brianwozeniak / **codeigniter-modular-extensions-hmvc**

This is a fork from wiredesignz's implementation of HMVC for Codeigniter.

20 commits **1** branch **0** releases **0** contributors

Branch: master New pull request Create new file Upload files Find file Clone or download

Brian Wozeniak (Bigwebmaster) This fixes #3 Latest commit 53eac31 on Sep 24, 2013

core	Bug fixes, strict trailing slash support, APP files now override modu…	5 years ago
third_party/MX	This fixes #3	4 years ago
.hg_archival.txt	Initial fork from Bitbucket from commit 868e975 to GitHub	5 years ago
readme.md	Updated readme to include what's new on August 6, 2013	5 years ago

readme.md

Modular Extensions - HMVC (a forked version with better module routing, speed optimizations)

You can find the main repo here:

https://bitbucket.org/wiredesignz/codeigniter-modular-extensions-hmvc

This is a fork via bitbucket commit #868e975 to this github repo.

There are two reasons for having this on GitHub as opposed to a Mercurial repo on bitbucket. The first was to have a repo on GitHub that I can easily pull from for my projects, and the second was to add support for better custom routing within modules.

Wiredesignz did an awesome job with the HMVC extension, however, it would not pick up custom routing files in modules if the first segment of the URI did not match the module's directory name. With my modules I don't necessary always want the route to have to start with the module name for the first segment. Let me explain with an example:

If the URI requested is this:

first-segment/second-segment

What happens normally with Wiredesignz implementation is that first the routing is looked up at:

application/config/routes.php

If nothing is found there that matches the request, then the "first-segment" is used as the module name and searched for in all of your module locations to see if anything matches. In my case I have numerous module locations to act as categories, to make my modules more organized by their purpose. So in this case it looks for a module named "first-segment" in every single module location. If no module name matches "first-segment" in all those locations, then the default controller or 404 controller is loaded next and it ends here.

What I wanted was for these first two steps to happen similiary, but in the case nothing is found after the first two steps, I then wanted all routes.php files to be processed from all modules in all module locations to see if anything exists that matches the URI requested. If any routes match, then the appropriate module's controller and method would be used, otherwise it would proceed as normal to the default or 404 controller. So in other words if you use this fork and still setup your modules and URIs the same as before where the first segment matches the module's name, then the 3rd step would never be processed since it would be found in step 2. For that case, the only time the third method would be processed is if someone comes to your website to a route that doesn't exist as it will try to search through everything at that point parsing all of the routing files.

There will be a negligible hit to peformance if you use routes that don't start with your module's name since more locations are searched and more routes.php file are loaded, however the cost is small and for me worth keep everything contained within modules if you like to keep custom module routing there to make it easier to use in other applications. Simply drop it in a module location and that is it, no worrying about adding routes if the module name doesn't match the route you want to use to load a controller. I did some before and after tests using the original versus this forked version and it took approximately 0.005 seconds longer with the forked version. So the performance hit seems very small if that is important to you.

However with the above changes that decrease performance ever so slightly, we also made numerous other changes that speed up performance. The original extension by wiredesignz was inefficient when it came down to locating modules and controllers due to the fact that for each one it tried to find, it would search every single module location again for every single module to see what it can find. This is redundant since the same exact paths were already searched before if you had more than one module. This version actually maps out all the locations of every module and then caches that result so that when needing to load a module's controller, library, etc, it doesn't have to search all over the place again as it already knows exactly where to look. So if you have quite a few modules these changes should help offset some of the performance costs of having these extra features.

Finally before by default your module location will automatically be seen as a route if the controller name matches the module name unless otherwise specified. I didn't like this as I could easily see forgetting to make a module unaccessable and thus creating unwanted access via a direct URL if only other controllers should have access to a module. For example if you have something at yourmodule/yourcontroller/yourmethod where yourcontroller actually has the same name as yourmodule, then that URI as well as yourmodule or yourmodule/yourmodule would also work unless you add something like this contained in your module/config/routes.php file:

```
<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');
/*
|--------------------------------------------------------------------------
| Remove the default route set by the module extensions
|--------------------------------------------------------------------------
|
| Normally by default this route is accepted:
|
| module/controller/method
|
| If you do not want to allow access via that route you should add:
|
| $route['module'] = "";
| $route['module/(:any)'] = "";
|
*/
$route['yourmodule'] = "";
$route['yourmodule/(:any)'] = "";

/*
|--------------------------------------------------------------------------
| Routes to accept
|--------------------------------------------------------------------------
|
| Map all of your valid module routes here such as:
|
| $route['your/custom/route'] = "controller/method";
|
*/
$route['good-route'] = "yourcontroller/yourmethod";

// Original version would have to have yourmodule at the start of the route for the routes.php to be
read
$route['yourmodule/good-route'] = "yourcontroller/yourmethod";
```

So that is the work around, unfortunately I tend to overlook things and could easily forget to setup that rule at the start to remove the default route for the module and it's controllers. So with this fork by default all those routes are removed and only routes you specify will work. You can change this behavior if you woud like by editing the Router.php file and changing:

protected static $remove_default_routes = TRUE;

to

protected static $remove_default_routes = FALSE;

Final words on this fork

Keep in mind new bugs may be introduced from this fork. If you find any bugs or have any fixes would love to hear from you so that they can be addressed. Use at your own risk and with plenty of testing.

Update Aug 6, 2013

Files have been updated to now support app files taking precedence and overriding module files. To override they must be placed in the appropriate directories with identical names.

Support has also been added to be strict with trailing slashes. I am particular about how my URLs are formed, and want regular pages or endpoints to function the same way as CodeIgniter with no trailing slash. However, URLs that actually represent a directory or have additional URLs under it I do want to have a final slash to indicate that its a parent of other resources. As an example I would prefer the following:

```
http://www.example.com/
http://www.example.com/page
http://www.example.com/directory/
http://www.example.com/directory/page
http://www.example.com/pagewithextensions.html
```

Codeigniter by default does not allow you to choose a structure like this. It will simply accept both ways which is possible to cause duplicate content.

While a subtle change, required many changes to be able to have a website operate this way strictly. If a directory is loaded without the trailing slash, the result will be a 404 page unless you provide routes to honor both situations. The goal is to be able to have you specify in your routing configuration if the URL should or should not have a final slash, that way you have the freedom to use to your preferences, and be strict at the same time only allowing what you specify when it comes to trailing slashes. If you want a trailing slash to be allowed simply add the trailing slash directly in the routing file such as this:

```
$route['mydirectory/'] = "directory_controller/index";
```

This will only allow a URL like the following:

```
http://www.example.com/mydirectory/
```

If you need both the trailing slash version and non-trailing slash version to work simply specify both:

```
$route['mydirectory/'] = "directory_controller/index";
$route['mydirectory'] = "directory_controller/index";
```

If you only want to use standard URIs as generated by CodeIgniter, these changes should not affect you since CodeIgniter strips all trailing slashes for default routes and links. These changes will still require you to format your URLs correctly to link to the proper pages whether it is with or without a trailing slash.

Here is the original documentation provided by wiredesignz for reference:

Support development of Modular Extensions - HMVC

Donate

Modular Extensions - HMVC

Modular Extensions makes the CodeIgniter PHP framework modular. Modules are groups of independent components, typically model, controller and view, arranged in an application modules sub-directory that can be dropped into other CodeIgniter applications.

HMVC stands for Hierarchical Model View Controller.

Module Controllers can be used as normal Controllers or HMVC Controllers and they can be used as widgets to help you build view partials.

Features:

All controllers can contain an $autoload class variable, which holds an array of items to load prior to running the constructor. This can be used together with module/config/autoload.php, however using the $autoload variable only works for that specific controller.

```
:::php
<?php
class Xyz extends MX_Controller
{
        $autoload = array(
                'helpers'   => array('url', 'form'),
                'libraries' => array('email'),
        );
}
```

The Modules::$locations array may be set in the application/config.php file. ie:

```
:::php
<?php
$config['modules_locations'] = array(
    APPPATH.'modules/' => '../modules/',
);
```

Modules::run() output is buffered, so any data returned or output directly from the controller is caught and returned to the caller. In particular, $this->load->view() can be used as you would in a normal controller, without the need for return.

Controllers can be loaded as class variables of other controllers using $this->load->module('module/controller'); or simply $this->load->module('module'); if the controller name matches the module name.

Any loaded module controller can then be used like a library, ie: $this->controller->method(), but it has access to its own models and libraries independently from the caller.

All module controllers are accessible from the URL via module/controller/method or simply module/method if the module and controller names match. If you add the _remap() method to your controllers you can prevent unwanted access to them from the URL and redirect or flag an error as you like.

Notes:

To use HMVC functionality, such as Modules::run(), controllers must extend the MX_Controller class.

To use Modular Separation only, without HMVC, controllers will extend the CodeIgniter Controller class.

You must use PHP5 style constructors in your controllers. ie:

```
:::php
<?php
class Xyz extends MX_Controller
{
        function __construct()
        {
                parent::__construct();
        }
}
```

Constructors are not required unless you need to load or process something when the controller is first created.

All MY_ extension libraries should include (require) their equivalent MX library file and extend their equivalent MX_ class

Each module may contain a config/routes.php file where routing and a default controller can be defined for that module using:

```
:::php
<?php
$route['module_name'] = 'controller_name';
```

Controllers may be loaded from application/controllers sub-directories.

Controllers may also be loaded from module/controllers sub-directories.

Resources may be cross loaded between modules. ie: $this->load->model('module/model');

Modules::run() is designed for returning view partials, and it will return buffered output (a view) from a controller. The syntax for using modules::run is a URI style segmented string and unlimited variables.

```
:::php
<?php
/** module and controller names are different, you must include the method name also, including 'index'
**/
modules::run('module/controller/method', $params, $...);

/** module and controller names are the same but the method is not 'index' **/
modules::run('module/method', $params, $...);

/** module and controller names are the same and the method is 'index' **/
modules::run('module', $params, $...);

/** Parameters are optional, You may pass any number of parameters. **/
```

To call a module controller from within a controller you can use $this->load->module() or Modules::load() and PHP5 method chaining is available for any object loaded by MX. ie: $this->load->library(‘validation’)->run().

To load languages for modules it is recommended to use the Loader method which will pass the active module name to the Lang instance; ie: $this->load->language('language_file');

The PHP5 spl_autoload feature allows you to freely extend your controllers, models and libraries from application/core or application/libraries base classes without the need to specifically include or require them.

The library loader has also been updated to accommodate some CI 1.7 features: ie Library aliases are accepted in the same fashion as model aliases, and loading config files from the module config directory as library parameters (re: form_validation.php) have beed added.

$config = $this->load->config(‘config_file’), Returns the loaded config array to your variable.

Models and libraries can also be loaded from sub-directories in their respective application directories.

When using form validation with MX you will need to extend the CI_Form_validation class as shown below,

```
:::php
<?php
/** application/libraries/MY_Form_validation **/
class MY_Form_validation extends CI_Form_validation
{
        public $CI;
}
```

before assigning the current controller as the $CI variable to the form_validation library. This will allow your callback methods to function properly. (This has been discussed on the CI forums also).

```
:::php
<?php
class Xyz extends MX_Controller
{
        function __construct()
        {
                parent::__construct();

                $this->load->library('form_validation');
                $this->form_validation->CI =& $this;
        }
}
```

View Partials

Using a Module as a view partial from within a view is as easy as writing:

```
:::php
<?php echo Modules::run('module/controller/method', $param, $...); ?>
```

Parameters are optional, You may pass any number of parameters.

Modular Extensions installation

1. Start with a clean CI install
2. Set $config['base_url'] correctly for your installation
3. Access the URL /index.php/welcome => shows Welcome to CodeIgniter
4. Drop Modular Extensions third_party files into the CI 2.0 application/third_party directory
5. Drop Modular Extensions core files into application/core, the MY_Controller.php file is not required unless you wish to create your own controller extension
6. Access the URL /index.php/welcome => shows Welcome to CodeIgniter
7. Create module directory structure application/modules/welcome/controllers
8. Move controller application/controllers/welcome.php to application/modules/welcome/controllers/welcome.php
9. Access the URL /index.php/welcome => shows Welcome to CodeIgniter
10. Create directory application/modules/welcome/views
11. Move view application/views/welcome_message.php to application/modules/welcome/views/welcome_message.php
12. Access the URL /index.php/welcome => shows Welcome to CodeIgniter

You should now have a running Modular Extensions installation.

Installation Guide Hints:

-Steps 1-3 tell you how to get a standard CI install working - if you have a clean/tested CI install, skip to step 4.

-Steps 4-5 show that normal CI still works after installing MX - it shouldn't interfere with the normal CI setup.

-Steps 6-8 show MX working alongside CI - controller moved to the "welcome" module, the view file remains in the CI application/views directory - MX can find module resources in several places, including the application directory.

-Steps 9-11 show MX working with both controller and view in the "welcome" module - there should be no files in the application/controllers or application/views directories.

FAQ

Q. What are modules, why should I use them?

A. (http://en.wikipedia.org/wiki/Module)

(http://en.wikipedia.org/wiki/Modular_programming)

(http://blog.fedecarg.com/2008/06/28/a-modular-approach-to-web-development)

Q. What is Modular HMVC, why should I use it?

A. Modular HMVC = Multiple MVC triads

This is most useful when you need to load a view and its data within a view. Think about adding a shopping cart to a page. The shopping cart needs its own controller which may call a model to get cart data. Then the controller needs to load the data into a view. So instead of the main controller handling the page and the shopping cart, the shopping cart MVC can be loaded directly in the page. The main controller doesn't need to know about it, and is totally isolated from it.

In CI we can't call more than 1 controller per request. Therefore, to achieve HMVC, we have to simulate controllers. It can be done with libraries, or with this "Modular Extensions HMVC" contribution.

The differences between using a library and a “Modular HMVC” HMVC class is:

1. No need to get and use the CI instance within an HMVC class
2. HMVC classes are stored in a modules directory as opposed to the libraries directory.

Q. Is Modular Extensions HMVC the same as Modular Separation?

A. Yes and No. Like Modular Separation, Modular Extensions makes modules “portable” to other installations. For example, if you make a nice self-contained model-controller-view set of files you can bring that MVC into another project by copying just one folder - everything is in one place instead of spread around model, view and controller folders.

Modular HMVC means modular MVC triads. Modular Separation and Modular Extensions allows related controllers, models, libraries, views, etc. to be grouped together in module directories and used like a mini application. But, Modular Extensions goes one step further and allows those modules to “talk” to each other. You can get controller output without having to go out through the http interface again.