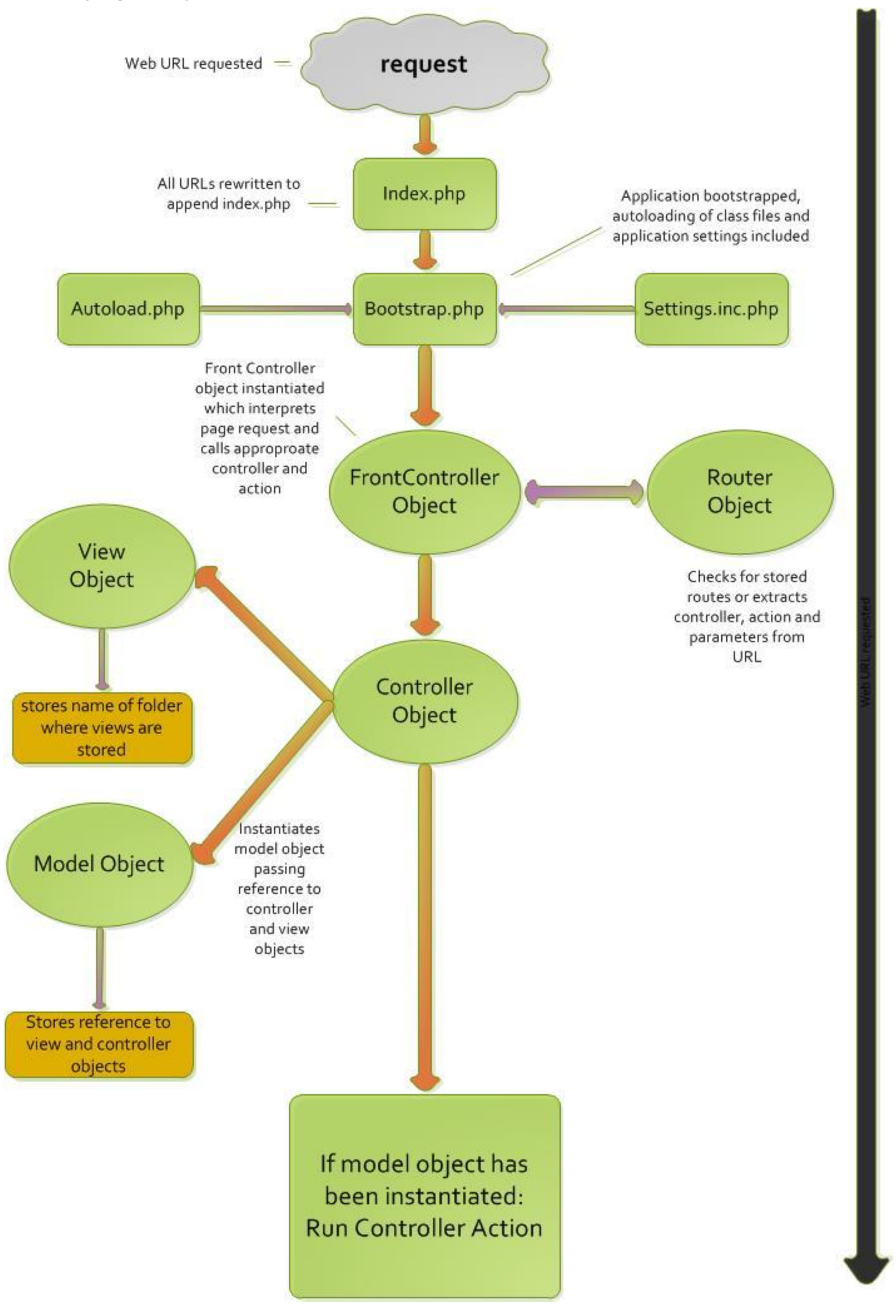
v0.3 Framework Documentation

Here is some collected information about this little framework. It has been conceived as a learning exercise, and was therefore a way to gain a better understanding of the MVC pattern, OOP PHP and latterly the Repository Pattern. It is based on a mode/view/controller framework idea (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
Router Object
View Object
Checks for stored routes or extracts controller, action and parameters from URL
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
Web URL requested

1)Browser requests URL

2)All queries are passed through index.php (apache mod_rewrite)

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

4)A front controller object is instantiated that queries a router object for stored routes or parses 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?!

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 *[controllername*]_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 *[controllername*].php using capitalisation of each word e.g. Book.php . The new model class must extend the abstract base model class:

   ```
   <?php
   class Book 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 model name in lower case e.g. book

Routing

The router objects means that you can define custom URIs that point to a specified controller and action. For example, imagine rather than http://website.com/book/details/8715671839012 to view the details of a specific book, you would rather use the URL structure http://website.com/showbook/87156718839012. Defining a custom route would allow you to do this using the existing book controller/model without sticking to the strict URL structure.

Define custom routes in application/routes.php. The declaration is $this->addRoute(); and accepts 3 values:

1) uri (string) – the custom uri, which can contain placeholders (beginning ':') to represent input parameters in the uri.
2) Route (array) – an associative array of three values: 'controller','action' and 'params'(optional) that represent the controller/action (and fixed params if set) when the custom route is matched.
3) Filters (array)(optional) – an associative array of key => value pairs that match placeholder names and declare a regex filter that the value has to match. If no filters are declared, the placeholders are simply replaced by the corresponding URI value in a matching route without checking.

```
$this->addRoute('/showbook/:isbn',
        array(
            'controller'=>book',
            'action'=>'details',
         ),
         array(
            'isbn'=>'/^[0-9]{13}$/'
         ));
```

In the case of the above route, if the URI is /showbook with a trailing 13 digit number then the route will match and the 'details' action of the 'book' controller will be called with the 13 digit number as the single parameter.

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 inside view files to insert page title if specified in controller or model: **<?php echo $this->title; ?>** and also hook in the template used to inject view file: **<?php include("$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

Connecting a model to the data source

Database connection credentials and dsn are set in the settings file, and so far have only been tested with MySQL.

Having done a bit of reading, and trying to interpret what I found, it seems that an increasingly popular method for communicating with a data source is the use of the "Repository Pattern". This abstracts actual platform specific queries away from the model, meaning the model can contain business logic operations without containing any code referencing a particular storage platform. The advantages of this are that it makes it easier to move the application to a new storage platform e.g. SQLite rather than MySQL, and this approach makes it easier to unit test the model by sending fake/dummy data to it. Moving to a new storage platform would involve editing the repository classes for the models rather than the models themselves.

Each model that requires access to the data source has a class in the repository folder. This class has the same name as the
model with the suffix "_Repository". The repository needs to implement the Repository_Interface class:

```
class Widget_Repository implements Repository_Interface {
   public function findAll(){
      //define findAll
   }

   public function findById($id){
```

```
            //define findById
        }

        public function create($params) {
            //define create
        }

        public function update(){
            //define update
        }

        public function destroy($id){
            //define destroy
        }
    }
```

By implementing the Repository_Interface interface, the repository class has to declare the 5 default functions shown above. In addition it can define any further custom functions.

So within the application you may have a view that displays all the widgets. The controller action might call the displayAllWidgets() method from the widgets model, which needs to actually retrieve a list of widgets from the datasource apply any business logic to the list and sent it to the view. Within the widgets model the function will get the list from the database by calling:

```
    $this->_repository->findAll();
```

This method in the repository actually connects to the database using a db connection class, and performs the actual query:

```
    public function findAll(){
        $this->db->initDB();
        $sql = "SELECT * FROM widgets";
        $query = $this->db->_dbHandle->prepare($sql);
        $query->execute();
        $result = $query->fetchAll();
        $this->db->quitDB();
        return $result;
    }
```