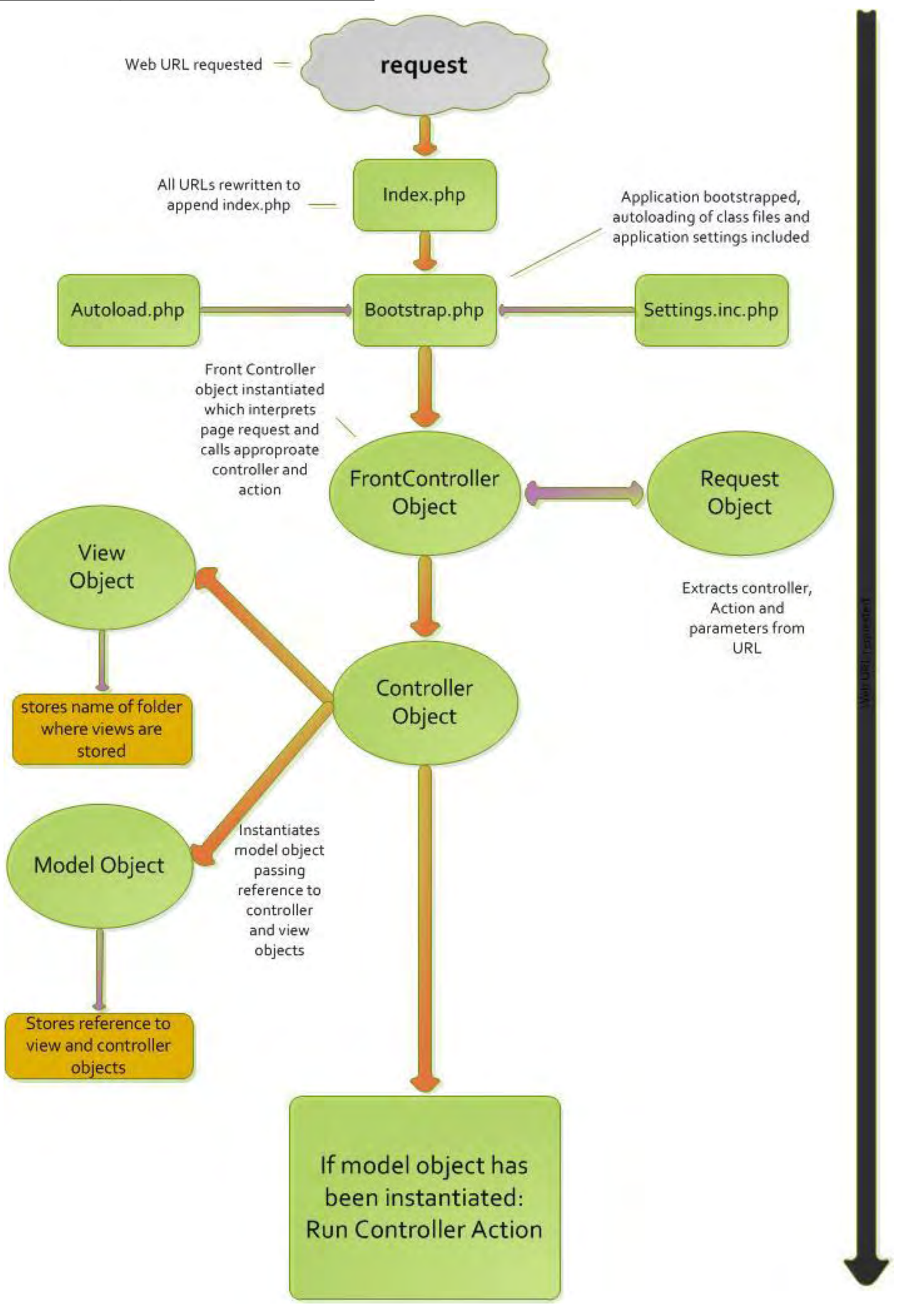
v0.1 Framework Documentation

Here is some collected information about this little framework. It is loosely based on a mode/view/controller framework (although those who know far more than I do would probably argue that this paradigm doesn't really exist for web programming or that this doesn't conform to that model!), and is very much at an early and untested phase. If treated as a boilerplate, it could be used as a way of logically organising code and application layout, while being highly customisable for small to medium sized applications. Far more developed MVC frameworks exist and should be considered, but have a steep learning curve and can take some time to customise as required.

How a page request is handled


Web URL requested
request
All URLs rewritten to append index.php
Index.php
Application bootstrapped, autoloading of class files and application settings included
Autoload.php
Bootstrap.php
Settings.inc.php
Front Controller object instantiated which interprets page request and calls approproate controller and action
FrontController Object
Request Object
Extracts controller, Action and parameters from URL
View Object
stores name of folder where views are stored
Controller Object
Model Object
Instantiates model object passing reference to controller and view objects
Stores reference to view and controller objects
If model object has been instantiated: Run Controller Action

1)Browser requests URL

2)All queries are passed through index.php

3)Application is bootstrapped, calling class autoloaders and application settings

4)A front controller object is instantiated interpreting the URL

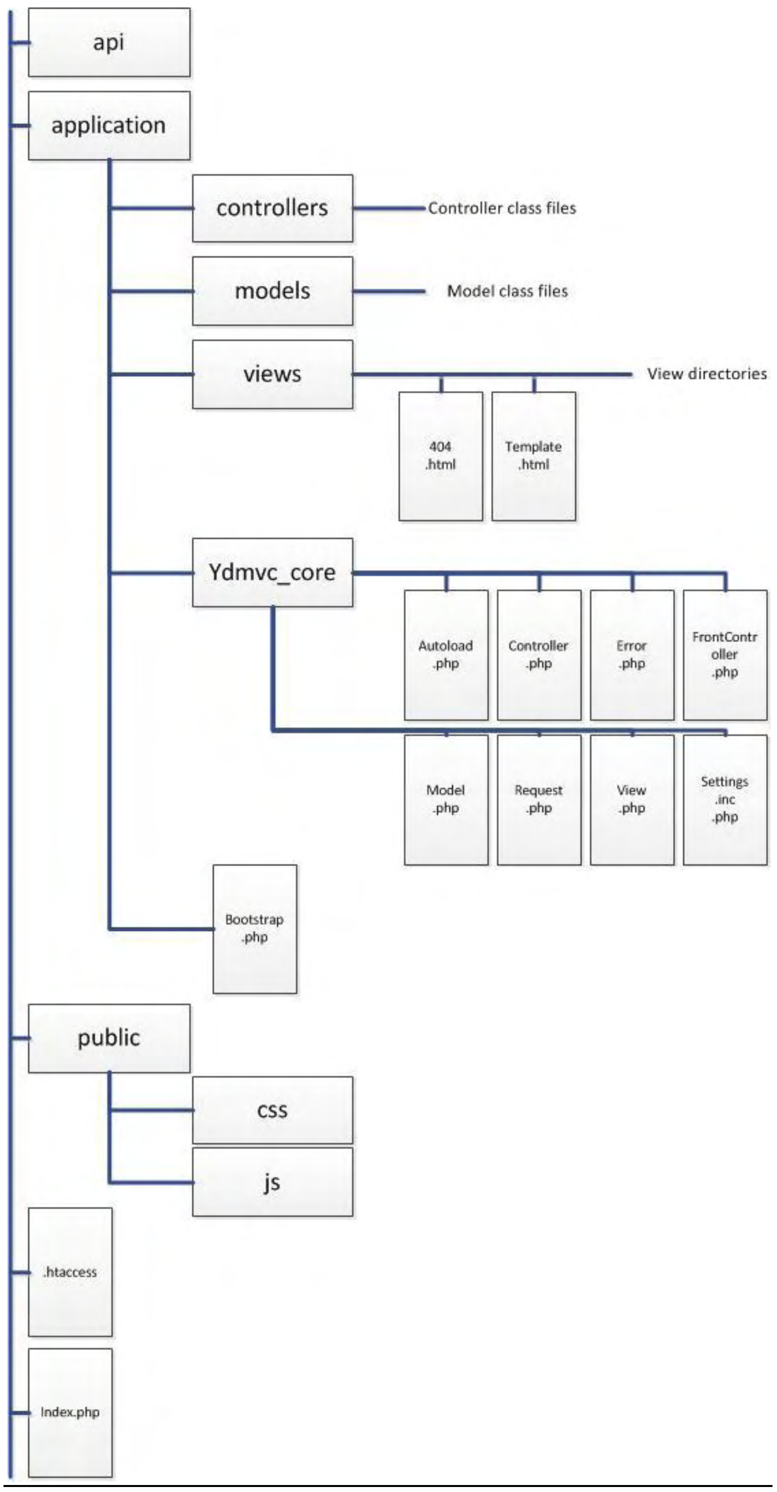
5)An appropriate controller object is instantiated based on URL and this in turn instantiates a view and model object

6)The controller object calls the requested action

URL Structure
http://website.com/controller/action/param1/param2...etc

e.g. http://website.com/book/details/8715671839012 would initiate a controller named book, calling the ‘details’ action and passing a single numerical parameter that might be the ISBN number. It might be that this action displays details of book with the given ISBN. Simple huh?!

File Structure



Creating new controller
When adding creating a new controller to provide a custom piece of functionality the following needs to happen:

1) Create controller class in the application/controllers directory. The controller class file name must follow the convention *[controller_name*]_Controller.php using capitalisation of each word e.g. Book_Controller.php . The new controller class must extend the abstract base controller class:

   ```
   <?php
   class Book_Controller extends Controller {
           actions/functions go here
   }
   ```

2) Create model class in the application/models directory. The model class file name must follow the convention *[controller_name*]_Model.php using capitalisation of each word e.g. Book_Model.php . The new model class must extend the abstract base model class:

   ```
   <?php
   class Book_Model extends Model {
           actions/functions go here
   }
   ```

3) Create a directory in application/views for the view html templates to live in. The directory name must mirror the controller name in lower case e.g. book

Application Settings File
Application/ydmvc_core/settings.inc.php

Define application wide settings e.g. document and url root, db connection details etc

Page template
Application/views/template.html

If enabled in settings file, this template is used to construct common page wrapper and inject content – edit to desired state. Note hook to insert page title if specified in controller or model: **<?php echo $this->title; ?>** and also hook used to inject view file into template: **<?php include("$file"); ?>**

Partials
Application/views/partials

Partials are used to render a partial piece of a view. Useful for keeping view markup clean and easily readable, by removing sections that require a lot of conditional markup or loops e.g. forms or

displaying dynamic data in tables etc. Partials files are created inside the partial folder above and called using a hook in the view file:
<?php echo $this->partial('userform.html',$fruits); ?> this echo's the content of the partial into the current view file, calling the partial by its filename and passing any data that it needs (whether this is a variable, array etc)

Controllers and Models

There are debates as to how controllers and models are utilised, e.g. skinny controller or skinny model. The wider belief is that the controller should be kept as thin as possible, acting as mediator between the view and model. Basically as much business data manipulation as possible is supposed to happen in the model, while the controller should interpret data being submitted by the view and pass it to the appropriate model (e.g. URL parameters, forms). That's my understanding anyway!
Here are a few snippets of core data to get started with…look at the example controller/model/views to get started and customise, expand etc.

- Initialise model action/function from controller: $**this->_model->*function*(*'param'*);**
- Send values to view from controller or model: **$this->_view->setData('*variable*', $data); -** where **'variable'** is the variable name that will be available in the view and **$data** is the value to be passed (value, array etc)
- Dispatch view from controller or model: **$this->_view->load('*view.html*');** - the view file name is optional, if not set then index.html is assumed
- Set page title from either controller or view: **$this->_view->title = "Title";** - assumes that template or pages being used have hook described in paragraphs above to output the title

SQL model methods inherited from abstract base class
Database connection credentials and dsn are set in the settings file, and so far have only been tested with MySQL. Only a couple of database functions exist in their infancy and take advantage of the PDO abstraction class that is included with PHP from about version 5.1. These functions are to be expanded and built upon, to allow easy DB integration in the safest possible way, such as prepared statements etc.

- Set SQL statement to be executed in model: **$this->_setSql($sql); -** where $sql is the $sql string e.g. "SELECT * FROM table";
- Get array containing all rows from sql statement: **$data = $this->_getAll();**
- Insert associative array of data into table – use _setSql function to register a sql insert statement using variable placeholders e.g. **"INSERT INTO User (firstname,surname,fruit) VALUES (:firstname,:surname,:fruit)"** then call **$result = $this->_insertAll($userData);** - where **$userData** is an associative array with key names that match the placeholders e.g. [firstname] => 'Joe', [surname] => 'Blogs' etc. This function may beed to be tweaked to execute inserts with multiple rows being submitted at once?