=================================INRTRODUCTION==============================

There are so many PHP frameworks out there one of the first things that you as a developer might ask why in God's name did I create this one? The very short and simple answer to that is because most if not all the others are too complex and stray very far away from being bare bones. In my opinion a good framework should consist of what it needs and nothing else any other special functionality that the developer felt the need to implement to make life easier should be in the form of a plug-in of some sort.

What I wanted to create was a framework a new developer might unzip look at the 4 or so core files, look through the 3 or so folders and get a feel for the application instantly. Too many times did I unzip a framework and was entirely discouraged about the amount of files and directories I had to look through.

If you are like me and you have the need to know absolutely how every single piece of your application works SimpleMvC should be the thing for you just by reading the notes I have in plain English.

===========================NO NO NO!! THIS IS NOT MVC =========================

The thing people need to understand about the MVC design pattern is this it’s not necessarily a pattern but a principle. The reason why MVC is able to cross borders from its origins and make its way just about into every programming language known to man is because it’s a concept and a principle not a strict set of implementation details that have to be followed. This should be more so evident in a language like PHP that, due to the stateless nature of http requests.

If you are not familiar with the MVC design pattern I invite you to do a quick Google research on the topic but in a nutshell it’s a principle of separation that you probably already thought about ( I did) and even used without knowing they had a general name for it.

Over time developers have come up with interpretations of how they think it should be implemented again I invite you to do a quick Google research and join the developer sheep or shepherds but my advice is to try and grasp the concept and not any specific implementation I have no doubt that will be more valuable to you as a developer.

IF YOU CAN’T BOTHER TO GOOGLE it the acronym stands for

MODEL-VIEW-CONTROLLER and it’s basically a way of separating the different components of your application and defining how they should interact with each other (This is where different developers usually start throwing cyber punches).

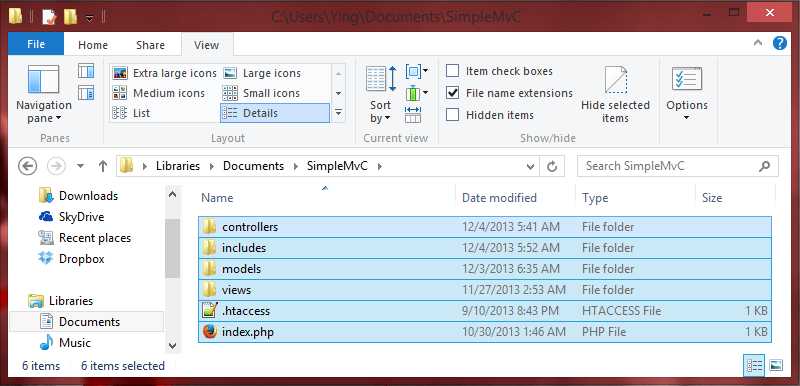
If you are wondering how it is implemented in this framework you will see I practice what I preach when I demonstrate it in the two most common ways I see it being used but for most instances this is entirely up to you. I hope that sorted all questions related to that.

PLEASE DO NOT EMAIL ME ABOUT THIS OR ASK ME ABOUT THIS STATING YOUR OPINIONS!!. EMAIL ME IF YOU THINK I HAVE DONE SOMETHING WRONG AND I WILL EXPLAIN WHY I DID IT THE WAY I DID. THIS IS OPEN SOURCE SO MOST LIKLY IF YOU HAVE A VALID NONE OPINIONATED POINT WE WILL BE WORKING ON VERSION 1.1 TOGETHER :-).

===========================OK DOWN TO THE CODE==========================

Ok so lets get down to it, how do you use this thing and what are the benefits that it provides? Lets see if we can make this “folder structure” as I like to consider it speak for itself.

*Fig.1 “the structure”*

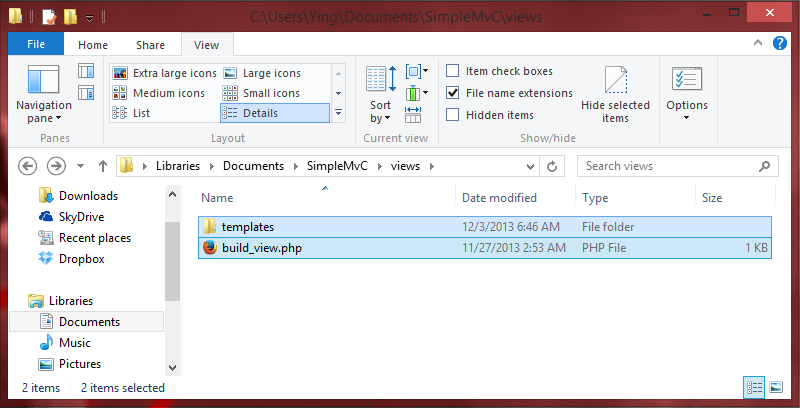


As soon as you unzip the SimpleMvC.zip file you will be presented with a folder structure exactly resembling this one. Within these folders you may create sub-directories or if you so please create entirely new folder structure. But since these are core lets get a description of each.

Models -: Consider this the layer of your application that does the real heavy lifting, the business logic as it is often referred to. How you implement this is entirely up to you but for example purposes I will always include one or two files that can be deleted, edited or modified. Purely as a choice of style I will design my models as if they were and API simply because this leaves for great reusability I will always be returning something a data set, a variable or an object but never null or an explicit action such as a redirect. Should you decide to make that authentication an ajax request would you still need that redirect while trying to authenticate the user? There are more advantages to this that’s not why we are here for more on this watch [Google Talks: How To Design A Good API and Why it Matters](http://www.youtube.com/watch?v=aAb7hSCtvGw). The point is make wise choices because here it’s all your choice.

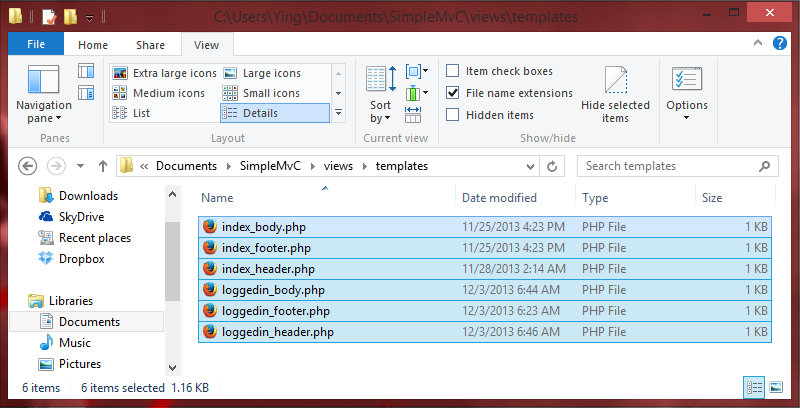
Views -: Hopefully this folder was self-descriptive. Yup this guy holds all those nice little template files that you will be using and reusing all the time to define how your website/app looks

*Fig.2 “the views folder”*



Wait where are the different view files? Exactly where you would imagine, in the ‘*views/templates’* folder. Then why did I ruin the beautiful single layer folder structure by doing that? It easily became tedious to constantly typing the ‘require’ function to load the template files. So a conscious decision was made to have a simple class that’s sole design is to define how to put together an entire page using the name of a header, body and footer filename I pass to it. The question you should be asking yourself now is why I created class methods to define what a simple function should be able to solve? It does seem like a bit of an over kill but the exact reason why I did this is for pure example purposes. What and how you want to define views are 100% up to you. In the case you want to define them as more than just requiring a few files you could define a class or multiple classed where you could do something infinitely more interesting than what I have done here .What this means is that plugins, pre-made template engines (smarty, twig, Rain TPL) and custom temple engines all can be used as effortlessly as possible. What I have done is provide the most simple possible template engine to provide the user with an example and avoid leaving a void in the application.

*Fig.3 “the ../ views/templates/ folder”*



As promised inside the templates sub-folder of the the views folder are just some very simple template files that host the header, body, and footer of any page or multiple pages. With some very minor changes to the code of the default views builder you can change the name of the directory or directories that SimpleMvC will look at for your specified template files. In fact the entire folder structure can be changed just by editing the main config file of SimpleMvC.

Controllers -: Arguably controllers are the most central point of SimpleMvC and I am about to explain why:

This is because of something that I call **The** **SimpleMvC Concept/Philosophy**.

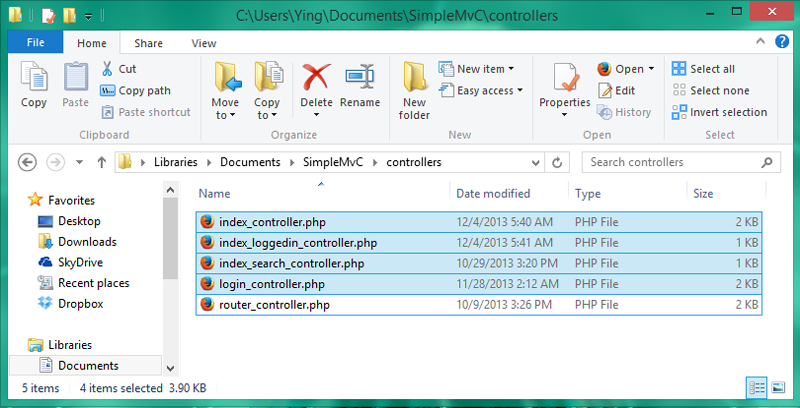
**This is without a doubt the most hardcoded thing in this entire application…….. and its not even code.**

Here is the concept/philosophy “**ANY ACTION THAT OCCURS ON A WEBPAGE IS REPRESENTITIVE OF \*ONE\* SINGLE LOGICAL SEQUENCE OF STEPS AND HENCE CAN BE REPRESENTED AS SUCH**” that is, any click, select or submit that a user has inflicted on a webpage in the attempt to try and get a result of some sort may be represented by an action or any sequence of actions that eventually yields the ‘desired result’.

Developers that are familiar with the concept of OOP or OBJECT ORIENTED PROGRAMMING will quickly recognize this principle as being in cohesion with one of the most important concepts of the technique, the single responsibility principle

So what does all this have to do with SimpleMvC controllers? Everything this principle is what makes SimpleMvC controllers simple and no matter what blend of MvC you are doing or how you choose to implement this, approaching the controllers or anything for that matter with this principle has its clear advantages. I will list those in another section [here](#.com). But realize for now that being able to implement this would lead to less ‘If/else’ clauses more simplified, reusable and clear code.

*Fig.4 “the controllers folder”*



Having controllers exhibit this behavior of one direction is great it makes the life of debugging a breeze and it makes it easy to create and expand on existing features simply by creating a new controller that controls that feature or linking together controllers to expand yet isolate individual features.

But there is a very obvious downside. Doing this one way approach as I say, removing all those if-else statements how can we know if the user is where he is suppose to be or he has entered valid input after all THE USER SHOULD NEVER BE TRUSTED especially not to make our lives easier. There are multiple ways to do this and honestly speaking you could do this in whatever way you choose but this framework just wouldn’t be proof of concept without some solid examples so lets take a look at example controller.

**<?php**

class index\_controller

{

protected $parameters=array();// a list of arguments to the controller

protected $view\_template=array()// a list of the template file names we use to render

protected $object\_factory;// more on this in this short read

public function \_\_construct(object\_factory\_inc $factory,array $parameters )

{

// intialization of parameters (pre-varification setup)

try

{

$this->varify\_controller();

$this->execute();

}

catch(Exception $e)

{

//throw error or call another controller to handle the specific exception

}

}

private function varify\_controller()

{

//does the user have permissions to be here?

//anything that should stop the user from executing this controller

//in this case if user is logged in lets run loggedin version of this controller

}

private function execute()

{

//whatever this actually does. I.E render the index page

$view=$this->object\_factory->build\_view($this->view\_template,$args=array());

$view->render();

}

}

**?>**

**COLOR CODE:**

**BLACK:THE COMMENT THAT YOU SHOULD BE READING TO GET AN UNDERSTANDING OF WHAT DOES WHAT**

**GREEN:FUNCTIONS (KEEP THE CONTROLLER AS SIMPLE AS POSSIBLE, how much green do you see)**

**BLUE:COMMENTS THAT DESCRIBE WHAT I ACTUALLY DID IE EXAMPLE USES**

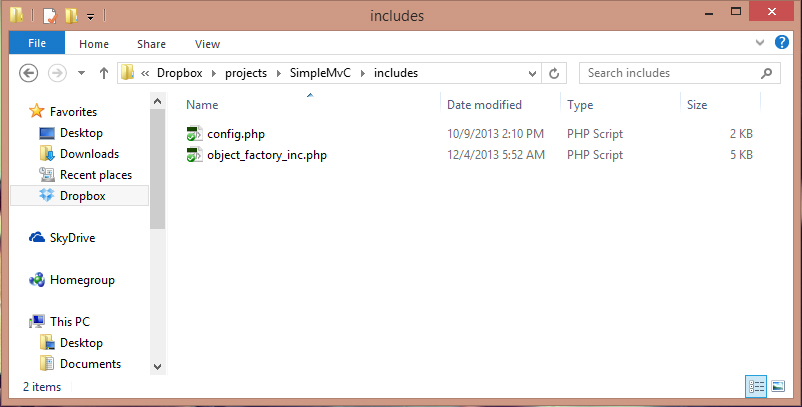
**RED:ACTUAL CODE FROM THE ACTUAL FILES THAT ARE IN USE RIGHT NOW**

For the point we were making about how to handle flags or clauses the sections of interest to us are that try/catch block in the constructer and the verify function. Wrapping our checks in a single method as we did makes debugging that much easier to handle. Further wrapping that verify function in a try/catch block and allowing the different checks to throw an exception in the case something went wrong allows us to handle it. If its an error that we recognize to handle it in anyway we specify and if not to give the user some or the specific error message. For example in the case of my “index\_controller” I exploited this fact and used the verify method to check if the user is actually logged in already and if so throw an exception that when caught initiates an instance of the “index\_loggedin\_controller” and it then takes control of what is going on. This is a cleaner to the alternative of putting that check somewhere in our view templates or in the core of our controller that both breaks our single responsibility principle and reusability.

There is however one decent argument against this approach. We are doing to a too much in our constructers!! This would usually be a compelling argument that is were it not for the SimpleMvC principle which says controllers do one thing and one thing only. That being the case it should be reasonable even rational to say controllers should know how to that one thing since it does it in one way even more it should be able to do it immediately and that is the approach we have taken here. But what should happen if something goes wrong in that controller we are doing too much in? This again is a simple conceptual idea, what is the point of having an object that cannot do the one thing it was created to do? If something were to go wrong such as an error that we did not prepare to catch or any error in general, because these sections of code are in our constructer the object would fail to create. This is ideal an accidental use case is good in the real world but not in the world of computer programming where that would be known as an logic error.

Includes -: if you have any form of previous programming experience you probably already know what is in this directory.

*Fig.5“the includes folder”*



This folder is host to the files that would otherwise not be in direct essence with the normal workflow but would host things such as library files and class blueprints . You can put whatever you choose to here but in here are two files that I would consider core to the framework. If you haven’t already read the specifics of the object factory well that’s exactly what it is a factory for objects in here is where you would define how to instantiate the various objects that your app/website would use. All objects are created through the object factory as a matter of fact it is explicitly required in all the controllers’ constructer. Here is the important thing to remember, all controllers are representative of actions taken by the user or client. Having an object factory allows us to define how the action of the user is interpreted. The best way to demonstrate this is with an example so let’s do that with the ‘index\_controller’ This is what happens in the building of the ‘index\_controller’

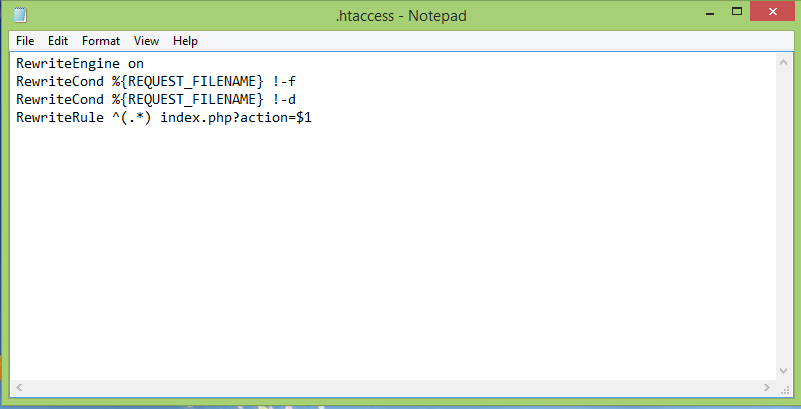
**“When the user visits our domain the .htaacess takes the requested uri and turns it into http://ourwebsite.index.php?action=$requested-uri. This undoubtedly allows only one access point to our app/website. If the uri the user requested just so happens to be null (most times it is just www.ourwebsite.com) the default action is set to index. The router controller (which is a core object) then takes over and uses our action parameter to tell us what controller was requested and gives us any additional arguments the user typed in in a very convenient associative array (which you see in every controllers constructer). If that controller doesn’t exist (the user requested a resource that doesn’t exist) by this time its still very early in our application so the autoload function is able to catch that a requested CONTROLLER dosnt exist. We can do something about it but really we should let the user know they did something wrong so we should throw them a 404 error(In the form of a controller maybe). In the case the requested resource exists then we continue to use the object factory to build that controller.**

**In the case of the index controller we decided that it can have related controllers like the index\_admin and index\_search controller so we decide to use the second parameter of the requested action to determine which one of these should be initiated. This means of isolation allows our controllers to be as simple as possible while maintaining complexity in the form of relationships and being able to define how the different controllers should be interpreted. We then interpret and initiate the requested controller (index\_controller) passing in the relevant arguments we got from the router and an instance of the object factory so we can reuse the same object later and initiate other objects.”**

That in great detail describes the flow that follows how the index controller works and how the object factory in general meshes inside the framework. All of this would have and still is more easily summed up by following its code

The other file in this directory is the config.php and its really very simple so take a look at it all it has is our auto load function and our directory variables. In future releases this will be revised to look much more like standard English.

*Fig.6“htaccess file”*



Very simple, very straight forward this simply says that if the user sends a request that is not a real directory or filename turn anything the user enters into a parameter named action. That’s all this does but is still very relevant to the framework as it ensures all our setup works and there is only one entry point to our app/website

The other file that we are yet to talk about is the index.php. That file is the ground zero for our application and if you ever want to follow the flow of code from the very start you can start from this file. The file is so simple its easier to go read it than listen to me in fact this paragraph exceeds its length minus comments. Errors will rarely be found here in fact I wrote the code and since have never had to go back to this file. The best thing you can ask for as a user is to forget the details that don’t matter to what your doing and as a user I actually forgot how this looked in addition to the config.php and router.php.

=============================Finally the end=============================

If you managed to make it here I congratulate you as you now know all you need to to write this thing from scratch but why would you do that, send me a post on github and we can write version 1.1. If you went ahead and read the code your probably amazed I could explain something you understood so easily in soooo much detail ☺. To the timid programmer like me that read every bit of this seriously it’s that easy just go edit that template I made for you to destroy if that’s possible.

**For any question comments, critique or queries email me at** [**king.zan@hotmail.com**](mailto:king.zan@hotmail.com)**,** [**king.xanda@gmail.com**](mailto:king.xanda@gmail.com) **or send me a post on github @Alex-Nabu .**