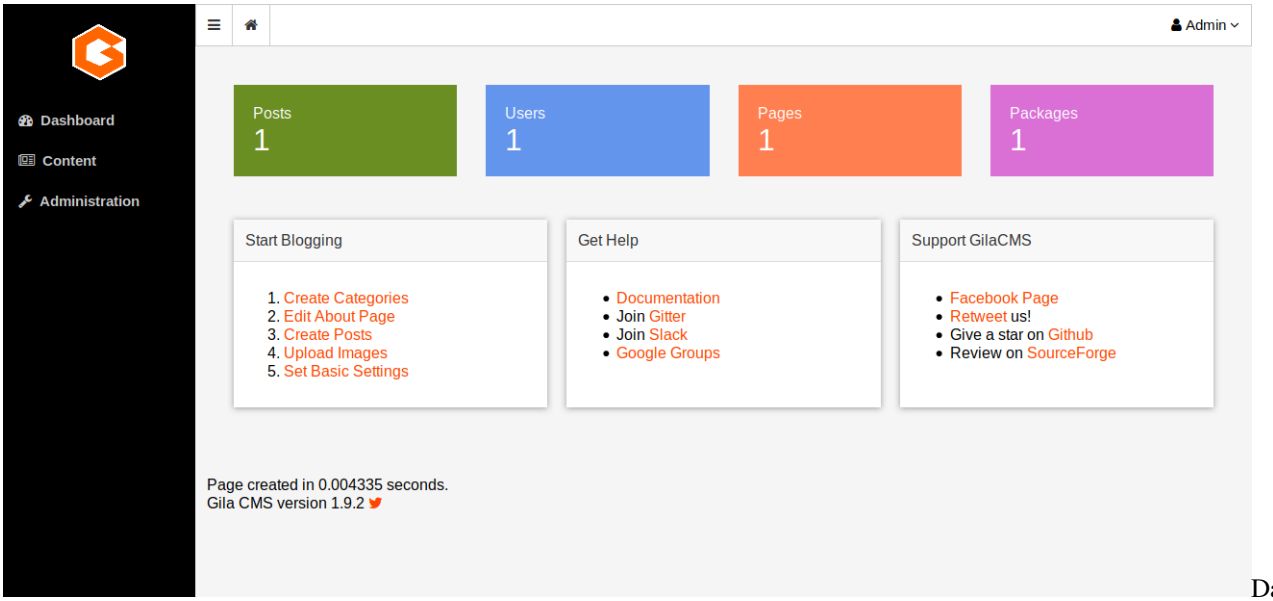


When installation is finished we can enter on the admin panel using the admin email and password that we wrote before.



We can always access in the login page from these links *mysite.com/* **/login** it redirects to the front page of the website *mysite.com/* **/admin** it redirects to the administration

We enter in the administration dashboard.



Dashboard

On the administration menu we see two submenus

- **Content** to add or edit content like pages, posts or files
- **Administration** to edit users or change the settings of the website.

CHAPTER 3

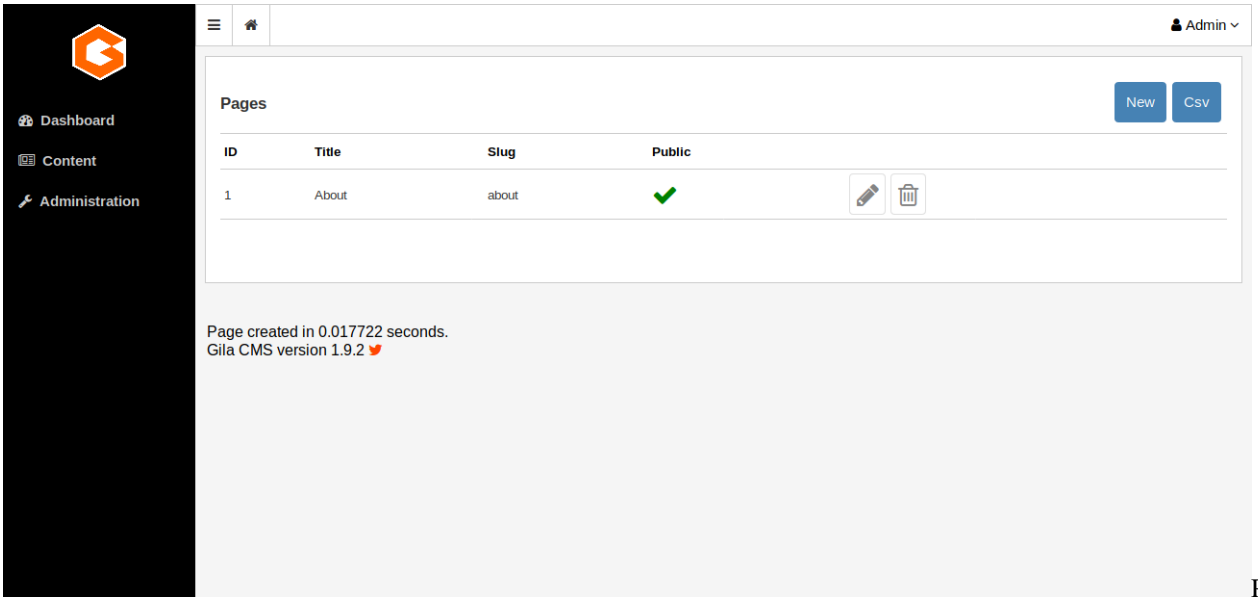
Content

In the administration menu the Content option gives a submenu of the basic content types of Gila:

- ***Pages***
- ***Posts***
- ***Categories***
- ***Media***

3.1 Pages

Pages are the basic content type. A page can be just a text or have media. The information of a page is independent of time so you want them to be found by the visitor in the same place, like on the menu of the website.

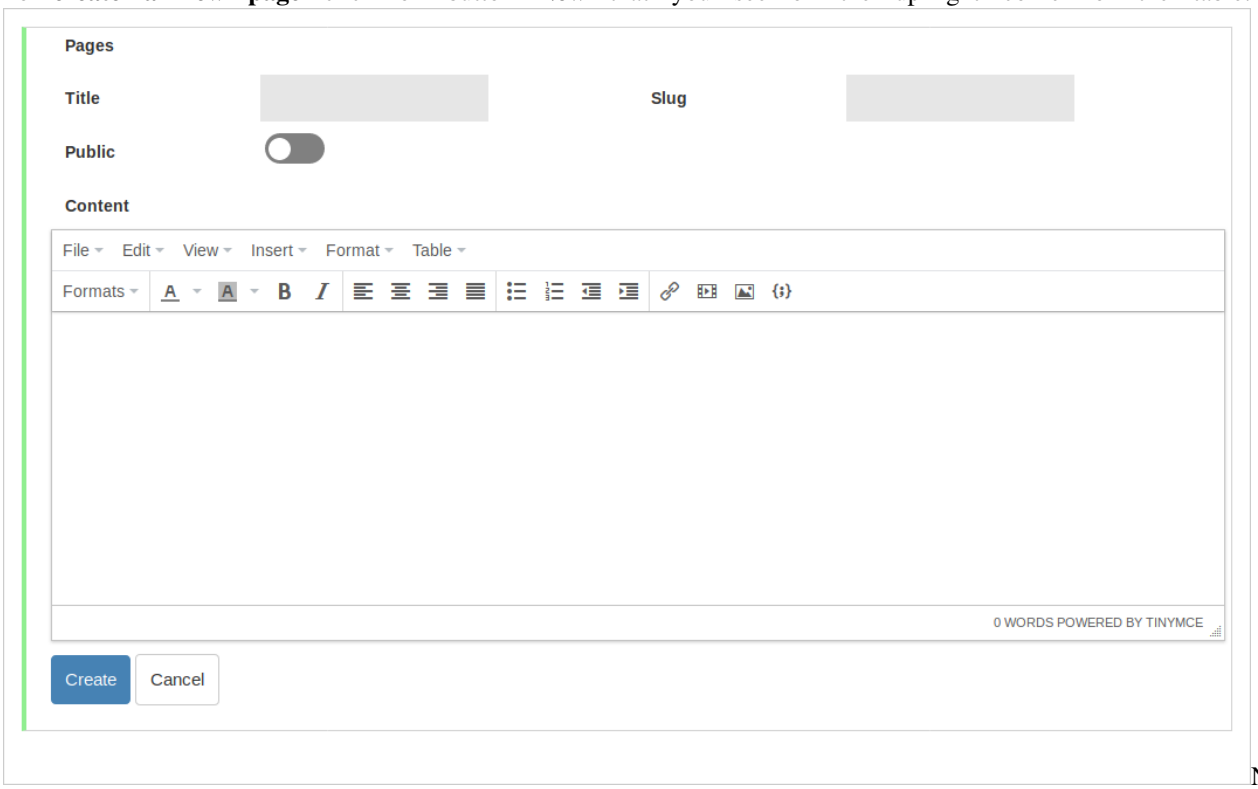


Pages

Every page have four values:

- **ID**: a unique identifier
- **Title**: the title of the page
- **Slug**: is the path of the page. For example the path of a page with title 'My Page' will be *mysite.com/my-page*
- **Public**: an on/off flag. If Public value is off for 'My Page' then *mysite.com/my-page* wont be accessible from the browser.

To **create a new page** click on button **New** that you see on the up-right corner of the table.

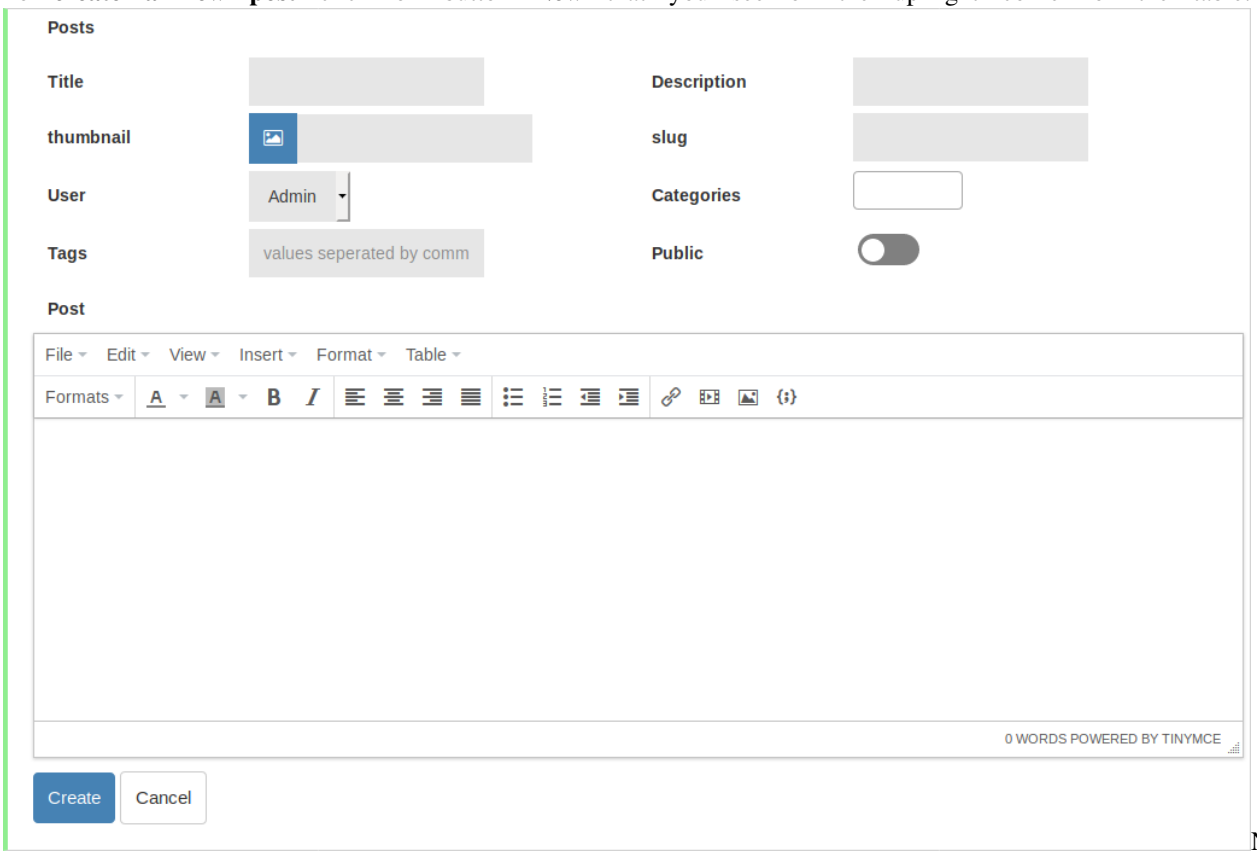


New Page

3.2 Posts

The posts can be news or articles about your business or the interests of the website. They are organized in categories and are listed in chronological order.

To **create a new post** click on button **New** that you see on the up-right corner of the table.



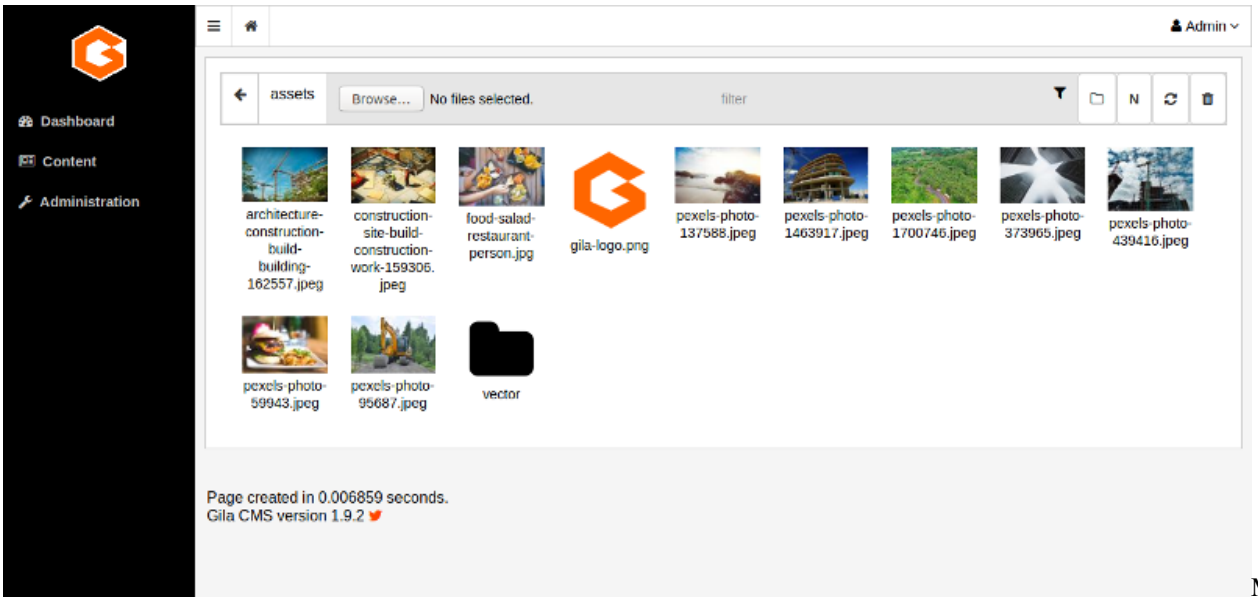
New Post

3.3 Categories

Categories are used to categorize posts or maybe other popular content that you could use later. You only add or edit the names of the categories.

3.4 Media

Media are the images that you want to use for your posts. They are saved as files and not in the database like the other content types. The root directory of media is */assets*. The files and subfolders of */assets* are visible in the public by the path *mysite.com/assets* so you should not upload files or images that you don't want to be found from search engines.


Dashboard
Content
Administration
Admin
assets
Browse...
No files selected.
filter
N
architecture-construction-build-building-162557.jpeg
construction-site-build-construction-work-159306.jpeg
food-salad-restaurant-person.jpg
gila-logo.png
pexels-photo-137588.jpeg
pexels-photo-1463917.jpeg
pexels-photo-1700746.jpeg
pexels-photo-373965.jpeg
pexels-photo-439416.jpeg
pexels-photo-59943.jpeg
pexels-photo-95687.jpeg
vector
Page created in 0.006859 seconds.
Gila CMS version 1.9.2

Media

CHAPTER 4

Administration

In the administration menu you the Administration option gives a sub menu of the basic administration areas

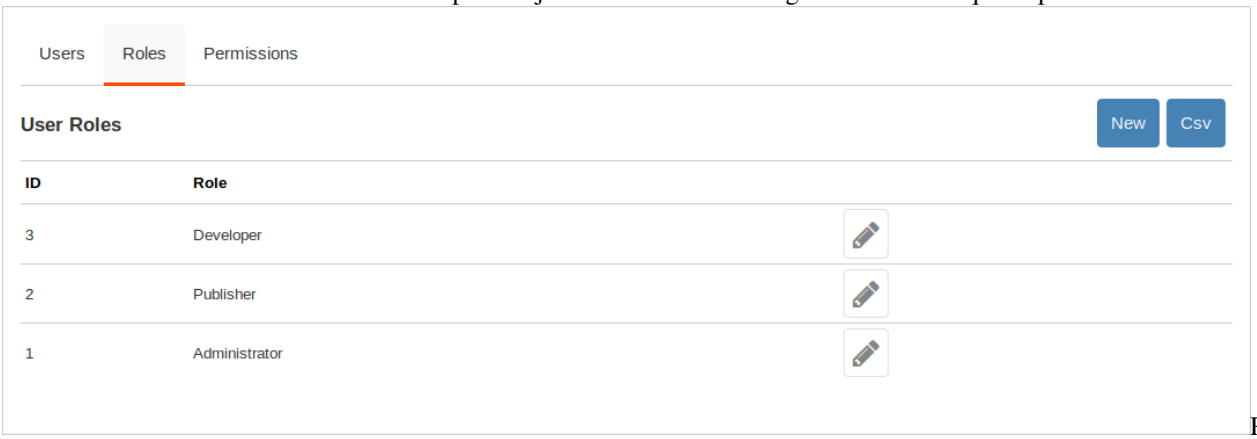
- ***Users***
- ***Main Menu***
- ***Widgets***
- ***Packages***
- ***Themes***
- ***Settings***
- ***Logs***
- ***DB Backups***

4.1 Users

Users are the persons that you can grand access to website and give them privileges to create or edit content. On *Users* tab you can add new or modify existing users.

4.1.1 User Roles

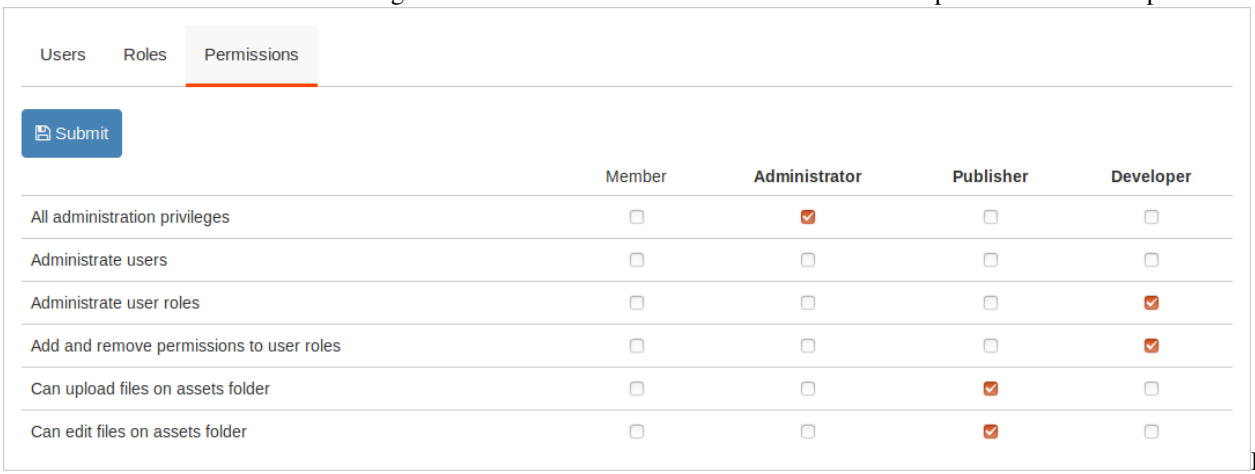
On *Roles* you can add new roles for the users. Roles represent job functions and through them users acquire specific permissions.



Role Permissions

4.1.2 Permissions

On *Permissions* you can set the common permissions that all registered users can have or to specific user roles. Permissions grand to the users access to resources or perform certain operations.



4.2 Main Menu

The main menu is displayed in the navigation bar of the front pages of the website. By default the menu will include a link to the home page, the post categories and the publish pages. In this area you can modify the with the menu editor. This video demonstrates quickly how you can edit the menu:

4.3 Widgets

Widgets are some blocks that you can show them on the layout of the website and improve the user experience of the visitors. Widgets can be for example *menus, comment sections, text blocks, lists of links*. Every page have four values:

- **ID**: a unique identifier

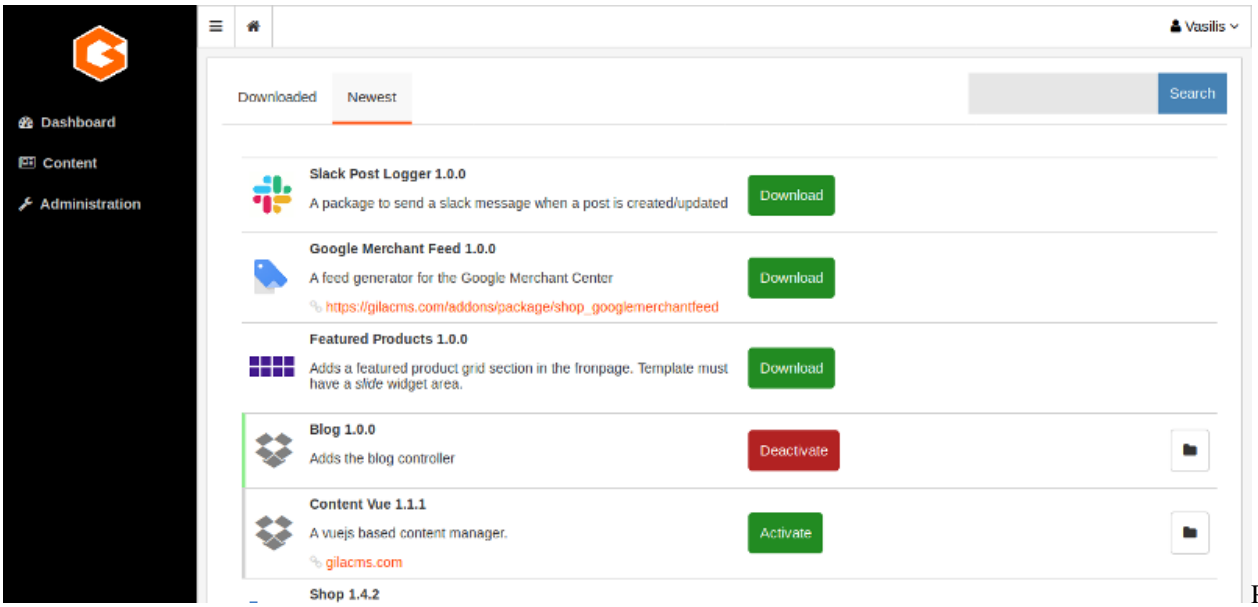
- **Title**: the title of the widget
- **Type**: widget type
- **Widget Area**: is where the widget will be displayed. Can be an area in a view from the website theme or the administration.
- **Position**: the position of the widget in the widget area.
- **Active**: if the widget is visible or not.

Notice: When you create a new widget you can set the type of the widget, this field cannot not be changed later since it determines the widget's structure of the data.

4.4 Packages

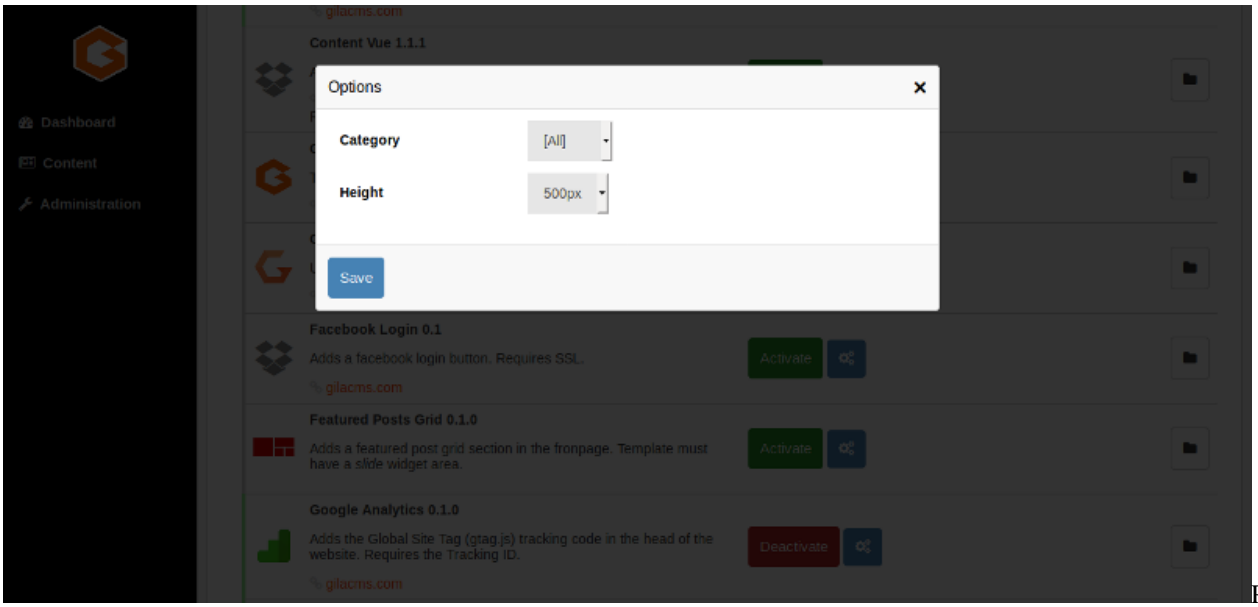
Packages give new functionalities on your web application. They may add a specific widget, a few new links in the administration menu or add new content and new templates to show the content. For example *Facebook Comments Plugin* add a facebook comments section below every page post. *Featured Posts Grid* show the thumbnails photos from featured posts in the front page of a blog theme.

You can administrate packages from Administration->Packages



Packages

The installed packages usually show an **Options** button. By clicking this button you can change some parameters for the specific package. When you save the settings the changes will take effect by reloading the page.

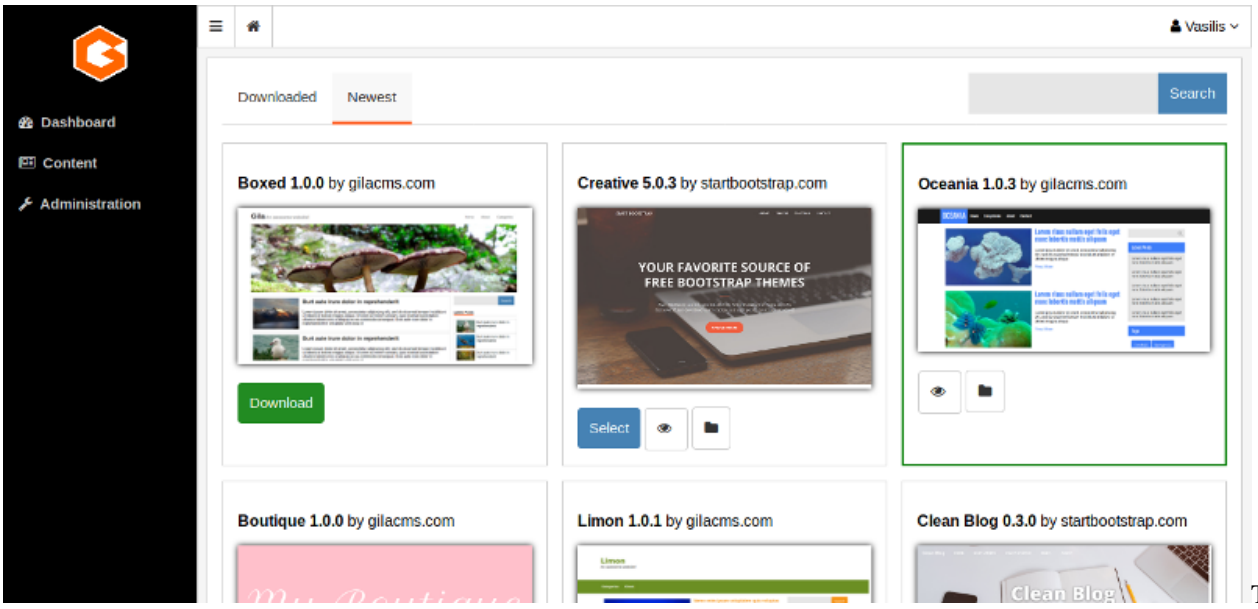


Packages

4.5 Themes

Themes change the look and style of your website. They use different colors and fonts and helps your visitors identify your website and improve their user experience (UX).

You can select the theme from Administration->Themes



Themes

The selected theme usually shows an **Options** button. By clicking this button you can change some options for the theme like the header image (logo) of the website or the main color.

4.6 Settings

On Administration->Settings page and we can make the following configurations

4.6.1 Basic Settings

- **Title** is the website title. It will appear up from the menu if we don't use a logo from the theme options.
- **Description** is a small text that describes the website.
- **Website URL** the url path for exmple 'https://mysite.com'
- **Admin Email** sets the email of the administration.
- **New users can register** adds the registration form for the visitors so they can register as users.
- **New users activation** How the new users are activated.
- **Theme** changes the look and style of your website. You can also change the theme from *Administration->Themes*
- **Timezone** The dates and times saved in posts, logs and the rest of the content will be based on the selected timezone.
- **Language** The language of the website and the administration menu.
- **Admin Logo** set the image to display to administration menu and login screen.
- **Favicon** set the icon for your website.

4.6.2 Advanced Setting

You change these setting if you are developing or set up the installtion.

- **Use CDN** Use CDN for static files of popular libraries (jquery.js, vue.js). It's not advised for local installation where internet connection may fail.
- **Pretty Urls** turns *?c=blog&action=tag&tag=linux* into *blog/tag/linux*. If is not selected by default then your apache server may not have the *mod_rewrite* enabled.
- **Default controller** The controller that will be used if the calling path do not provide it as first parameter. For example the **Admin** controller is used when we call mysite.com/admin but when we call *mysite.com* or *mysite.com/my-post* the default controller will be used, which is **Blog**, so these paths are egual with *mysite.com* and *mysite.com/my-post*. There is not need to change the default controller unless you want to change how the website will be used.
- **Environment** If changes to *Development* the website wont use the combine *load.php* from the packages and will display all notices and errors of the back end. Must use it when you make changes in the code.
- **Check For Updates** will automatically search for new updates on packages and display alerts.
- **Use WEBP** your website will save resized images as webp images, their size is al least 10% small from jpeg or png.
- **Max Media Upload (px)** The maximum width and height of uploaded images (in pixels). If these values are set, then the uploaded images with bigger sizes will be resized. This is a way to avoid excessive use of disk space from photos.

The following settings can be set directly in *config.php* but you be careful, because they could create security issues with your website.

- **trusted_domains** An array of domains that your website will work. For example the main domain, the ip and parked domains.
- **cors** (Cross-Origin Resource Sharing) An array of domains that your website will respond in requests.

4.7 Logs

In this page you can navigate inside the log files of the installation.

4.8 DB Backups

You can create a new database backup and then download it or restore(Load) it later.

4.9 /admin/phpinfo

This option will display the settings of the php moduls on the server. This is for informational purposes only. DO NOT share screenshots in the public of this page as it includes data about the server configuration.

4.10 /admin/content

In this page you can see all the registered tables in your system.

CHAPTER 5

Structure

In the main folder we can see these folders and files.

```
assets/
log/
tmp/
src/
sites/
themes/
index.php
config.php
```

assets/ A public folder that includes the assets from the packages, third party libraries, and media files that we upload.

log/ A private folder that save logs and the user sessions.

tmp/ A public folder with temporally files created.

src/ The folder of installed packages. Here is all the code of the system.

sites/ A folder that we can add more websites that share the same installation.

themes/ The folder of installed themes.

index.php The main index file. For any call, execution starts from here.

config.php The configuration file. It is generated after installation.

CHAPTER 6

Packages

The source code of Gila CMS is split into packages, even the core files are part of the main package called *core*. The package folders are placed inside *src/* folder and desirably have a similar structure:

```
assets/
controllers/
models/
views/
lang/
package.json
load.php
logo.png
update.php
```

The folders are optional but very useful to organize better the code. The only requires files are the **package.json** which has the basic information of the package, and the **load.php**

6.1 package.json

A simple **package.json** file:

```
{
        "name":"Package Name",
        "version":"1.0.0",
        "description":"A short descriptive text of your package for what it does.",
        "url":"package_url.com",
  "author":"Your Name"
}
```

You can also add another index in the object called *options*. It will be an array of objects, the objects are the options to be stored. The index is the option name and it can have optional values with the following indexes:

- **title** the option name to be displayed, if not specified, the index is used

- **type** select | postcategory
- **options** array of {value:display_text}, it is required if is set type:select

```
{
    ...
        "options":{
        "option1":{},
                "category":{
                        "type":"postcategory"
                },
                "lang":{
            "title":"Language",
            "type":"select",
                        "options":{
                                "en":"English","es":"Spanish","el":"Greek"
                        }
                }
        }
}
```

You can get the option values like that:

```
// options are saved using as prefix the package's folder name
// for example if the package has the folder my_package/

$option1 = Config::option("my_package.option1");
$lang = Config::option("my_package.lang","en"); // use default value 'en'
```

More information for **package.json schema**.

6.2 load.php

This file is excecuted in every request to the website, so instead of adding many lines of code we usually register to the system new controllers, new routes or include on more files only when is needed. A simple **load.php** file could be:

```
<?php
// display text below any post
Event::listen('post.after',function(){
        echo 'This is printed after post.';
}
```

IMPORTANT: The first line of the load.php file should include only the opening tag *<?php* and not close with the closing tag.

Some things you can do in a load file:

```
<?php

// add menu item or menu sub item
Config::amenu(['mymenuitem'=>['Item',"myctr",'icon'=>'link']]);
Config::amenu_child('mymenuitem',['Sub Item',"myctr/sub",'icon'=>'link']);

// add an event listener
Event::listen('load', function() {
  // this function will run after all load.php from active packages
```

(continues on next page)

(continued from previous page)

```
  if(Config::hasPrivilege('admin')==false) {
    View::renderFile('landing-page.php', 'mypackage');
    exit;
  }
}

// register new content type
Config::content('mytable', 'mypackage/tables/mytable.php');

// add new column on an existing content type
Config::contentInit('mytable', function(&$table) {
    $table['fields']['newfield'] = [
        'title'=>"New Field",   // the label to display
        'qtype'=>'varchar(80)', // the column type at database schema
    ];
});

// register a controller
// all /blog/* request are processed from class Blog in
// blog/controllers/blogController.php
Router::controller('blog', 'blog/controllers/blogController', 'Blog');

// add a new action for blog controller (/blog/topics)
Router::action('blog','topics',function(){
  View::render('blog-topics.php', 'mypackage');
});
```

6.3 logo.png

An optional small(120x120px) png file

6.4 update.php

This file is excecuted when a package is activated or downloaded (while active). It serves to update the database schema if the package requires it.

6.5 assets/

When package is activated, the files from this folder will be copied in a public folder *assets/{package-name}/*

6.6 views/

The view files are added here. they can be rendered like this:

```
View::render('view-file.php', '[package-name]');
```

CHAPTER 7

Examples

Here are some examples to you get into the of package creation.

7.1 Event: Post Tags

In this example we will list the tags of a post just after it. Create a folder inside *src/* and name it *post-tags*. Inside it create the following files

```
package.json
load.php
logo.png
```

package.json

```
{
        "name":"Post Tags",
        "version":"1.0.0",
        "description":"A package to show tags below the post."
}
```

load.php will run a code when the package is active. We will register a function to run right after the post display.

```
<?php

Event::listen('post.after', function(){
    global $g; // $g will give us the post id
    $tags = core\models\post::meta($g->id,'tag');     // get the tag list of post
    echo "<strong>TAGS:</strong> ";
    foreach ($tags as $tag) echo " <a href='tag/$tag'>#$tag</a>";
});
```

This function will run when the *post.after* event is dispatched. That happens with *Event::fire('post.after');* or *Event::widgetArea('post.after');* from *single-post.php* view file.

logo.png is the package's logo and is displayed in the package list.

Activate the package in */admin/packages*. After that you should see the list of TAGS below any blog post.

7.2 Widget: Twitter Timeline

In this example we will create a widget that displays the last tweets of an account. Instead of using an event to run the code we let the user create instances of the widget choose in which widget area want to display the twitter plugin. Inside *src/* create a folder *twitter-timelines* and add the following files:

```
package.json
load.php
widgets/twitter-timeline/widget.php
widgets/twitter-timeline/twitter-timeline.php
```

package.json:

```
{
        "name":"Twitter Timelines",
        "version":"1.0.0",
        "description":"Installs a widget to display twitter timelines."
}
```

load.php:

```
<?php

// registers the widget name and its path
Config::widgets([
    'twitter-timeline'=>'twitter-timelines/widgets/twitter-timeline'
]);
```

widgets/twitter-timeline/widget.php will include the widget options we want to use. In this case we need the user account and the name to be displayed.

```
<?php

$options=[
    'accountID'=>[
        'title'=>'Twitter Account'
    ]
];
```

widgets/twitter-timeline/twitter-timeline.php is the view file of the widget, it will generate the html code. We use the embedding Twitter content from here

```
<?php
$account = Config::option('twitter-timelines.accountID','gilacms');
?>
<a class="twitter-timeline" data-height="400" href="https://twitter.com/<?=$account?>
↪">Tweets by <?=$account?></a>
<script async src="https://platform.twitter.com/widgets.js" charset="utf-8"></script>
```

Config::option() gets the option of the package that we set up in the package settings. A default value can be used if the option is null.

Activate the package. Now in */admin/widgets* you can create a new widget with type *twitter-timeline* and set the widget area *sidebar* or *dashboard* to see it.

7.3 User: Address

In this example we will add a new field for the users. Instead of adding a new column in the database table we will use the usermeta table to store new values that link to the users. Inside *src/* create a folder *user-address* and add the following files:

```
package.json
load.php
```

package.json:

```
{
        "name":"User Address",
        "version":"1.0.0",
        "description":"Adds a new field for the users."
}
```

load.php:

```
<?php

// make changes to the user content type

Config::contentInit('user', function(&$table){
    $table['fields']['useraddress'] = [
        'title'=>"Address",   //the label
        'type'=>'meta',       //the values of the field will be stored in a meta table
        'input-type'=>'text', //use the text input type
        'edit'=>true,         //is editable from user
        'values'=>1,          //this field gets only one value
        "mt"=>['usermeta', 'user_id', 'value'], //meta table, meta column that links␣
↪to user table,meta column of the value
        'metatype'=>['vartype', 'address'] //meta column of the value type, value type
    ];
});
```

This package will add a new new Address field for users in *Administration->Users*

CHAPTER 8

Themes

Themes change the look and style of your website. You can also change the theme from *Administration->Themes*. They are located in **themes/**.

What themes do

- They override the view files from the packages. View files in packages are placed in a **views/** folder, in themes are placed directly in their root folder. For example when a controller will try to render a view file from the package core it will call **View::renderFile('blog-post.php','core')**. If exists the file **themes//blog-post.php** it will rendered instead of **src/core/views/blog-post**
- They override the response files of the widgets. These files are placed in **widgets/** folder of the theme. For example, **themes//widgets/text.php** will replace **src/core/widgets/text/text.php** when a widget of text will be rendered. This is useful when you want to add styling to widgets similar to the theme. Note that you don't have to create folders for each widget.

Structure

Inside a theme folder we need to have the following files:

```
package.json
screenshot.png
load.php
widgets/
```

- **package.json** has the basic information of the theme and uses a similar schema with the packages.
- **screenshot.png** can be an image of maximum width of 320 pixels. It is used to display the theme on the the themes selection.
- **load.php** registers new variables to the system, like the widget areas that offers, routes for static files or a parent theme.
- **widgets/** folder include the responses of the widgets that you want to override.

8.1 New theme

How to create new theme:

Create a new folder inside **themes/** with name *my-new-theme*. Inside create a new *package.json* file:

```
{
    "name": "My New Theme",
    "version": "1.0.0",
    "author": "My Name"
}
```

Now you have a new but empty theme to choose in *Administration->Themes*. It is only missing is a *screenshot.png* file to display.

Although it has no view files of its own, when is selected, it will use the original view files of the **src/core/views/** folder.

8.2 Child themes

Child themes is the easiest way to make a new theme by making just a few changes on an existing theme. Make changes directly on another theme is a bad idea, since it can be updated any time and you lose all the adjustments you made.

How to create a child theme:

We choose *gila-blog* as parent, first create a new folder inside **themes/** with name *my-child-theme* and a *package.json*:

```
{
    "name": "My Child Theme",
    "version": "1.0.0",
    "author": "My Name",
    "parent": "gila-blog"
}
```

We create *load.php* file:

```
<?php

View::$parent_theme = 'gila-blog';
... copy here the code of themes/gila-blog/load.php
```

We need to let view class know which is the parent theme, so first the rendering methods will try to find the view files in the parent theme when they fail to find theme in the selected(child) theme.

Now inside the folder of this theme you can edit the view files. Giving to the folder a name with prefix **my-** or **custom-** you make sure that the name will not conflict with a public theme from theme manager.

8.3 Cloned themes

If you want to create a new theme based on another one you can just copy its files. In this way you dont have to keep installed the original theme.

How to clone a theme:

Make a copy of *gila-blog* and name it *my-clone-theme*. In *package.json* file change the name:

```
...
    "name": "My Clone Theme",
...
```

8.4 load.php

Some use case of load.php

```
// add new widget areas that theme includes
array_push(Config::$widget_area, 'footer','sidebar','post.after');

// include stylesheet
View::stylesheet('lib/font-awesome/css/font-awesome.min.css');
```

CHAPTER 9

Schemas

9.1 package.json

package.json is used from both packages and themes to set their values. This schema is saved in json format and uses these indeces:

- **name** (string)

Name of the package or theme.

- **version** (string)

Version in (semantic versioning)[https://semver.org/]

- **description** (string)

A small paragraph explaining what the package does.

- **url** (string)

A path to more information for this package.

- **logo** (string)

An image path to display as logo. Used only by packages.

- **screenshot** (string)

An image path to display as screenshot. Used only by themes.

- **parent** (string)

The parent theme. Used only by themes.

- **options** (object)

An array of fields that uses this package as options. See *Field Schema*.

- **permissions** (object)

A list of new permissions that this package uses. Used only by packages.

Example

```
"permissions": {
                "admin":"All administration privileges",
                "admin_user":"Administrate users"
}
```

- **lang** (string)

A relative path to the language prefix that translates the strings set in the package.json like permissions and option labels.

9.2 Table Schema

The table schema is used for a content type. It gives to the application the structure of the database table that stores the data. So the content administration page will be generated on its own and the content creators can manipulate the data with no further code.

- **name** (string)

The name of the database table that data is stored.

- **title** (string)

The title to display at content administration.

- **id** (string)

(Optional) The field name that is the primary key on database table. Default value is *id*.

- **permissions** (assoc array)

(Optional) Associative array of user permissions required to run the actions. Example:

```
'permissions'=>[
    'read'=>['admin','content-contributor','content-editor'],
    'create'=>['admin','content-contributor'],
    'update'=>['admin','content-editor'],
    'delete'=>['admin']
],
```

- **fields** (assoc array)

Associative array of the content fields. They follow the *Field Schema*. Example:

```
'field_name'=> [
    'title'=>'Title',
],
```

- **pagination** (int)

Number of results per page in content administration.

- **lang** (string)

(Optional) A relative path to the language prefix that translates the strings set in thetable schema like permissions and field titles.

- **search_box** (boolean)

(Optional) If true, it displays a search box in content administration.

- **tools** (array)

(Optional) An array of tools that will be displayed in content administration.

- **commands** (array)

(Optional) An array of commands that will be displayed in content administration.

- **search_boxes**

(Optional) An array of field names. Their search filters will be displayed in content administration as.

- **children** (assoc array)

(Optional) References to other content types that are partials of the parent content. The index of a child must be an existing content type. The child is an associative array with two indeces: - **parent_id** (string) The field of child table that points to the parent's id. - **list** (array) The listed fields of child table. The schema of the child must result in the same list of fields.

Example child for a *shop_order* content type:

```
'children'=>[
    'shop_orderitem'=>[
        'parent_id'=>'order_id',
        'list'=>['id','image','product_id','description','qty','cost']
    ]
]
```

- **events** (array of [string,function])

(Optional) The first value is the event name and the second value is the function that will be triggered. The function gets a reference to the specific row of the table. Examples:

```
'events'=>[
    ['change', function(&$row) {
      // runs when a row is created or updated
      // update $row values
    }],
    ['delete', function($id) {
      // runs on deletion of a row
    }]
]
```

- **meta_table** (array)

(Optional) Set a default meta table for the meta fields of schema table. This table will also be created when the schema table is updated. Example:

```
'meta_table'=> ['usermeta', 'user_id', 'vartype', 'value']
```

9.3 Field Schema

Fields are used as options from packages and widgets or as columns from table schemas. When are used as options from packages their format is JSON, the other cases are as php associative arrays.

[Field Schema is still unfinished, please join a chat for more information]

- **title** (string)

The label of the field to display.

- **default** (any)

The default value to use in input field.

- **type** (string)

The field type. Specifies how the data is processed. If *input_type* is not specified it will also be used as input type. These values can be:

- text
- number
- select
- meta

- **input_type** (string)

Specifies what input type will be used. Default values for 1.8.0: select, meta, radio, postcategory, media, textarea, tinymce, checkbox, list (list cannot be used in table schemas)

- **allow_tags** (boolean/string)

Lets the field value keep html tags or remove them. The default value of *allow-tags* is false.

Example

```
"allow_tags": "<a><p><ul><li>",
```

- **options** (boolean/string)

When type is *select* then an option attributes is required for the values

CHAPTER 10

Class Router

The class that manages the routes and request parameters

10.1 controller ()

(static) Register a new controller.

Parameters

- $c:string Controllers name
- $file:string Controller's filepath without the php extension

Example:

```
Router::controller('my-ctrl', 'my_package/controllers/ctrl','myctrl');
```

10.2 add ()

(static) Registers a new route.

Parameters

- $r:string The path
- $fn:function Callback for the route
- $method:string (optional) GET|POST
- $permissions:string (optional) User persmissions that restrict access

Examples:

```
Router::route('some.txt', function(){ echo 'Some text.'; });
# route with regex
Router::route('hello/(.*)', function($x){ echo 'Hello '.$x; });
Router::route('edit_page_/(.*)', function($x){ ... }, 'POST', 'editor');
```

10.3 post ()

(static) Registers a new route.

Parameters

- $r:string The path
- $fn:function Callback for the route
- $permissions:string (optional) User persmissions that restrict access

10.4 get ()

(static) Registers a new route.

Parameters

- $r:string The path
- $fn:function Callback for the route
- $permissions:string (optional) User persmissions that restrict access

10.5 onController ()

(static) Registers a function to run right after the controller class construction.

Parameters

- $c:string The controller's class name
- $fn:function Callback

Example:

```
Router::onController('blog', function(){ blog::ppp = 24; });
```

10.6 action ()

(static) Registers a new action or replaces an existing for a controller.

Parameters

- $c:string The controller's class name
- $action:string The action
- $fn:function Callback

Example:

```
Router::action('blog', 'topics', function(){ blog::tagsAction(); });
```

10.7 onAction ()

(static) Runs after an action and before the display of view file.

Parameters

- $c:string The controller's class name
- $action:string The action
- $fn:function Callback

Example:

```
Router::onAction('blog', 'topics', function(){ View::set('new_variable', 'value'); });
```

10.8 before ()

(static) Registers a function to run before the function of a specific action.

Parameters

- $c:string The controller's class name
- $action:string The action
- $fn:function Callback

10.9 getController ()

(static) Returns the name of the controller that runs. Default: as set in the settings

10.10 getAction ()

(static) Returns the name of the action that runs. Default: 'index'

CHAPTER 11

Class Config

Common methods for Gila CMS

11.1 addLang ()

(static) Adds language translations from a json file.

Parameters

- $path:string Path to the folder/prefix of language json files

Example:

```
Config::addLang('mypackages/lang/');
```

11.2 addList ()

(static) Adds en element in a global array.

- string $list: Name of the list
- mixed $el: Value

11.3 getList ()

(static) Returns the array of a list.

Parameters

- $list:string Name of the list

11.4 widgets ()

(static) Register new widgets.

Parameters

- $list:Array

Example:

```
Config::widgets( ['wdg'=>'my_package/widgets/wdg'] );
```

11.5 content ()

(static) Register new content type.

Parameters

- String $key Name of content type
- String $path Path to the table file

Example:

```
Config::content( 'mytable', 'package_name/content/mytable.php' );
```

11.6 contentInit ()

(static) Make changes on an existing content type.

Parameters

- String $key: Name of content type
- Function $init: Function to run when initializes the content type object

Example:

```
Config::contentInit( 'mytable', function(&$table){
    // unlist a column from content administration
    &$table['fields']['column1']['list] = false;
});
```

11.7 packages ()

(static) Returns an array with the active packages names.

11.8 amenu ()

(static) Add new elements on administration menu.

Parameters

- $items:Assoc menu items

Example:

```
Config::amenu([
  'item'=>['Item','controller/action','icon'=>'item-icon']
]);
```

11.9 amenu_child ()

(static) Add a child element on administration menu.

Parameters

- $key:string Index of the parent item.
- $item:Array The iten

Example:

```
Config::amenu_child('item', ['Child Item','controller/action','icon'=>'item-icon']);
```

11.10 get ()

(static) Gets the value of configuration/option element.

Parameters

- $key:string Index of the element.

11.11 set ()

(static) Sets the value of configuration/option element.

Parameters

- $key:string Index of the element.
- $value:mixed The value to set.

11.12 updateConfigFile ()

(static) Updates the config.php file.

11.13 hash ()

(static) Generates a hash password from a string. Returns hashed password.

- $pass:string The string to be hashed.

11.14 getArray ()

(static) Gets an option value in array form if it was saved in json format.

Parameters

- string $option: Option name.

11.15 dir ()

(static) Creates the folder if does not exist and returns the path.

Parameters

- $path:string Folder path.

11.16 url ()

(static) Generates a url. Return the url path to print.

Parameters

- $str:string The path.
- $args:Array (optional) The query parameters in array.

Examples:

```
$url = Config::url('blog/post',['id'=>1]);`` returns mysite.com/blog/post?id=1
```

11.17 base ()

(static) Generates a url. Returns the full url path to print.

Parameters

- $str:string The path.

Examples:

```
$url = Config::base('blog/post?id=1');`` returns https://mysite.com/blog/post?id=1
```

11.18 mt ()

(static) Returns modification times in seconds.

Parameters

- string/Array $arg: Names of keys. :returns: string/Array

Example:

```
Config::mt('my-table')
```

11.19 updateMt ()

(static) Updates modification time in seconds. You can use this function from your model classes. The *cm* controller runs *updateMt()* for any content type in update action.

Parameters

- string/Array $arg: Names of keys. :returns: string/Array

Example:

```
Config::updateMt('my-table')
```

CHAPTER 12

Class View

Have methods that outputs the HTML

12.1 set ()

(static) Sets a parameter from a controller action that can be used later from View file.

Parameters

- $param:string The parameter name.
- $value:(mixed) The value.

12.2 meta ()

(static) Sets a meta value that is printed later from View::head ().

Parameters

- $meta:string The meta name.
- $value:string The value.

12.3 stylesheet ()

(static) Adds a new stylesheet link that is printed later from View::head ().

Parameters

- $href:string The href attribute from the link.

12.4 script ()

(static) Adds a new script to be included in the output HTML.

Parameters

- $script:string The src attribute from the script.
- $prop:string (optional) A property for the script.

12.5 getThemePath ()

(static) Returns the path of the current theme.

12.6 head()

(static) Prints all the head information in tag.

Parameters

- $meta:Array (optional) Meta values to be printed.

Example

```
View::head([
  'twitter:image:src'=> 'https://domain.com/img.jpg',
  'og:image'=> 'https://domain.com/img.jpg'
]);
```

12.7 getViewFile ()

(static) Returns the path of a file inside theme or package folder. Return false if file is not found.

- $file:string The file path.
- $package:string (optional) The package folder where the file is located if is not found in theme folder.

12.8 setViewFile ()

(static) Overrides the path of a View file.

Parameters

- $file:string The file path.
- $package:string The package folder where the file is located.

Example:

```
View::setViewFile('admin/settings.php','new-package');
/*
src/new-package/Views/admin/settings.php
overrides
themes/my-theme/admin/settings.php
src/core/Views/admin/settings.php
*/
```

12.9 render ()

(static) Prints the View file adding the header.php and footer.php from theme.

Parameters

- $file:string The file path.
- $package:string (optional) The package folder where the file is located if is not found in theme folder.

12.10 renderAdmin ()

(static) Prints the View file adding the admin/header.php and admin/footer.php from theme.

Parameters

- $file:string The file path.
- $package:string (optional) The package folder where the file is located if is not found in theme folder.

12.11 renderFile ()

(static) Prints the view file without header and footer.

Parameters

- $file:string The file path.
- $package:string (optional) The package folder where the file is located if is not found in theme folder.

12.12 includeFile ()

(static) Prints a partial view file.

Parameters

- $file:string Relative path of the View file.
- $package:string (optional) The package folder where the file is located if is not found in theme folder.

12.13 setViewFile ()

(static) Overrides a View file. Overrides file from any package or the theme.

Parameters

- $file:string Relative path of the View file.
- $package:string (optional) The package folder where the file is located if is not found in theme folder.

12.14 menu ()

(static) Displays a menu.

Parameters

- $menu:string (optional) Name of the menu.
- $tpl:string (optional) The View template to generate html.

Example

```
View::menu('mainmenu', 'tpl/menu.php');
```

12.15 widgetArea ()

(static) Prints the widgets of a specific area.

Parameters

- $area:string The widget area name.
- $div:bool (optional) Also print or not the widget inside a tag with its title. Default=true

12.16 getWidgetArea ()

(static) Returns the widgets of a specific area.

Parameters

- $area:string The widget area name.

Example

```
View::getWidgetArea('sidebar');
```

12.17 getWidgetBody ()

(static) Returns the body of a widget type.

Parameters

- $type:string Name of the widget type

- $widget_data:array (optional) The data to be used
- $widget_file:string (optional) Alternative wiget View file

12.18 img ()

(static) Returns the html markup for an image or thumbnail image

Parameters

- $src:string The path of original image.
- $prefix:string (optional) The prefix name of the thumbnail. Default=”
- $max:integer (optional) The maximum dimension of the thumbnail in pixels. Default=180

Example

```
View::img('assets/image.png', 'md/', 500);

// for images in private folders, preix is not used, so you can skip it
View::img('data/uploads/image.png', 500);
```

12.19 thumb ()

(static) Returns the path of a thumbnail image of specified dimensions. If thumbnail does not exist it will create one.

Parameters

- $src:string The path of original image.
- $prefix:string (optional) The prefix name of the thumbnail. Default=”
- $max:int (optional) The maximum width or height of thumbnail in pixels. Default=180

12.20 thumbStack ()

(static) Returns path to revisioned stucked image and the list of stucked photos. If image does not exist it will be created on the fly.

Parameters

- $src_array:Array The images to stack.
- $file:string The name of the stucked image. It must have png extension.
- $max:int (optional) The maximum width or height of thumbnails in pixels.

Example:

```
$img = ["image1.png","image2.png"];
list($file,$stacked) = View::thumbStack($img, "tmp/stacked_file.png",80);

/* Returned values

$file: tmp/stacked_file.png?12
```

(continues on next page)

(continued from previous page)

```
$stacked[0]: ["src"=>"image1.png","src_width"=>200,"src_height"=>1"width"=>80,"height
↪"=>60,"type"=>2,"top"=>0]

$stacked[1]: false (2nd image was not stacked)
*/
```

12.21 thumbSrcset ()

(static) Returns an array of resized versions of an image

Parameters

- $src:string The image path
- $sizes:int array (optional) The maximum sizes in pixels

CHAPTER 13

Class Session

Methods related to the current sessions, link with a user

13.1 userId ()

(static) Returns the id of the current user.

Example

```
$userId = Session::userId();
```

13.2 hasPrivilege ()

(static) Checks if logged in user has at least one of the required privileges. Return True or false.

Parameters

- $pri:string/Array The privilege(s) to check.

13.3 permissions ()

(static) Get the list of the permissions for the current user.

CHAPTER 14

Class Event

Registers and fires events (hooks)

14.1 listen ()

(static) Sets a new function to run when an event is triggered later.

Parameters

- $event:string The event name.
- $handler:(function) The function to call. The functions should expect the parameters sent from *Event::fire()*, and return value when is called from *Event::get()*

14.2 fire ()

(static) Fires an event and calls all handling functions.

Parameters

- $event:string The event name.
- $params:array (optional) Parameters to send to handlers.

14.3 get ()

(static) Fires an event and calls the handling function (only one should be set). Returns the result of the handler.

Parameters

- $event:string The event name.
- $default:mixed The value to return if there was no handler called.

- $params:array (optional) Parameters to send to handler.

14.4 Basic events

- **load** Runs after all packages where loaded. Useful to overrite values
- **head** Runs inside <*head*> tag in a public page
- **foot** Runs inside <*head*> tag in a public page
- **sendmail** Replaces *mail()* function in Sendmail class
- **validateUserPassword** Run from *User::create()* to accept the new password
- **recaptcha.form** Can print add a recaptha input in post form like register/contact-form
- **recaptcha** is called to verify the recaptcha code send from form
- **login.btn** can print new buttons in login form
- **login.callback** Runs from /login/callback endpoint

CHAPTER 15

Class Db

Class db prepare statements for mysql queries to the connected database. We use the global `$db` instance to access its methods.

15.1 query ()

Runs a query and returns the result.

Parameters

- $q:string The query.
- $args:array (optional) Values to prepare the statement.

Examples:

```
$result1 = $db->query("SELECT title,author FROM post;");
$result2 = $db->query("SELECT title,author FROM post WHERE user_id=?;",[Session::user_
↪id()]);
```

15.2 get ()

Runs a query and returns the results as an array.

Parameters

- $q:string The query.
- $args:array (optional) Values to prepare the statement.

Example:

```
$result = $db->get("SELECT title,author FROM post;");
// Returns
[
  0=>[0=>'Lorem ipsum', 'title'=>'Lorem ipsum', 1=>'John', 'author'=>'John'],
  1=>[0=>'Duis aute irure', 'title'=>'Duis aute irure', 1=>'John', 'author'=>'John'],
]
```

15.3 getAssoc ()

Runs a query and returns the results as an associative array.

Parameters

- $q:string The query.
- $args:array (optional) Values to prepare the statement.

15.4 gen ()

Runs a query and returns a generator that yields the rows.

Parameters

- $q:string The query.
- $args:array (optional) Values to prepare the statement.

Example:

```
$generator = $db->gen("SELECT title,author FROM post;");
```

15.5 getOne ()

Runs a query and returns the first result as an array.

Parameters

- $q:string The query.
- $args:array (optional) Values to prepare the statement.

Example:

```
$result = $db->get("SELECT title,author FROM post;");
// Returns
[0=>'Lorem ipsum', 'title'=>'Lorem ipsum', 1=>'John', 'author'=>'John']
```

15.6 getRows ()

Runs a query and returns the results as an array. With rows fetched with mysqli_fetch_row().

Parameters

- $q:string The query.
- $args:array (optional) Values to prepare the statement.

Example:

```
$result = $db->get("SELECT title,author FROM post;");
// Returns
[
  0=>[0=>'Lorem ipsum',1=>'John'],
  1=>[0=>'Duis aute irure',1=>'John'],
]
```

15.7 value ()

Runs a query and returns the value of the first column of the first row of the results.

Parameters

- $q:string The query.
- $args:array (optional) Values to prepare the statement.

Example:

```
$res = $db->get("SELECT title FROM post WHERE id=1;");
// returns
'Lorem ipsum'
```

15.8 getList ()

Runs a query and returns an array with the values of the first columns from the results.

Parameters

- $q:string The query.
- $args:array (optional) Values to prepare the statement.

Example:

```
$titles = $db->get("SELECT title,author FROM post;");
// Returns
[0=>'Lorem ipsum', 1=>'Duis aute irure']
```

15.9 getOptions ()

Returns an associative array using the first column as keys, and the second column as values.

Parameters

- $q:string The query.
- $args:array (optional) Values to prepare the statement.

Example:

```
$res = $db->get("SELECT id, title FROM post;");
// returns
[1=>'Lorem ipsum', 2=>'Lorem ipsum2']


### error ()
Return an error if exists from the last query executed.

Example:
```

$res = $db->get(”SELECT title,author FROM post;”); if ($error = $db->error()) { trigger_error($error); }

```


### close ()
Closes the connection to the database.

Example
```

$db->close();

```

```

CHAPTER 16

Class Table

Class Table is a tool to make queries to the database, that escapes sql injections and checks the user permissions for you.

How to create an instance:

```
$userTable = new Table('user');
$userTable = new Table('src/core/tables/user.php');
```

Parameters

- $name:string the table name or the relevant schema file
- $permissions:assoc the permissions list Default: admin

The permissions that will be send will override this array:

```
[
  'create'=> ['admin'],
  'read'=> ['admin'],
  'update'=> ['admin'],
  'delete'=> ['admin']
]
```

So, by default the created Table instance uses the admin permission, and it will compare them with the permissions that the table schema accepts. The The keys of the array can have a string array of permissions or boolean (true|false) for value.

16.1 name ()

Returns the table name

16.2 id ()

Returns the field name used as primary key

16.3 can ()

Returns true if an action is permited based on permissions. When a field name is no specified the response applies for the default permission for all the fields.

Parameters

- $action:string the action name
- $field:string (optional) field name

The permissions that will be send will override this array:

```
$permissions = user::permissions(Session::user_id());
$userTable = new Table('user', $permissions);
$userTable->can('read', 'password');
$userTable->can('delete'); // create & delete are not specified for fields
```

16.4 getTable ()

Returns all table schema

16.5 getFields ()

Returns field schemas

16.6 getEmpty ()

Returns a new row with empty and predefined values

16.7 getRow ()

Returns a result in assoc arrow

Parameters

- $filters: the filters
- $args: (optional) more arguments

Example

```
$row = $user->getRow(['id'=>1]);
```

Argument keys

- select:array the list of fields to return
- orderby:assoc the prefered order of the results
- limit:int/array the limit values for the query
- page:int the page of total results (calculates the limit values)

16.8 getRows ()

Returns all results

Parameters

- $filters:assoc (optional) the filters
- $args:assoc (optional) more arguments

Example

```
$userNames = $user->getRows([
  'active'=> 1
],
[
  'select'=> ['name'],
  'orderby'=> ['name'=>'ASC'],
  'page'=> 1
]);
```

16.9 getRowsIndexed ()

Like getRows(), but rows are indexed arrays not associative arrays

Parameters

- $filters:assoc (optional) the filters
- $args:assoc (optional) more arguments

16.10 totalRows ()

Returns the number of rows found

Parameters

- $filters:assoc (optional) the filters

16.11 deleteRow ()

Deletes a row from the database table

Parameters

- $id:int the value of primary key

16.12 update ()

Updates or creates table in database based on schema. Can be used from update.php file from your package.

Example

```
Config::content('post','core/tables/post.php');
$postTable = new Table('post');
$postTable->update();
```

CHAPTER 17

More classes

17.1 Class HttpPost

Make easy post requests from the server with the constructor of the class. Use:

```
$postData = ['id'=> 100];
$args = ['type'=> 'x-www-form-urlencoded'];

$response = new HttpPost('https://api.example.com/get', $postData, $args);
$list = $response->json();
```

Parameters

- $url:string the url, or
- $data:assoc array (optional) posted data
- $args:assoc array (optional) options
- $name:string (optional) base name

Option keys

- type: x-www-form-urlencoded|json Default: json
- ignore_errors:boolean Default: true
- header:assoc array of headers
- method:string Default: POST
- url:string the url, applies only on method set() (see below)

17.1.1 body ()

Returns the row contents of the response

17.1.2 json ()

Returns the response data in object format or null

17.1.3 header ()

Returns a header value. If header is not specified, it returns the array of headers

Parameters

- $key: (optional) the header name

17.1.4 set ()

(static) Sets the prefix arguments of a base HttpPost

Parameters

- $name:string the base name
- $args:assoc options to save

Examples

```
$postData = ['id'=> 100];
$args = ['type'=> 'x-www-form-urlencoded'];

// directly to endpoint
$response = new HttpPost('https://api.example.com/get', $postData, $args);

// using a base, you can skip sending empty arguments as third parameter,
// and send the base api name
$args['url] = 'https://api.example.com/';
$args['header'] = ['Authorization'=> 'Bearer <token>'];
HttpPost::set('api_ex', $args);
$response = new HttpPost('get', $postData, 'api_ex');
```

17.2 Class Form

Displays forms

17.2.1 posted ()

(static) It compares the value *formToken* from the request (GET/POST) with the stored token in session. If the name is specified the stored token will be removed in this function. Return boolean.

Parameters

- $name: (optional) the form token name.

17.2.2 verifyToken ()

Compares a value to the stored token in session. Returns boolean

Parameters

- $name: the form token name
- $check: the value

17.2.3 getToken ()

(static) Creates and returns a new form token.

Parameters

- $name: the form token name

17.2.4 hiddenInput ()

(static) Prints a hidden input with the value of the form token.

Parameters

- $name: (optional) the form token name

17.2.5 html ()

(static) Prints the input fields for a form.

Parameters

- $fields:assoc the fields to print as input elements
- $values:assoc (optional) values
- $preffic:string (optional) preffix fot the input names
- $suffix:string (optional) suffix fot the input names

Example

```
Form::html([
  'group'=>[
    'type'=>'select',
    'options'=>[0=>'Group A', 0=>'Group B']
  ]
],
[
  'group'=>1
]);
```

17.2.6 input ()

(static) Prints an input tag.

Parameters

- $name:string the input name
- $op:assoc the field schema
- $ov:string (optional) current value
- $key:string (optional) input label

17.2.7 addInputType ()

(static) Create a new input type for Form class.

Parameters

- $name:string the input type name
- $function:function a function that returns the html

Function Parameters

- $name:string the input name
- $field:assoc the field schema
- $value:mixed the current value

Example

```
Form::addInputType('group-select', function($name, $field, $value) {
  // a web coomponent that will be rendered with vuejs
  $valueProp = 'value="' . $value . '"';
  $dataProp = 'data-group="' . json_encode($field['options']) . '"';
  return "<group-select $valueProp $dataProp></group-select>";
});
```

17.3 Class Cache

Caches data or a page for faster loads.

17.3.1 remember ()

Loads or updates a string.

Parameters

- $name:string The item name to save the data
- $time:int Time in seconds to keep the value
- $fn:function The function that calculates and return the string if it is not cached
- $uniques:array A list of values that expire the cache if they change

Example

```
Cache::remember('post-'.$id, 3600, function($list) use($id){
  return 'Post#'.$id.'. Updated at '.data($list[0]);
}, [Config::mt('post')]);
```

17.3.2 page ()

Saves or loads the rest of the output from cache. The remember() method should be prefered but when you have a lot of requests and the output is not probably going to change soon, this method can give a faster response.

Parameters

- $name:string The item name to save the data
- $time:int Time in seconds to keep the value
- $uniques:array A list of values that expire the cache if they change

Example

```
if(Session::userId()===0) {
  Cache::page('page.post-'.$id, 3600, [Config::mt('post')]);
}
```

17.4 Class UserNotification

17.4.1 send ()

Creates a notification for a user

Parameters

- $user_id:int The id of the receiver
- $type:string The type of notification
- $details:string (optional) The message to display to the receiver
- $url:string (optional) A link that is related to the notification

Example

```
UserNotification::send(3, 'new_user', 'A new user was registered', 'admin/user/40');
```

17.4.2 countNew ()

Returns the number of unread notifications

Parameters

- $type:string (optional) The type of notification

CHAPTER 18

Authentication

In order to make the calls of Web APIs from a different domain you will need to use the token from your user.

How to generate a Token

You can generate a unique token key from the */admin/profile* page. Keep this key in secret.

Use the Token from server calls (PHP)

You can send the token as post parameter:

```
$url = "https://example.com/cm/delete/post";
$token = "<UNIQUE_TOKEN>";
$options = [
  'http' => [
    'method'  => 'POST',
    'header'  => "Content-type: application/json",
    'content' => http_build_query(['id'=>2, 'token'=>$token]),
    'ignore_errors' => true
  ]
];

$context = stream_context_create($options);
$response = file_get_contents($url, false, $context);
```

Authenticate from front-end (Javascript)

In order to make calls from a different domain, you should include the domain of your front-end app in the website's **cors** value. In *config.php* of your installation add:

```
'cors'=> ['myapp.com']
```

From javascript you should authenticate first the user with cedentials and then use the token in your calls.

Example using axios:

```
// authenticate
axios.post('https://example.com/login/auth', {
  email: 'user@mail.com',
  password: 'password'
})
.then((response) => {
  token_key = response.data.token;
});

// send a request
axios.post('https://example.com/cm/delete/post', {
  id: 2,
  token: token_key
})
```

CHAPTER 19

Content Manager

Content Manager controller gets calls from the front end and responds in json format. In order to make these calls sucessfully you need to be recognised as a user who has the permissions for these actions. For your admin user, you can generate a *Unique Token Key* from **/admin/profile** and send it as GET/POST parameter. This token should be kept private and not used from javascript calls.

19.1 /cm/describe

Returns the schema of a content type

Parameters

- `t` The name of the table (GET)

Example:

```
curl 'https://gilacms.com/cm/describe/?t=post&token=<unique_token_key>'
```

Result:

```
{
    "name": "post",
    "title": "Posts",
    "pagination": 15,
    "id": "id",
    "tools": [
        "new_post",
        "csv"
    ],
    "csv": [
        "id",
        "title",
        "slug",
        "user_id",
```

(continues on next page)

(continued from previous page)

```
        "updated",
        "publish",
        "post"
    ],
    "commands": [
        "edit",
        "clone",
        "delete"
    ],
    "lang": "core\/lang\/admin\/",
    "permissions": {
        "create": [
            "admin"
        ],
        "update": [
            "admin"
        ],
        "delete": [
            "admin"
        ],
        "read": [
            "admin"
        ]
    },
    "search-box": true,
    "search-boxes": [
        "user_id"
    ],
    "fields": {
        "id": {
            "title": "ID",
            "style": "width:5%",
            "create": false,
            "edit": false
        },
        "title": {
            "title": "Title"
        },
        "thumbnail": {
            "type": "meta",
            "list": false,
            "input-type": "media",
            "meta-csv": true,
            "mt": [
                "postmeta",
                "post_id",
                "value"
            ],
            "metatype": [
                "vartype",
                "thumbnail"
            ],
            "title": "thumbnail"
        },
        "slug": {
            "list": false,
            "title": "slug"
```

(continues on next page)

(continued from previous page)


```
        },
        "user_id": {
            "title": "User",
            "type": "select",
            "options": {
                "1": "Admin"
            }
        },
        "updated": {
            "title": "Last updated",
            "type": "date",
            "searchbox": "period",
            "edit": false,
            "create": false
        },
        "categories": {
            "edit": true,
            "type": "meta",
            "mt": [
                "postmeta",
                "post_id",
                "value"
            ],
            "metatype": [
                "vartype",
                "category"
            ],
            "title": "Categories",
            "options": []
        },
        "tags": {
            "list": false,
            "edit": true,
            "type": "meta",
            "meta-csv": true,
            "mt": [
                "postmeta",
                "post_id",
                "value"
            ],
            "metatype": [
                "vartype",
                "tag"
            ],
            "title": "Tags"
        },
        "publish": {
            "title": "Public",
            "style": "width:8%",
            "type": "checkbox",
            "edit": true
        },
        "commands": {
            "title": "",
            "eval": "dv='<a href=\"admin\/posts\/'+rv.id+'\">Edit<\/a>';"
        },
        "post": {
```


(continues on next page)

(continued from previous page)

```
            "list": false,
            "title": "Post",
            "edit": true,
            "type": "textarea",
            "input-type": "tinymce",
            "allow-tags": true
        }
    },
    "events": [
        [
            "change",
            {}
        ]
    ]
}
```

19.2 /cm/list_rows

Returns the rows as array

Parameters

- `t` The name of the table (GET)
- `select` (optional) The fields to return (GET)
- `orderby` (optional) Ordering the results: Examples: `id` `id_ASC` `id_DESC` (GET)
- `groupby` (optional) Groups the results by a field or more (comma seperated) (GET)
- `<field_name>` (optional) A filter to apply on any field (GET) More options:
  - `<field_name>[gt]` Greater than
  - `<field_name>[ge]` Greater or equal than
  - `<field_name>[lt]` Less than
  - `<field_name>[le]` Less or equal than
  - `<field_name>[begin]` A string that begins with
  - `<field_name>[end]` A string that ends with
  - `<field_name>[has]` A string includes value
  - `<field_name>[in]` A number that is in a list

19.3 /cm/update_rows

Updates entry

Parameters

- `t` The name of the table (GET)
- `id` The id of row to update or a comma seperated list od ids, if is not set it will create a new entry. (GET)
- `<field_name>` The value of the field for the update or insert action (POST)

19.4 /cm/empty_row

Returns a row with the default values

Parameters

- `t` The name of the table (GET)

19.5 /cm/insert_row

Inserts a new row in the content table

Parameters

- `t` The name of the table (GET)
- `<field_name>` The value of the field for the update or insert action (POST)

19.6 /cm/delete

Deletes a row

Parameters

- `t` The name of the table (GET)
- `id` The id of row to delete (POST)

19.7 /cm/list

Returns the rows as an array of objects in json format. I wont return the total rows Parameters are like `/list_rows`

19.8 /cm/csv

Returns the rows in csv format for download Parameters are like `/list_rows`

19.9 Multiqueries

These actions can be used in a /cm multiquery(https://gilacms.com/blog/43)

- list
- list_rows
- describe

CHAPTER 20

File Manager

File Manager controller gets calls from the front end and responds in json format.

20.1 /fm/dir

Returns the contents of a directory

Parameters

- `path` The relative path (GET/POST)

Example:

```
curl 'https://gilacms.com/fm/dir/?t=assets'
```

Result:

```
[
  {
    "name": image.jpg,
    "size": 145152,
    "mtime": 2019-02-01 11:01:01,
    "mode": 33206,
    "ext": "jpg"
  },
  .....
]
```

20.2 /fm/save

Saves contents in a file

Parameters

- `path` The destination file (GET/POST)
- `contents` The data to save (POST)

20.3 /fm/newfolder

Creates a new folder

Parameters

- `path` The destination folder (GET/POST)

20.4 /fm/newfile

Creates a new empty file

Parameters

- `path` The destination file (GET/POST)

20.5 /fm/move

Renames a folder or a file

Parameters

- `path` The source relative path (GET/POST)
- `newpath` The destination relative path (POST)

20.6 /fm/delete

Deletes a folder or a file

Parameters

- `newpath` The relative path (GET/POST)

CHAPTER 21

Indices and tables

- genindex