

CogentMode Simple Framework (CMSF)

CMSF is a PHP based framework aimed at small custom projects maintained by 1 or 2 developers.  It uses PDO database objects and procedural logic, along with PHP best security practices to build simple sites quickly without requiring an extensive learning curve.  It trades performance and extendibility for simplicity, speed of delivery and flexibility.

**Set-up Instructions:**

To create the file structure, un-archive the project into your preferred web directory. When setting up Apache, make sure that the root of the site is the folder *named* “root” (containing index.php and .htaccess), NOT the top folder.

A file named /site/pregen.json will need to exist and be writable by the web server admin account in order for some modules to work correctly.

Next, the site admin must set all the constant values in site.conf.php. Each is commented with a description of its function. Most can be left with the default values, but the database variables will have to be edited.

To create the table structure, and populate with the required initial data, copy “db\_install.func.php” from the /install folder to /core. Uncomment the “access override” cookie setting commands in /root/index.php. Finally, run the install script at:

http://<domain>/db\_install.func

When the process completes, re-comment the access override and delete “db\_install.func.php” from the core directory.

The initial login is admin/weakpassword. You’ll want to change that immediately.

**Nomenclature:**

**Module**: A "chunk" of functionality, like "login" or "admin".  Could be a single file, or a collection of them.  Also called “feature”, in some places.

**Repository**: The type/location of a group of Modules. “3rd party”, “Core”, or “Custom”.

**Naming Conventions**:

ClassName

methodName

propertyName

function\_name

folder\_name

$variable\_name

$o\_object\_variable\_name

$\_global\_variable\_name,

$\_SESSION[‘\_global\_variable\_name’]

CONSTANT\_NAME

\_GLOBAL\_CONSTANT\_NAME

table\_names

field\_names

file.class.php // Class files

file.func.php // function collections

file.conf.php // configurations and settings

file.tpl.php // layout templates

file.inc.php // For general file inclusions

Custom modules have additional conventions that are important to the safe integration of non-core code. To prevent conflict between any core/native functionality and any included with a custom module, all non-local variables and functions should be preceded by the module name:

function employee\_directory\_getUsers()  
var employee\_directory\_LOGLOC  
employee\_directory\_iCounter

**Structure:**

Top site directory // (e.g. [www.sample.org](http://www.sample.org)) . NOT the “root” directory.

  /3rdparty

    /angular.js (e.g.)

  /core  //collections of related modules, functions and classes

    login.php

    url.php (e.g.)

    /includes

      bootstrap.func.php (global functions and variables)

[db.class.php](http://culttt.com/2012/10/01/roll-your-own-pdo-php-class/)

[log.func.php](http://www.xcruft.com/content/php-error-logging)  // logging, particularly for errors

login.css // module specific style sheets

      url.func.php  // secure file uploading and downloading to offline directories, image handling, including access to offline template images

  /custom //collections like core, but specific to the current project

  /docs

    guidebook.rtf // this document

    license.txt // license for this project

 /root //webserver site root

    index.php

    .htacess // primarily for mod-rewrite and security

    bots.txt

  /install // scripts used for site creation and maintenance MUST BE SET TO RESTRICTED

    db\_install.func.php  // creates a (mostly) empty database

  /site  // variables and files related to this specific site

     /files  // Uploads and other non-code files

     /logs  //output from site logging.  Separate from server logs. Requires chmod 777

     site.conf.php  // a collection of "global" php variables

  /templates

    /themename

      /themeimages

      main.tpl.php

      main.tpl.css

      main.tpl.js

      subsection.tpl.php

      subsection.tpl.css

      /module (same name as module using this sub-theme)

        main.tpl.php

        main.tpl.css

        main.tpl.js

        subsection.tpl.php

        subsection.tpl.css

  /files // uploaded files not involved in site functionality

  /logs // default location for log files

**Basic Architecture:**

"site.conf.php" is included at the top of "bootstrap.func.php ", and nowhere else.

Template files contain replacement blocks, using the defined section name:

*e.g. <?PHP echo $menu ?>*

Replacement blocks are named for where the content fits on the page.  If a module requires a unique block layout, the template should have a module specific subtheme.

"main.tpl.php" will contain only metadata and subsection layout.  No logic or visible replacement blocks.

"main.tpl.php" must have that name, but subsections can be named anything.

  > subsection layout is controlled by main.tpl.php

  > modules will only know theme name and defined blocks.

Replacement blocks must adhere to a defined list.

"themename" is a variable in site.conf.php

**Theming:**

All content is “rendered” to content blocks positioned by themes. Most content blocks are passed to the template as part of $rend\_array. The current valid content blocks are:

$head: The area inside the HTML “Head” tag containing page metadata.

$banner: The area usually at the top of the page that contains site logos, mottos and other descriptive visual elements.

$menu: This area will contain site navigation links, delivered as an array and rendered according to the instructions in this block.

$body: The focal content of the page, usually in the center.

$footer: Small print and metadata usually found at the bottom of a page.

$mod\_include: These are include files to be inserted in the header of the page.

$message: Important information passed from the site to the end-user, such as warnings and confirmations. This is a session variable.

If no information for a block is passed, that section will blank.

All variables are stored in an array called “$rend\_array”, and passed along with the theme:

$rend\_array['menu'] = array('item1','item2','item3');

$out = renderPhpToString(\_THEME\_MAIN, $rend\_array);

echo $out;

The included default “gearshift” theme is intended as a simple template for more advanced themes.

NOTE: Some module, like “wiki”, will require an alternate theme template named “nowrap.tpl.php”. This template excludes headers and menus to present a concise but consistent look for pop-ups and similar content.

**Key Site Variables:**

$\_SESSION[‘\_pagevars’]: An array of name/value pairs sent from a calling page to a called page when a POST is not used. [NOTE: Not currently in use as of 1/2/2019]

$\_SESSION[‘\_message’]: A one-time message sent from one page to the next.

**Security Model:**

The logged in user’s ID and access token are stored in a client-side cookie containing an encoded JSON array. The token must be decoded using a key stored on the server in order to extract or change the contained credentials. The credentials will be set to work only on the main site domain (by default configuration) and will expire at some preset number of minutes (set by the administrator) after the last page refresh..

$token = array();

$token[‘luid’] = "1";

$token['laccess'] = array('1', '3');

$tokencode = jwt::encode($token, \_TOKEN\_KEY);

$cookie\_life = time()+ 60 \* \_COOKIE\_LIFE;

setcookie("token\_cookie", $tokencode, $cookie\_life, "/", \_DOMAIN);`

$tokendecode = jwt::decode($\_COOKIE["token\_cookie"], \_TOKEN\_KEY);

$laccess = $tokendecode->laccess;

The “laccess” values are stored in normalized tables and compiled upon login. All subsequent access validation is against that variable. If it is absent, the user will be prompted to login. The goal of this model is to be stateless and secure from modification or cross-site exploits.

For simplicity’s sake, the access roles are initially restricted to three “Core Roles” (admin, superuser, & user). Core Module access is hardcoded to function based upon Core Roles. Custom Roles only apply to Custom Modules. Menu items and module access are separate. A “user” might be able to see and click on a module link, but if the module is restricted to the “superuser” and “custom1” roles, it would not display anything. Likewise, a menu item could theoretically be hidden from someone with Role rights to it, but they could still use the module if they logged in and then manually entered the correct address.

Custom modules can add new roles to manage access to their functionality. These must be created upon module activation or manually inserted into the “roles” table”, and then assigned by “rid” to any users that require it. Modules will need to reference the role by “rname” since the ID is dynamically assigned. This will initiate a quick “rid” look-up to allow the “confirmAccess” function to continue the comparison.

Usage:

$req\_access = array('e\_d\_manage'); // “e\_d” is short for “employee\_directory”, in this example  
if(!confirmAccess($req\_access)){…}

Note that the “rname” field” is 20 characters, which will be too short for the convention of using the module name in “global” variables to prevent conflict. Since Camshaft is intended for small projects, any reasonably unique, descriptive, and concise prefix should work.

**Custom Modules and Functions:**

url.php: Allows direct access to images and files outside public folders.  Specific extentions, only.

get($name=NULL, $value=false, $option="default"):

  A safer version of PHPs "$\_GET" function, for URL variables.

  The variables are named "p1" (parameter one), "p2", "p3", and "p4".

  Note that, by default, CMSF allows up to 4 variables to be passed in the URL.  However, P1 is always for "Module".  The remaining three variables should be reserved for "sub-features", not user variables.

post($name=NULL, $value=false): Like get(), for $\_POST variables.

renderPhpToString($file, $vars=null):

  Interprets PHP page outside public folders, and renders their results as a string, with internal variables replaced.

confirmAccess($req\_roles): Returns Boolean answer to whether the user has any of the passed roles. Can take either a role id (“rid”) or a role name (“rname”). Custom modules will not know the automatically assigned “rid” of any roles created for that module.

**Core Data Architecture:**

While CMSF has a data abstraction layer and uses generic data access convention, it is primarily written with the MySQL/MariaDB data structure in mind. Usage of other data tools will require changes to the abstractions layer (db.class.php) and install scripts (db\_install.func.php).

users: User table

uid: Numeric incremented ID for users. Key field. Written to $\_SESSION[‘\_luid’] on login.

uname: String used to login. Unique, but changeable. Required.

upass: Hashed password string. Required.

umail: User’s full email address. Required.

active: Account active flag.

lname: User’s Last Name

fname: User’s First Name

mname: User’s Middle Name

pname: User’s preferred name (if not first)

create\_date: Date recrd inserted. Required.

change\_date: Date of last change.

change\_id: uid of last changer.

roles: List of Role Available

rid: Numeric incremented ID for users. Key field.

rname: Display name for role. Unique/

rdesc: Description of role.

userroles: List of roles assigned to users

id: Numeric incremented ID for row. Key field.

uid: User UID. FK to users.

rid: Role ID. FK to roles.

create\_date: Date record inserted. Required.

change\_id: uid of last changer.

menu: List of items and attibutes related to the menu array

mid: Numeric incremented ID for row. Key field.

mname: Machine name of menu item. Adheres to standard variable naming conventions

dname: Display name of menu item.

maddr: Module address. Formatted location of the menu item (e.g. “access-reset” or “mycustomemodule-subfeature1”)

weight: Positive or negative number controlling sort order of menu items. Larger negative “rise” toward the top, large positives fall towards the bottom.

mroles: A comma delimited list of roles the item is viewable for. Unenforced FK to roles.rid.

active: Item active flag.

**Custom Modules**

To install a Custom module, unarchive it and move to the “/custom” folder. For example:

/custom/new\_module/new\_module.php

To activate the module, log in as a site administrator and navigate to “Site Configuration, Modules”. If the module is properly constructed (see the “Writing Your First Custom Module” section), you will see it listed as inactive. Click the module link, and then the “Activate” button on the next page.

This will add that module’s menu and access variables to the appropriate tables, but you will need to assign access to it in “Site Configuration, Menu items” and “Site Configuration, User Roles.”

To remove the module, return to “Site Configuration, Modules”, select the module and click “Deactivate”. This will remove the module from the menu and hide its content. This is an interim state, for use in maintenance and update tasks, as no permanent change is made, yet. The next step would be to return to the same page and click the “Purge” button, which deletes all access entries in the database. Finally, you would need to manually remove the module directory to make it no longer installable from this list.

**Writing Your First Custom Module**

Custom modules are maintained in a dedicated folder, under the /custom folder. That folder will be named the same as the module, all lower case, with underscores (“\_”) in place of spaces. We will use a sample custom project named “Employee Directory” for purposes of this tutorial. In that case, all files would be stored in /custom/employee\_directory.

Each project will consist of at least 2 files, to be stored in the root of that folder: A PHP module file containing *at least* the “gateway” logic for this module, and an “\*.info” file containing metadata required to register the module. Those two files must be named the same as the project:

/custom/employee\_directory/employee\_directory.php

/custom/employee\_directory/employee\_directory.info

The “\*.info” file is in JSON format, and will have at least six values required by the framework:

"name": A reader-friendly name for the module, to be shown in module management.  
"dependencies": An array of additional module IDs required to make the module work.  
"menuname": A reader-friendly name for the module, to be shown on the menu.  
"description": A reader-friendly description for the module, to be shown in module management or on menu mouse-overs.  
"roles": An array of name value pairs of role name and descriptions to be inserted into the roles table upon activation.  
"version": The current version of the module, to be used when checking for dependencies or updates.

For or tutorial module, it might look like this:

{

"name" : "Employee Directory",  
 "dependencies" : [user\_info, field\_formats],

"menuname" : " Employee Directory ",

"description" : "A directory of information about company employees",

"roles" : {

"emp\_dir\_user":"View Employee Directory",

"emp\_dir\_edit":" Edit Employee Directory Information",

"emp\_dir\_admin":"Add & emove Emploees from Directory"

},

"version" : "1.0.1"

}

Additional files, like a “module.config.xml” file for admin configuration or a “module\_maint.php” file to handle activities around managing the module can be added but are not required. Furthermore, since all module files are contained in the module subfolder, they can be named virtually anything, but it is advised to retain the “module\_subfeature.ext” format for ease of maintenance.

If a module has an optional subfolder named “files”, that subfolder will automatically added to the list of “direct access” folders upon activation (/site/pregen.json). That means that files that need client-side accessible URLs can be put in that folder and called using the built in URL decoder. For instance, an image for our theoretical Employee Directory could be dropped in:

/custom/ employee\_directory/files/ book-icon.png

and then called like this:

<img src ='/url/custom-employee\_directory-cmsf\_osm-files/book-icon.png'>

Note that even if a directory is direct access enabled, not all file extenstions are. See site.conf.php to view and edit allowed file extenstions ($\_IMG\_EXTS\_ALLOWED, $\_DOC\_EXTS\_ALLOWED, $\_FIL\_EXTS\_ALLOWED). The different extension types are served up in different ways; either served up, like an image, included like JSON file, or opened, like a PDF.

module.vars:

A planned feature for when a use case arises is a JSON file, similar to “feature.info”, that will include variables that are accessible across the entire Camshaft installation. This would be useful if your new module provides additional functionality to *other* modules, such as more sophisticated image viewing or text editing. These variables would be added to the globally available list upon activation, or when the managing module chose to update them.

Troubleshooting your new module:

Tracking down errors on a theme-driven site can be challenging as it is sometimes difficult to force debug information to the screen. When working on javascript, the “console.log()” method should be used to force the necessary information to your browsers console view. When working with PHP, the custom “write\_log()” function should be used to write the information to the site’s default error log, usually at “/site/logs/site.log”. You can find the full function at “/core/includes/log.func.php” for the full feature set.

**API Reference**

Camshaft has included a large number of handy functions to ease the process of development for the platform. All of the functions and variables listed in this section are globally available.

PHP API:

$\_BASE\_URL – Protocal and domain upon which the site runs. *e.g. https://www.mysite.com/*

$\_TIME\_STAMP – the current time stamp (time()) from when the page loaded.

CustomException – A class extension of PDOException that provides additional class variables for error message and code.

confirmAccess($req\_roles) - Given the access requirements, returns whether or not user should see page content.

convertDirs($dir\_str) – Given the file location section of the URL, returns the intended directory structure.

get($name=NULL, $value=false) – Given a variable name from the $\_GET collection, “cleans” the string and makes it safe for usage. Used primarily to prevent injection tricks.

getLUID() - Returns the ID of the current logged on user.

getMenu() – Builds an array of menu items for the current user. Generally used for replaceable block content. *e.g. $rend\_array['menu'] = getMenu();*

getPersistent($name\_persistent) – Retrieves the named persistent variable value set using “setPersistent”.

killPersistent() – Blanks/Unsets all variables set using “setPersistent”.

post($name=NULL, $value=false) – Given a variable name from the $\_POST collection, “cleans” the string and makes it safe for usage. Used primarily to prevent injection tricks.

renderPhpToString($file, $vars=null) – Given a PHP files and, possibly, an array of variables to replace within that file, merges the values and interprets the PHP into a stream suitable for display.

replaceKeyStrings($str, $arr=null)– Given a string and an array of variables to replace within that string, merges the values and outputs a stream suitable for display.

setMessage($some\_message) – Sets a message to be displayed prominently on the next page load. Used for warnings and alerts.

setPersistent($name\_persistent, $val\_persistent) – Sets a persistent variable, a cookie or server variable (determined by site configuration) of whatever name and value is passed.

sqlStringify($strIn) – Given a variable, ensures it is a string. Used primarily to prevent submitted values from being used in a SQL injection attack.

JavaScript API:

ajaxErrors(jqXHR, exception) – Given the JQuery XMLHttpRequest object and exception from the $.ajax function, returns a human readable error message to the console.

futureUS(DaysToAdd) - Given a number of days future, returns that date in MM/DD/YYYY format.

isDate(txtDate) – Validates whether or not date string is in MM/DD/YYYY format”.

isNumeric(n) – Given a value, returns true if it is numeric and false, otherwise.

todayUS() - Returns today's date in MM/DD/YYYY format.

wait(ms) – Given a value in milliseconds, causes an JavaScript script to “wait” that many milliseconds. Only useful when performing synchronous operations.