John Squibb

Build a PHP MVC Framework in One Hour.

For every language out there, there is a multitude of corresponding frameworks that have been written, tested, extended, and proven. Why then, would you want to roll your own? Maybe you haven't found a framework that suits you. Maybe you've got a vault of code in the treasure chest that you have been saving up, and just wish you could take the time to organize it so that is reusable in future projects. Maybe you are bored or fall into a mile-long list of other reasons that would persuade you to craft your own.

We're going to show you how to roll your own framework, using the Model-View-Controller pattern.

Because we are going to do this in such a short amount of time, you must understand the following:

- It's going to be simple.
- It's going to be light.
- It may not take into consideration advanced practices or security measures.

Learning Curve

We are going to assume the following:

- You know PHP, preferably version 5 looking forward to 6.
- You at least understand the tenets of object oriented design.
- You know how to set up PHP include paths on your server, have the ability to change permissions settings, configure apache, etc.
- We are developers in a LAMP world, and will be writing from that perspective.

Getting Started

In your web root, create the following folders:

- models
- views
- controllers

Now, in the web root, create a file called:

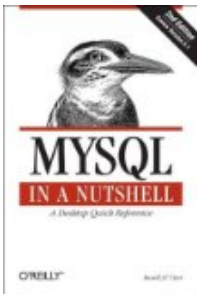
- index.php

Your application hierarchy should now look something like this (depending on your sort method):

- WEB_ROOT_FOLDER

Suggested Reading

MySQL in a Nutshell
by Russel Dyer



MySQL in a Nutshell, like all nutshell books by O'Reilly, is a great reference guide to the multitude of available MySQL functions, administrator tasks, and programming APIs. It organizes the functions by category -- string, date, mathematical, and so forth -- and provides a quick rundown of each function's usage. It also has a great set of appendices, my personal favorite being Appendix A, which provides a concise list of data types and their storage limitations.

- index.php
- models/
- views/
- controllers/

The index.php is going to be the Front Controller of your application, meaning that all page requests will go through it. We are going to use some code to route all user requests to the appropriate files in the controllers folder. When we are done, your users will navigate pages using links like the following:

- *http://yoursite.com/index.php?page1*
- *http://yoursite.com/index.php?page2*
- *http://yoursite.com/index.php?page3*

More on how this is going to work in a bit.

For the remainder of this article, we are going to assume the following server layout:

- web root: /var/www/
- domain name: http://www.yourdomain.com

if your paths are different, you will need to adjust them accordingly throughout the examples. Luckily, you should only have to change them in one location! Hey this framework stuff is working out already!

Setting up the Front Controller

In index.php we are going to define our web root and domain name so they are available to the whole application:

```php
<?php

/**
 * WEB_ROOT_FOLDER is the name of the parent folder you created these
 * documents in.
 */
define('SERVER_ROOT' , '/var/www/WEB_ROOT_FOLDER');

//yoursite.com is your webserver
define('SITE_ROOT' , 'http://yoursite.com');
```

Toggle plain-text

The reason we need both of these, is because internally your application needs to be able to include files relative to its root directory, and externally, your images, javascript, links, videos, etc. need to be able to intertwine. Since index.php is going to be the starting point for the entire application, by defining these values here, they will be available to every subsequent file!

Now let's break away from the index for a moment, but keep it open and nearby, as we will be revisiting it soon!

Setting up the Router

In the *controllers* folder you just created, let's create a file called ***router.php***. This file is going to be the handler for all web

page requests. Think of it like you think of your wireless router in your house, it takes connections from the cable or satellite modem, and routes internet to every computer in the house. Your router.php is going to take web page requests passed to index.php and route the request to different files (controllers) in your application.

in ***router.php*** add the following code:

```php
<?php

//fetch the passed request
$request = $_SERVER['QUERY_STRING'];
```

Toggle plain-text

this is going to grab the request string passed to the application. The request string is everything following the '?' in the URL. Using the examples from earlier:

- *http://yoursite.com/index.php?page1*

will yield *page1* in the request string. Let's make this happen by adding the following lines to ***router.php***:

```php
//parse the page request and other GET variables
$parsed = explode('&' , $request);

//the page is the first element
$page = array_shift($parsed);

//the rest of the array are get statements, parse them out.
$getVars = array();
foreach ($parsed as $argument)
{
    //split GET vars along '=' symbol to separate variable, values
    list($variable , $value) = split('=' , $argument);
    $getVars[$variable] = $value;
}

//this is a test , and we will be removing it later
print "The page your requested is '$page'";
print '<br/>';
$vars = print_r($getVars, TRUE);
print "The following GET vars were passed to the page:<pre>".$vars."</pre>";
```

Toggle plain-text

Now we need to include the router from index.php, so update index.php to look like:

```php
<?php
/**
 * Define document paths
 */
define('SERVER_ROOT' , '/var/www/mvc');
define('SITE_ROOT' , 'http://localhost');

/**
 * Fetch the router
 */
require_once(SERVER_ROOT . '/controllers/' . 'router.php');
```

Toggle plain-text

If everything is going smoothly, you should be able to open up your browser and enter the following:

- *http://yoursite.com/index.php?news&article=howtobuildaframework*

you should see the following output:

```
The page you requested is 'news'
The following GET vars were passed to the page:

Array
(
        [article] => howtobuildaframework
)
```

If you don't, check the steps above, make sure your web server is configured appropriately, check syntax, etc.

Now, let's add a page to our website. Once that is done, we will tweak the router.php to serve the page rather than just print the message above.

Create a Controller

Create a document in ***controllers*** called ***news.php*** and add the following class:

```php
<?php
/**
 * This file handles the retrieval and serving of news articles
 */
class News_Controller
{
    /**
     * This template variable will hold the 'view' portion of our MVC for this
     * controller
     */
    public $template = 'news';

    /**
     * This is the default function that will be called by router.php
     *
     * @param array $getVars the GET variables posted to index.php
     */
    public function main(array $getVars)
    {
        //this is a test , and we will be removing it later
        print "We are in news!";
        print '<br/>';
        $vars = print_r($getVars, TRUE);
        print
        (
            "The following GET vars were passed to this controller:" .
            "<pre>".$vars."</pre>"
        );
    }
}
```

Toggle plain-text

Notice we copied and modified the test code from router.php and placed it with some modifications in our default `main()` function in news.php. Let's strip that code out of router.php and make our code look like this:

```php
<?php
/**
 * This controller routes all incoming requests to the appropriate controller
 */

//fetch the passed request
$request = $_SERVER['QUERY_STRING'];

//parse the page request and other GET variables
$parsed = explode('&' , $request);

//the page is the first element
$page = array_shift($parsed);

//the rest of the array are get statements, parse them out.
$getVars = array();
foreach ($parsed as $argument)
{
    //split GET vars along '=' symbol to separate variable, values
    list($variable , $value) = split('=' , $argument);
    $getVars[$variable] = $value;
}

//compute the path to the file
$target = SERVER_ROOT . '/controllers/' . $page . '.php';

//get target
if (file_exists($target))
{
    include_once($target);

    //modify page to fit naming convention
    $class = ucfirst($page) . '_Controller';

    //instantiate the appropriate class
    if (class_exists($class))
    {
        $controller = new $class;
    }
    else
    {
        //did we name our class correctly?
        die('class does not exist!');
    }
}
else
{
    //can't find the file in 'controllers'!
    die('page does not exist!');
}

//once we have the controller instantiated, execute the default function
//pass any GET varaibles to the main method
$controller->main($getVars);
```

Toggle plain-text

Be sure to include the comment added to the top of the script. I have added it now, because the file now has that functionality. If all goes well, you should be able to call the same URL as before and see the message now printed out from the `News_Controller`. Notice that we used `die()` statements to handle errors. Later on, we can replace these with better error handling methods, but for now these will work. Try using one of the other URLs from earlier to throw a 'page does not exist!' error.

Create a Model

Let's make the `News_Controller` do a bit more. Let's presume we have a small selection of news snippets that we would like to serve to the user. It should be the job of the `News_Controller` to call a model to fetch the appropriate news snippets, whether they be stored in a database, a flat file, etc. In the ***models*** folder create a new file called 'news.php'. And add the following code:

```php
<?php
/**
 * The News Model does the back-end heavy lifting for the News Controller
 */
class News_Model
{
    public function __construct()
    {
        print 'I am the news model';
    }
}
```

Toggle plain-text

`News_Controller main()` function to reflect the following: For now, we've got a simple test that will print when the model is instantiated. Now let's load the model. Modify the

```
public function main(array $getVars)
{
    $newsModel = new News_Model;
}
```

Toggle plain-text

Now reload the page and you should see:

```
Fatal error: Class 'News_Model' not found in /var/www/mvc/controllers/news.php on line xx
```

Wait a minute, that's not what we want! We are trying to load a class that does not exist. The reason it does not exist is because we haven't yet loaded the file containing it, */models/news.php*.

Let's take a minute to revisit the ***router.php*** file to add a bit of code to the top of the file:

```
//Automatically includes files containing classes that are called
function __autoload($className)
{
    //parse out filename where class should be located
    list($filename , $suffix) = split('_' , $className);

    //compose file name
    $file = SERVER_ROOT . '/models/' . strtolower($filename) . '.php';

    //fetch file
    if (file_exists($file))
    {
        //get file
        include_once($file);
    }
    else
    {
        //file does not exist!
        die("File '$filename' containing class '$className' not found.");
    }
}
```

Toggle plain-text

This function 'overloads' the autoload functionality built into PHP. Basically, this 'magic function' allows us to intercept the action that PHP takes when we try to instantiate a class that does not exist. By using the __autoload function in our router, we

can tell PHP where to find the file containing the class we are looking for. Assuming that you follow the class and file naming convention set forth in this article, everytime you need to instantiate a class, you can safely do so without having to manually include the file!

Save the changes to router.php and refresh your browser, now you should see:

```
I am the news model
```

Let's create some functionality in the model to provide some articles. For now, we'll simply create a class array with some articles in it, and provide a function to allow the controller to pull an article by name.

Change the news model to the following:

```php
<?php
/**
 * The News Model does the back-end heavy lifting for the News Controller
 */
class News_Model
{
    /**
     * Array of articles. Array keys are titles, array values are corresponding
     * articles.
     */
    private $articles = array
    (
        //article 1
        'new' => array
        (
            'title' => 'New Website' ,
            'content' => 'Welcome to the site! We are glad to have you here.'
        )
        ,
        //2
        'mvc' => array
        (
            'title' => 'PHP MVC Frameworks are Awesome!' ,
            'content' => 'It really is very easy. Take it from us!'
        )
        ,
        //3
        'test' => array
        (
            'title' => 'Testing' ,
            'content' => 'This is just a measly test article.'
        )
    );

    public function __construct()
    {
    }

    /**
     * Fetches article based on supplied name
     *
     * @param string $articleName
     *
     * @return array $article
     */
    public function get_article($articleName)
    {
        //fetch article from array
        $article = $this->articles[$articleName];

        return $article;
    }

}
```

Toggle plain-text

Now, in the news controller, update the main function to this:

```
public function main(array $getVars)
{
    $newsModel = new News_Model;

    //get an article
    $article = $newsModel->get_article('test');

    print_r($article);
}
```

Toggle plain-text

Give yourself a pat on the back if you noticed the security error in passing the unfiltered GET variable to the model. Give yourself an extra pat if it made you cringe. Although it is extremely important to always clean incoming data, don't worry too much about this right now, just keep it in mind for later.

if you call the file like so:

- *http://yourdomain.com/mvc/index.php?news&article=test*

you should see something along the lines of:

```
Array ( [title] => Testing [content] => This is just a measly test article. )
```

Creating the View

Now that we have the model and controller functionality going, the last step is to get the views going. Remember that the view is your presentation layer. It is the portion of your application that users will be most familiar with. Earlier I mentioned that the purpose of the view is to provide a user interface that is separate from the logic. There are many ways to go about this. You can use a templating engine such as Smarty or something similar. You could roll your own, but this can be a pretty daunting task. Lastly you could use PHP views.

For now, PHP views should work just fine. This harks back to the old HTML code with inline PHP statements. The only difference is that all of our logic is being kept out of the view. Consider the following code:

```
<html>
    <head></head>
    <body>
        <h1>Welcome to Our Website!</h1>
        <hr/>
        <h2>News</h2>
        <h4><?=$data['title'];?></h4>
        <p><?=$data['content'];?></p>
    </body>
</html>
```

Toggle plain-text

Notice the inline PHP tags utilizing the PHP shortcut operator. This will output our content directly into the HTML. Drop this code into a file in the ***views*** folder and name it 'news.php'.

Now that we have our view, we need a way to interact with it. In the ***models*** folder create a file called view.php with the following:

```
<?php
/**
```

```
 * Handles the view functionality of our MVC framework
 */
class View_Model
{
    /**
     * Holds variables assigned to template
     */
    private $data = array();

    /**
     * Holds render status of view.
     */
    private $render = FALSE;

    /**
     * Accept a template to load
     */
    public function __construct($template)
    {
        //compose file name
        $file = SERVER_ROOT . '/views/' . strtolower($template) . '.php';

        if (file_exists($file))
        {
            /**
             * trigger render to include file when this model is destroyed
             * if we render it now, we wouldn't be able to assign variables
             * to the view!
             */
            $this->render = $file;
        }
    }

    /**
     * Receives assignments from controller and stores in local data array
     *
     * @param $variable
     * @param $value
     */
    public function assign($variable , $value)
    {
        $this->data[$variable] = $value;
    }

    public function __destruct()
    {
        //parse data variables into local variables, so that they render to the view
        $data = $this->data;

        //render view
        include($this->render);
    }
}
```

Toggle plain-text

This file model will handle the loading and generation of our view. It works by keeping variable assignments passed to it in the `assign()` function in a local data array. The supplied template's existence is checked in `__construct` and is loaded and rendered in `__destruct`. When the view is rendered, the local data array is pushed to the view so that the assigned variables are output to the user interface.

Now, the last thing to do is load the view from the `News_Controller`. Modify *news.php* in the ***controllers*** folder to reflect the following:

```
<?php
/**
 * This file handles the retrieval and serving of news articles
 */
class News_Controller
{
    /**
     * This template variable will hold the 'view' portion of our MVC for this
     * controller
```

```
     */
    public $template = 'news';

    /**
     * This is the default function that will be called by router.php
     *
     * @param array $getVars the GET variables posted to index.php
     */
    public function main(array $getVars)
    {
        $newsModel = new News_Model;

        //get an article
        $article = $newsModel->get_article($getVars['article']);

        //create a new view and pass it our template
        $view = new View_Model($this->template);

        //assign article data to view
        $view->assign('title' , $article['title']);
        $view->assign('content' , $article['content']);
    }
}
```

Toggle plain-text

When you load your page, you should see your HTML template with the values for title and content parsed in appropriately! **That's it, your basic MVC is complete!**

Download Source Files

If you'd like to download the complete files used in this tutorial, choose from the following formats:

- .zip (click to download...)
- .tar.gz (click to download...)

Room for Improvement

Since this is a very quick, basic MVC framework, there is certainly room for a great deal of improvement. Take your framework to the next step by trying the following:

- Create some more pages in your application. Try to create new controllers, models, and views and make them work correctly.
- Eliminate some potential security risks. Be sure to clean all incoming data before trusting it.
- Create a MySQL database model. Simplify your database interactions by creating a model that wraps the common functions of whichever database library you ordinarily use.
- Try integrating a templating system like smarty into your application to handle the views. By using a templating system, you can effectively reduce your views down to pure Markup with all php removed.
- Check out some other frameworks. If you like the general setup of what I showed you, check out the Kohana Swift PHP framework (http://kohanaphp.com). You may also wish to check out the Zend framework(http://framework.zend.com/).
- Check back for part two of this tutorial, coming soon! In part two, we will explore some of the above items and more!

Got this part down? Head on over to Part Two to tackle the next part of this tutorial!

Tags: PHP,MVC,framework,one hour,part one
Short URL: http://sqb.in/62tH