Lab 12. MVC Architecture and PHP Framework

Prepared: TrangNTT

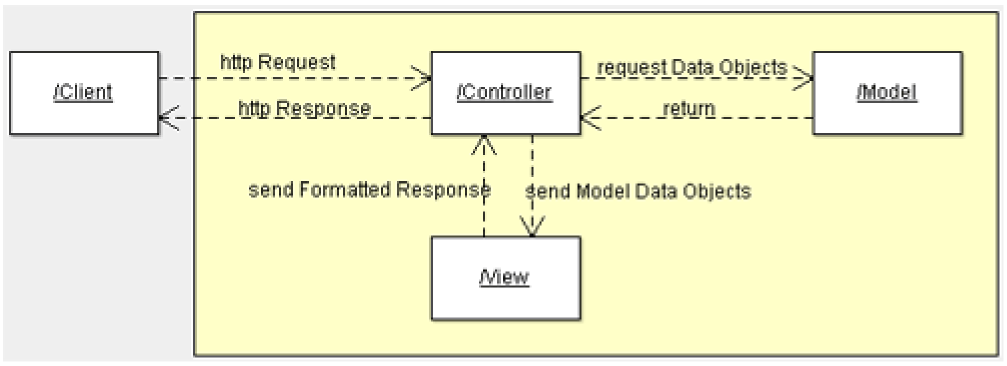
12.1. *A simple MVC project in PHP*

The model view controller pattern is the most used pattern for today’s world web applications. It has been used for the first time in Smalltalk and then adopted and popularized by Java. At present there are more than a dozen PHP web frameworks based on MVC pattern.

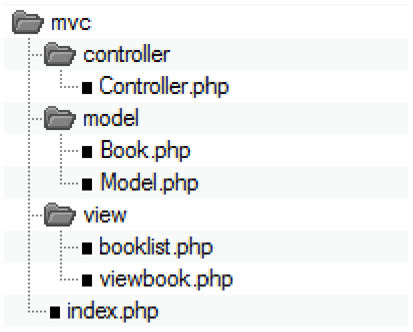
Despite the fact that the MVC pattern is very popular in PHP, is hard to find a proper tutorial accompanied by a simple source code example. That is the purpose of this tutorial.
The MVC pattern separates an application in 3 modules: Model, View and Controller:

- The **model** is responsible to manage the data; it stores and retrieves entities used by an application, usually from a database, and contains the logic implemented by the application.
- The **view (presentation)** is responsible to display the data provided by the model in a specific format. It has a similar usage with the template modules present in some popular web applications, like wordpress, joomla, …
- The **controller** handles the model and view layers to work together. The controller receives a request from the client, invoke the model to perform the requested operations and send the data to the View. The view format the data to be presented to the user, in a web application as an html output.

The above figure contains the MVC Collaboration Diagram, where the links and dependencies between figures can be observed:



Our short php example has a simple structure, putting each MVC module in one folder:



Controller

The controller is the first thing which takes a request, parse it, initialize and invoke the model and takes the model response and send it to the presentation layer. It’s practically the liant between the Model and the View, a small framework where Model and View are plugged in. In our naive php implementation the controller is implemented by only one class, named unexpectedly controller. The application entry point will be index.php. The index php file will delegate all the requests to the controller:

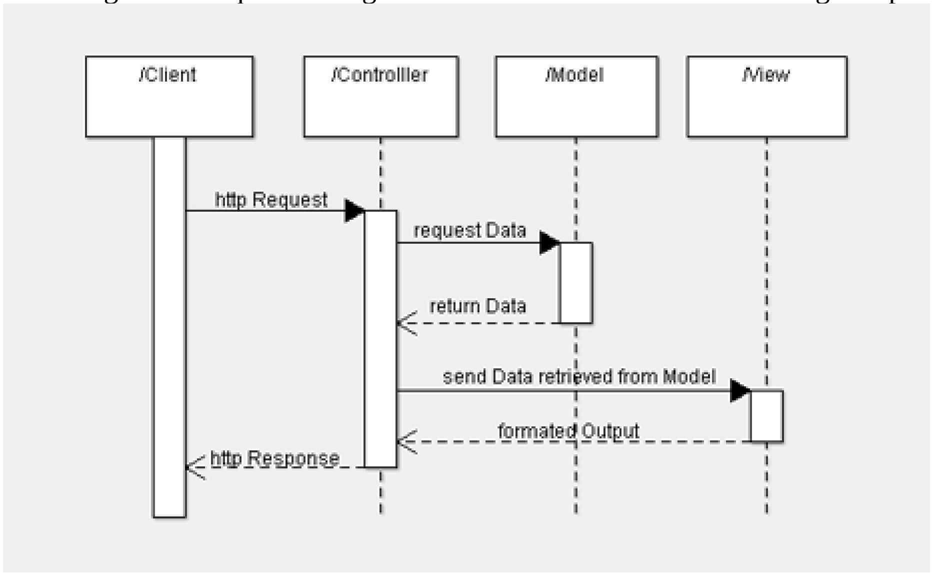
```
// index.php file
include_once("controller/Controller.php");

```

```
$controller = new Controller();
$controller->invoke();
```

Our Controller class has only one function and the constructor. The constructor instantiate a model class and when a request is done, the controller decide which data is required from the model. Then it calls the model class to retrieve the data. After that it calls the corresponding passing the data coming from the model. The code is extremely simple. Note that the controller does not know anything about the database or about how the page is generated.

```
include_once("model/Model.php");

class Controller {
    public $model;

    public function __construct()
    {
        $this->model = new Model();
    }

    public function invoke()
    {
        if (!isset($_GET['book']))
        {
            // no special book is requested, we'll show a list of all available books
            $books = $this->model->getBookList();
            include 'view/booklist.php';
        }
        else
        {
            // show the requested book
            $book = $this->model->getBook($_GET['book']);
            include 'view/viewbook.php';
        }
    }
}
```

In the following MVC Sequence Diagram it can be observed the flow during a http request:



Model and Entity Classes

The Model represents the data and the logic of an application, what many calls business logic. Usually, it’s responsible for:

- Storing, deleting, and updating the application data. Generally it includes the database operations, but implementing the same operations invoking external web services or APIs is not an unusual at all.
- Encapsulating the application logic. This is the layer that should implement all the logic of the application. The most common mistakes are to implement application logic operations inside the controller or the view (presentation) layer.

In our example the model is represented by 2 classes: the “Model” class and a “Book” class. The model doesn’t need any other presentation. The “Book” class is an entity class. This class should be exposed to the View layer and represents the format exported by the Model view. In a good implementation of the MVC pattern only entity classes should be exposed by the model and they should not encapsulate any business logic. Their solely purpose is to keep data. Depending on implementation Entity objects can be replaced by xml or json chunk of data. In the above snippet you can notice how Model is returning a specific book, or a list of all available books:

```
include_once("model/Book.php");

class Model {
    public function getBookList()
    {
        // here goes some hardcoded values to simulate the database
        return array(
            "Jungle Book" => new Book("Jungle Book", "R. Kipling", "A classic book."),
            "Moonwalker" => new Book("Moonwalker", "J. Walker", ""),
            "PHP for Dummies" => new Book("PHP for Dummies", "Some Smart Guy", "")
        );
    }

    public function getBook($title)
    {
        // we use the previous function to get all the books and then we return the requested one.
        // in a real life scenario this will be done through a db select command
        $allBooks = $this->getBookList();
        return $allBooks[$title];
    }

}
```

In our example the model layer includes the Book class. In a real scenario, the model will include all the entities and the classes to persist data into the database, and the classes encapsulating the business logic.

```
class Book {
    public $title;
    public $author;
    public $description;

    public function __construct($title, $author, $description)
    {
        $this->title = $title;
        $this->author = $author;
        $this->description = $description;
    }
}
```

View (Presentation)

The view (presentation layer) is responsible for formatting the data received from the model in a form accessible to the user. The data can come in different formats from the model: simple objects (sometimes called Value Objects), xml structures, json …

The view should not be confused to the template mechanism sometimes they work in the same manner and address similar issues. Both will reduce the dependency of the presentation layer of from rest of the system and separates the presentation elements (html) from the code. The controller delegate the data from the model to a specific view element, usually associated to the main entity in the model. For example the operation “display account” will be associated to a “display account” view. The view layer can use a template system to render the html pages. The template mechanism can reuse specific parts of the page: header, menus, footer, lists and tables... Speaking in the context of the MVC pattern.

In our example the view contains only 2 files one for displaying one book and the other one for displaying a list of books.

viewbook.php

```
<html>
<head></head>

<body>

    <?php

        echo 'Title:' . $book->title . '<br/>';
        echo 'Author:' . $book->author . '<br/>';
        echo 'Description:' . $book->description . '<br/>';

    ?>

</body>
</html>
```

booklist.php

```
<html>
<head></head>

<body>

    <table>
        <tbody><tr><td>Title</td><td>Author</td><td>Description</td></tr></tbody>
        <?php

            foreach ($books as $title => $book)
            {
                echo '<tr><td><a href="index.php?book='.$book->title.'">'.$book->title.'</a></td><td>'.$book->author.'</td><td>'.$book->description.'</td></tr>';
            }

        ?>
    </table>

```

```
</body>
</html>
```

The above example is a simplified implementation in PHP. Most of the PHP web frameworks based on MVC have similar implementations, in a much better shape. However, the possibility of MVC pattern is endless. For example different layers can be implemented in different languages or distributed on different machines. AJAX applications can implements the View layer directly in Javascript in the browser, invoking JSON services. The controller can be partially implemented on client, partially on server…

This post should not be ended before enumerating the advantages of Model View Controller pattern:

- the Model and View are separated, making the application more flexible.
- the Model and view can be changed separately, or replaced. For example a web application can be transformed in a smart client application just by writing a new View module, or an application can use web services in the backend instead of a database, just replacing the model module.
- each module can be tested and debugged separately.

12.2. Build your own PHP MVC Framework

Step 1. Part 1 – Some basic components

What is MVC?

From Wikipedia, Model–View–Controller (MVC) is an architectural pattern used in software engineering. Successful use of the pattern isolates business logic from user interface considerations, resulting in an application where it is easier to modify either the visual appearance of the application or the underlying business rules without affecting the other. In MVC, the model represents the information (the data) of the application; the view corresponds to elements of the user interface such as text, checkbox items, and so forth; and the controller manages the communication of data and the business rules used to manipulate the data to and from the model.

In simpler words-

1. Model handles all our database logic. Using the model we connect to our database and provide an abstraction layer.
2. Controller represents all our business logic i.e. all our ifs and else.
3. View represents our presentation logic i.e our HTML/XML/JSON code.

Why should I write my own framework?

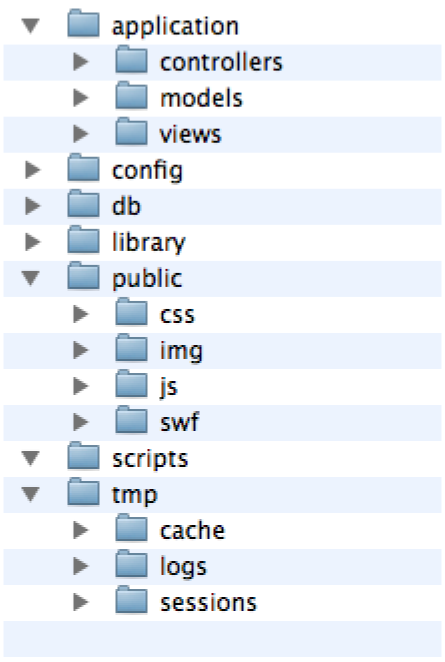
This tutorial is by no means a comprehensive/definitive solution to your framework needs. There are a lot of good php frameworks out there.

So why should you write your own framework? Firstly, it is a great learning experience. You will get to learn PHP inside-out. You will get to learn object-oriented programming, design patterns and considerations.

More importantly, you have complete control over your framework. Your ideas can be built right into your framework. Although always not beneficial, you can write coding conventions/functions/modules the way you like.

Let’s dive right in

The Directory Structure



```
application
    controllers
    models
    views
config
db
library
public
    css
    img
    js
    swf
scripts
tmp
    cache
    logs
    sessions
```

Although we will not be a couple of directories mentioned above for this tutorial, we should have them in place for future expansion. Let me explain the purpose of each directory:

application - *application specific code*
config - *database/server configuration*
db - *database backups*
library - *framework code*
public - *application specific js/css/images*
scripts - *command-line utilities*
tmp - *temporary data*

Once we have our directory structure ready, let us understand a few coding conventions.

Coding Conventions

1. MySQL tables will always be lowercase and plural e.g. items, cars
2. Models will always be singular and first letter capital e.g. Item, Car
3. Controllers will always have "Controller" appended to them. e.g. ItemsController, CarsController
4. Views will have plural name followed by action name as the file. e.g. items/view.php, cars/buy.php

We first add .htaccess file in the root directory which will redirect all calls to the public folder

```
<IfModule mod_rewrite.c>
    RewriteEngine on
    RewriteRule    ^$    public/    [L]
    RewriteRule    (.*) public/$1   [L]
 </IfModule>
```

We then add .htaccess file to our public folder which redirects all calls to index.php. Line 3 and 4 make sure that the path requested is not a filename or directory. Line 7 redirects all such paths to index.php?url=PATHNAME

```
01.<IfModule mod_rewrite.c>
02.RewriteEngine On
03.
04.RewriteCond %{REQUEST_FILENAME} !-f
05.RewriteCond %{REQUEST_FILENAME} !-d
06.
```

```
RewriteRule ^(.*)$ index.php?url=$1 [PT,L]

</IfModule>
```

This redirection has many advantages-
a) we can use it for bootstrapping i.e. all calls go via our index.php except for images/js/cs.
b) we can use pretty/seo-friendly URLS
c) we have a single entry point

Now we add index.php to our public folder

```
<?php

define('DS', DIRECTORY_SEPARATOR);
define('ROOT', dirname(dirname(__FILE__)));

$url = $_GET['url'];

require_once (ROOT . DS . 'library' . DS . 'bootstrap.php');
```

Notice that I have purposely not included the closing ?>. This is to avoid injection of any extra whitespaces in our output. For more, I suggest you view Zend's coding style.

Our index.php basically set the $url variable and calls bootstrap.php which resides in our library directory.

Now let's view our bootstrap.php

```
<?php

require_once (ROOT . DS . 'config' . DS . 'config.php');
require_once (ROOT . DS . 'library' . DS . 'shared.php');
```

Yes these require could be included directly in index.php. But have not been on purpose to allow future expansion of code.

Now let us have a look at shared.php, finally something that does some real work

```
<?php

/** Check if environment is development and display errors **/

function setReporting() {
if (DEVELOPMENT_ENVIRONMENT == true) {
    error_reporting(E_ALL);
    ini_set('display_errors','On');
} else {
    error_reporting(E_ALL);
    ini_set('display_errors','Off');
    ini_set('log_errors', 'On');
    ini_set('error_log', ROOT.DS.'tmp'.DS.'logs'.DS.'error.log');
}
}

/** Check for Magic Quotes and remove them **/

function stripSlashesDeep($value) {
    $value = is_array($value) ? array_map('stripSlashesDeep', $value) : stripslashes($value);
    return $value;
}

function removeMagicQuotes() {
if ( get_magic_quotes_gpc() ) {
```

```php
    $_GET  = stripSlashesDeep($_GET );
    $_POST  = stripSlashesDeep($_POST );
    $_COOKIE = stripSlashesDeep($_COOKIE);
}
}

/** Check register globals and remove them **/

function unregisterGlobals() {
    if (ini_get('register_globals')) {
        $array = array('_SESSION', '_POST', '_GET', '_COOKIE', '_REQUEST', '_SERVER', '_ENV', '_FILES');
        foreach ($array as $value) {
            foreach ($GLOBALS[$value] as $key => $var) {
                if ($var === $GLOBALS[$key]) {
                    unset($GLOBALS[$key]);
                }
            }
        }
    }
}

/** Main Call Function **/

function callHook() {
    global $url;

    $urlArray = array();
    $urlArray = explode("/",$url);

    $controller = $urlArray[0];
    array_shift($urlArray);
    $action = $urlArray[0];
    array_shift($urlArray);
    $queryString = $urlArray;

    $controllerName = $controller;
    $controller = ucwords($controller);
    $model = rtrim($controller, 's');
    $controller .= 'Controller';
    $dispatch = new $controller($model,$controllerName,$action);

    if ((int)method_exists($controller, $action)) {
        call_user_func_array(array($dispatch,$action),$queryString);
    } else {
        /* Error Generation Code Here */
    }
}

/** Autoload any classes that are required **/

function __autoload($className) {
    if (file_exists(ROOT . DS . 'library' . DS . strtolower($className) . '.class.php')) {
        require_once(ROOT . DS . 'library' . DS . strtolower($className) . '.class.php');
    } else if (file_exists(ROOT . DS . 'application' . DS . 'controllers' . DS . strtolower($className) . '.php')) {
        require_once(ROOT . DS . 'application' . DS . 'controllers' . DS . strtolower($className) . '.php');
    } else if (file_exists(ROOT . DS . 'application' . DS . 'models' . DS . strtolower($className) . '.php')) {
        require_once(ROOT . DS . 'application' . DS . 'models' . DS . strtolower($className) . '.php');
    } else {
        /* Error Generation Code Here */
    }
}

```

```
setReporting();
removeMagicQuotes();
unregisterGlobals();
callHook();
```

Let me explain the above code briefly. The *setReporting()* function helps us display errors only when the *DEVELOPMENT_ENVIRONMENT* is true. The next move is to remove global variables and magic quotes. Another function that we make use of is __autoload which helps us load our classes automagically. Finally, we execute the *callHook()* function which does the main processing.

First let me explain how each of our URLs will look - yoursite.com/controllerName/actionName/queryString

So *callHook()* basically takes the URL which we have received from index.php and separates it out as $controller, $action and the remaining as $queryString. $model is the singular version of $controller.

E.g. if our URL is todo.com/items/view/1/first-item, then
Controller is *items*
Model is *item* (corresponding mysql table)
View is *delete*
Action is *delete*
Query String is an *array (1,first-item)*

After the separation is done, it creates a new object of the class $controller."Controller" and calls the method $action of the class.

Now let us create a few classes first namely our base Controller class which will be used as the base class for all our controllers, our Model class which will be used as base class for all our models.

First the controller.class.php

```
<?php
class Controller {

    protected $_model;
    protected $_controller;
    protected $_action;
    protected $_template;

    function __construct($model, $controller, $action) {

        $this->_controller = $controller;
        $this->_action = $action;
        $this->_model = $model;

        $this->$model =&amp; new $model;
        $this->_template =&amp; new Template($controller,$action);

    }

    function set($name,$value) {
        $this->_template->set($name,$value);
    }

    function __destruct() {
            $this->_template->render();
    }

}
```

The above class is used for all communication between the controller, the model and the view (template class). It creates an object for the model class and an object for template class. The object for model class has the same name as the model itself, so that we can call it something like *$this->Item->selectAll();* from our controller.

While destroying the class we call the *render()* function which displays the view (template) file.

Now let us look at our model.class.php

```
<?php
class Model extends SQLQuery {
    protected $_model;

    function __construct() {

        $this->connect(DB_HOST,DB_USER,DB_PASSWORD,DB_NAME);
        $this->_model = get_class($this);
        $this->_table = strtolower($this->_model)."s";
    }

    function __destruct() {
    }
}
```

The Model class extends the SQLQuery class which basically is an abstraction layer for the mySQL connectivity. Depending on your requirements you can specify any other DB connection class that you may require.

Now let us have a look at the SQLQuery.class.php

```
<?php

class SQLQuery {
    protected $_dbHandle;
    protected $_result;

    /** Connects to database **/

    function connect($address, $account, $pwd, $name) {
        $this->_dbHandle = @mysql_connect($address, $account, $pwd);
        if ($this->_dbHandle != 0) {
            if (mysql_select_db($name, $this->_dbHandle)) {
                return 1;
            }
            else {
                return 0;
            }
        }
        else {
            return 0;
        }
    }

    /** Disconnects from database **/

    function disconnect() {
        if (@mysql_close($this->_dbHandle) != 0) {
            return 1;
        }  else {
            return 0;
        }
    }

    function selectAll() {
        $query = 'select * from `'.$this->_table.'`';
        return $this->query($query);
    }
38.
39.    function select($id) {
40.        $query = 'select * from `'.$this->_table.'` where `id` = \''.mysql_real_escape_string($id).'\'';
41.        return $this->query($query, 1);
42.    }
43.
44.    /** Custom SQL Query **/
45.
46.    function query($query, $singleResult = 0) {
47.
48.        $this->_result = mysql_query($query, $this->_dbHandle);
49.
50.        if (preg_match("/select/i",$query)) {
51.        $result = array();
52.        $table = array();
53.        $field = array();
54.        $tempResults = array();
55.        $numOfFields = mysql_num_fields($this->_result);
56.        for ($i = 0; $i < $numOfFields; ++$i) {
57.            array_push($table,mysql_field_table($this->_result, $i));
58.            array_push($field,mysql_field_name($this->_result, $i));
59.        }
60.
61.            while ($row = mysql_fetch_row($this->_result)) {
62.                for ($i = 0;$i < $numOfFields; ++$i) {
63.                    $table[$i] = trim(ucfirst($table[$i]),"s");
64.                    $tempResults[$table[$i]][$field[$i]] = $row[$i];
65.                }
66.                if ($singleResult == 1) {
67.                    mysql_free_result($this->_result);
68.                    return $tempResults;
69.                }
70.                array_push($result,$tempResults);
71.            }
72.            mysql_free_result($this->_result);
73.            return($result);
74.        }
75.
76.    }
77.
78.    /** Get number of rows **/
79.    function getNumRows() {
80.        return mysql_num_rows($this->_result);
81.    }
82.
83.    /** Free resources allocated by a query **/
84.
85.    function freeResult() {
86.        mysql_free_result($this->_result);
87.    }
88.
89.    /** Get error string **/
90.
91.    function getError() {
92.        return mysql_error($this->_dbHandle);
93.    }
94.}
```

The SQLQuery.class.php is the heart of our framework. Why? Simply because it can really help us reduce our work while programming by creating an SQL abstraction layer. We will dive into an advanced version of SQLQuery.class.php in the next tutorial. For now lets just keep it simple.

The *connect()* and *disconnect()* functions are fairly standard so I will not get into too much detail. Let me specifically talk about the query class. Line 48 first executes the query. Let me consider an example. Suppose our SQL Query is something like:

SELECT table1.field1 , table1.field2, table2.field3, table2.field4 FROM table1,table2 WHERE ….

Now what our script does is first find out all the output fields and their corresponding tables and place them in arrays - $field and $table at the same index value. For our above example, $table and $field will look like

$field = array(field1,field2,field3,field4);
$table = array(table1,table1,table2,table2);

The script then fetches all the rows, and converts the table to a Model name (i.e. removes the plural and capitalizes the first letter) and places it in our multi-dimensional array and returns the result. The result is of the form $var['modelName']['fieldName']. This style of output makes it easy for us to include db elements in our views.

Now let us have a look at template.class.php

```php
<?php
class Template {

    protected $variables = array();
    protected $_controller;
    protected $_action;

    function __construct($controller,$action) {
        $this->_controller = $controller;
        $this->_action = $action;
    }

    /** Set Variables **/

    function set($name,$value) {
        $this->variables[$name] = $value;
    }

    /** Display Template **/

    function render() {
        extract($this->variables);

       if (file_exists(ROOT . DS . 'application' . DS . 'views' . DS . $this->_controller . DS . 'header.php')) {
                include (ROOT . DS . 'application' . DS . 'views' . DS . $this->_controller . DS . 'header.php');
            } else {
                include (ROOT . DS . 'application' . DS . 'views' . DS . 'header.php');
            }

        include (ROOT . DS . 'application' . DS . 'views' . DS . $this->_controller . DS . $this->_action . '.php');

            if (file_exists(ROOT . DS . 'application' . DS . 'views' . DS . $this->_controller . DS . 'footer.php')) {
                include (ROOT . DS . 'application' . DS . 'views' . DS . $this->_controller . DS . 'footer.php');
            } else {
                include (ROOT . DS . 'application' . DS . 'views' . DS . 'footer.php');
            }
    }

}
```

The above code is pretty straight forward. Just one point- if it does not find header and footer in the *view/controllerName* folder then it goes for the global header and footer in the *view* folder.

Now all we have to add is a config.php in the config folder and we can begin creating our first model, view and controller!

```
<?php

/** Configuration Variables **/

define ('DEVELOPMENT_ENVIRONMENT',true);

define('DB_NAME', 'yourdatabasename');
define('DB_USER', 'yourusername');
define('DB_PASSWORD', 'yourpassword');
define('DB_HOST', 'localhost');
```

Phew! Now let us create our first mini-todo application. We first create a database "todo" and execute the following SQL queries

```
CREATE TABLE `items` (
  `id` int(11) NOT NULL auto_increment,
  `item_name` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
);

INSERT INTO `items` VALUES(1, 'Get Milk');
INSERT INTO `items` VALUES(2, 'Buy Application');
```

Once that is done, we add item.php to our model folder with the following contents

```
<?php

class Item extends Model {

}
```

Write it is empty, but will have more information when we expand our framework in Part 2.

Now create a file called itemscontroller.php in the controller folder

view sourceprint?

```
<?php

class ItemsController extends Controller {

    function view($id = null,$name = null) {

        $this->set('title',$name.' - My Todo List App');
        $this->set('todo',$this->Item->select($id));

    }

    function viewall() {

        $this->set('title','All Items - My Todo List App');
        $this->set('todo',$this->Item->selectAll());
    }

    function add() {
        $todo = $_POST['todo'];
        $this->set('title','Success - My Todo List App');
        $this->set('todo',$this->Item->query('insert into items (item_name) values
(\''.mysql_real_escape_string($todo).'\')'));
    }

    function delete($id = null) {
        $this->set('title','Success - My Todo List App');
        $this->set('todo',$this->Item->query('delete from items where id = \''.mysql_real_escape_string($id).'\''));
    }

}
```

Finally create a folder called *items* in the views folder and create the following files in it-view.php

```
<h2><?php echo $todo['Item']['item_name']?></h2>

    <a class="big" href="../../../items/delete/<?php echo $todo['Item']['id']?>">
    <span class="item">
    Delete this item
    </span>
    </a>
```

viewall.php

```
<form action="../items/add" method="post">
<input type="text" value="I have to..." onclick="this.value=''" name="todo"> <input type="submit" value="add">
</form>
<br/><br/>
<?php $number = 0?>

<?php foreach ($todo as $todoitem):?>
    <a class="big" href="../items/view/<?php echo $todoitem['Item']['id']?>/<?php echo strtolower(str_replace(" ","-",$todoitem['Item']['item_name']))?>">
    <span class="item">
    <?php echo ++$number?>
    <?php echo $todoitem['Item']['item_name']?>
    </span>
    </a><br/>
<?php endforeach?>
```

delete.php

```
<a class="big" href="../../items/viewall">Todo successfully deleted. Click here to go back.</a>
```

add.php

```
<a class="big" href="../items/viewall">Todo successfully added. Click here to go back.</a>
```

header.php

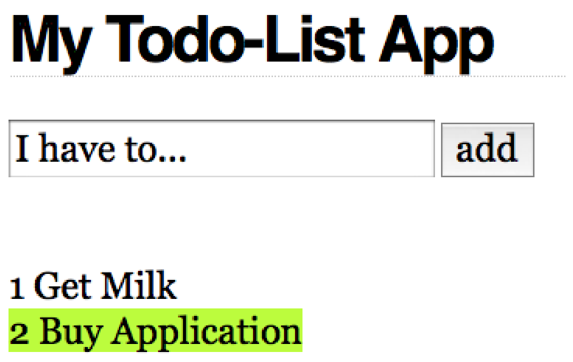
```
<html>
<head>
<title><?php echo $title?></title>
<style>
.item {
width:400px;

}

input {
    color:#222222;
font-family:georgia,times;
font-size:24px;
font-weight:normal;
line-height:1.2em;
    color:black;
}
```

```

a {
    color:#222222;
font-family:georgia,times;
font-size:24px;
font-weight:normal;
line-height:1.2em;
    color:black;
    text-decoration:none;

}

a:hover {
    background-color:#BCFC3D;
}
h1 {
color:#000000;
font-size:41px;
letter-spacing:-2px;
line-height:1em;
font-family:helvetica,arial,sans-serif;
border-bottom:1px dotted #cccccc;
}

h2 {
color:#000000;
font-size:34px;
letter-spacing:-2px;
line-height:1em;
font-family:helvetica,arial,sans-serif;

}
</style>
</head>
<body>
<h1>My Todo-List App</h1>
```

footer.php

```
</body>
</html>
```

Now assuming you have uploaded the directory structure to the todo folder, point your browser to: http://localhost/todo/items/viewall



My Todo-List App

1 Get Milk
2 Buy Application

Where do we go from here?

Firstly we have achieved a lot by completing this tutorial - we have been able to separate our presentation logic from our business logic and database logic. We have also been able to provide for pretty-urls and extensibility. This is just the beginning of our framework. There are loads to do. In the next part, I will be talking about how to make our model and SQLQuery class more robust.

Step 2. Part 2 – Simple e-commerce website

If you haven't already, please read Part 1 of the PHP MVC tutorial. A lot of changes have been incorporated in this part. Unlike the last tutorial, I have not pasted the code directly here, and I suggest you download the framework before reading on. In the example, we have implemented a simple e-commerce website consisting of categories, subcategories, products and tags (for products). The example will help you understand the various relationships between the tables and most of the new functionality provided by this part of the framework.

What's New

config/inflection.php
The inflection configuration file enables us to use irregular words i.e. words which do not have a standard plural name. This file is used in conjunction with *library/inflection.class.php*

config/routing.php
The routing configuration file enables us to specify default controller and action. We can also specify custom redirects using regular expressions. Currently I have specified only one redirect i.e.
http://localhost/framework/admin/categories/view will become
http://localhost/framework/admin/categories_view where admin is the controller and categories_view is the action. This will enable us to create an administration centre with pretty URLs. You can specify others as per your requirements. For more information on how to use regular expressions have a look at one of these tutorials.

library/cache.class.php
The cache class is in its infancy. Currently there are two simple functions- set and get. The data is stored in a flat text-file in the cache directory. Currently only the *describe* function of the SQLQuery class uses this cache function.

library/html.class.php
The HTML class is used to aid the template class. It allows you to use a few standard functions for creating links, adding javascript and css. I have also added a function to convert links to tinyurls. This class can be used only in the views e.g. $html->includeJs('generic.js');

library/inflection.class.php
In the previous part, plural of words were created by adding only "s" to the word. However, for a more full-fledged version, we now use the inflection class originally created by Sho Kuwamoto with slight modifications. If you have a look at the class, it makes use of simple regular expressions. It would be nice to add a cache function to this class in future.

library/template.class.php
I have added a new variable called doNotRenderHeader which will enable you to not output headers for a particular action. This can be used in AJAX calls when you do not want to return the headers. It has to be called by the controller e.g. $this->doNotRenderHeader = 1;

library/vanillacontroller.class.php & vanillamodel.class.php
Pretty much unchanged, but I have added a prefix vanilla to them to emphasize on its simplicity ☺

library/sqlquery.class.php
The SQLQuery class is the heart of this framework. This class will enable you to use your tables as objects.

Let us understand the easy functions first - save() and delete()

The save() function must be used from the controller. The save() function can have two options- if an id is set, then it will update the entry; if it is not set, then it will create a new entry. For example let us consider the categories class i.e. application/controllers/categoriescontroller.php. We can have a function like the one below.

```
function new() {
   $this->Category->id = $_POST['id'];
   $this->Category->name = $_POST['name'];
   $this->Category->save();
}
```

If $this->Category->id = null; then it will create a new record in the categories table.

The delete() function enables you to delete a record from the table. A sample code could be as below. Although you should use POST instead of GET queries for deleting records. As a rule idempotent operations should use GET. Idempotent operations are those which do not change the state of the database (i.e. do not update/delete/insert rows or modify the database in anyway)

```
function delete($categoryId) {
   $this->Category->id = $categoryId;
   $this->Category->delete();
}
```

Now let us look at the search() function. I know it is a bit intimidating at first. But it is pretty straight forward after we break it down. If you are unaware of database table relationships (1:1, 1:Many and Many:Many) have a quick look here. During database design, we use the following convention:

1:1 Relationship

For a one is to one relationship, suppose we have two tables students and mentors and each student hasOne mentor, then in the students table we will have a field called 'mentor_id' which will store the id of the mentor from the mentors table.

1:Many Relationship

For a one is to many relationship, suppose we have two tables parents and children and each parent hasMany children, then in the children table we will have a field called 'parent_id' which will store the id of the parent from the parents table.

Many:Many Relationship

For a many is to many relationship, suppose we have two tables students and teachers and each student hasManyAndBelongsToMany teachers, then we create a new table called students_teachers with three fields: id, student_id and teacher_id. The naming convention for this table is alphabetical. i.e. if our tables are cars and animals, then the table should be named animals_cars and not cars_animals.

Now once we have created our database as per these conventions, we must tell our framework about their existence. Let us have a look at models/product.php.

```
class Product extends VanillaModel {
    var $hasOne = array('Category' => 'Category');
    var $hasManyAndBelongsToMany = array('Tag' => 'Tag');
}
```

The first Category is the alias and the second Category is the actual model. In most cases both will be the same. Let us consider the models/category.php where they are not.

```
<?php

class Category extends VanillaModel {
        var $hasMany = array('Product' => 'Product');
        var $hasOne = array('Parent' => 'Category');

}
```

Here each category has a parent category, thus our alias is *Parent* while model is *Category*. Thus we will have a field called parent_id in the categories table. To clearly understand these relationships, I suggest you create a couple of tables and test them out for yourself. In order to see the output, use code similar to the following in your controller.

```
function view() {
    $this->Category->id = 1;
    $this->Category->showHasOne();
    $this->Category->showHasMany();
    $this->Category->showHMABTM();
    $data = $this->Category->search();
    print_r($data);
}
```

Now let us try and understand the search() function. If there are no relationships, then the function simply does a *select * from tableName* (tableName is same as controllerName). We can influence this statement, by using the following commands:

where('fieldName','value') => Appends WHERE 'fieldName' = 'value'
like('fieldName','value') => Appends WHERE 'fieldName' LIKE '%value%'
setPage('pageNumber') => Enables pagination and display only results for the set page number
setLimit('fieldName','value') => Allows you to modify the number of results per page if pageNumber is set. Its default value is the one set in config.php.
orderBy('fieldName','Order') => Appends ORDER BY 'fieldName' ASC/DESC
id = X => Will display only a single result of the row matching the id

Now let us consider when showHasOne() function has be called, then for each hasOne relationship, a LEFT JOIN is done (see line 91-99).

Now if showHasMany() function has been called, then for each result returned by the above query and for each hasMany relationship, it will find all those records in the second table which match the current result's id (see line 150). Then it will push all those results in the same array. For example, if teachers hasMany students, then $this->Teacher->showHasMany() will search for *teacher_id* in the *students* table.

Finally if showHMABTM() function has been called, then for each result returned by the first query and for each hasManyAndBelongsToMany relationship, it will find all those records which match the current result's id (see line 200-201). For example, if teachers hasManyAndBelongsToMany students, then $this->Teacher->showHMABTM() will search for *teacher_id* in *students_teachers* table.

On line 236, if *id* is set, then it will return a single result (and not an array), else it will return an array. The function then calls the clear() function which resets all the variables (line 368-380).

Now let us consider an example to enable pagination on products. The code in the controllers/productscontroller.php should look something similar to the following.

```
function page ($pageNumber = 1) {
    $this->Product->setPage($pageNumber);
    $this->Product->setLimit('10');
    $products = $this->Product->search();
    $totalPages = $this->Product->totalPages();
    $this->set('totalPages',$totalPages);
    $this->set('products',$products);
    $this->set('currentPageNumber',$pageNumber);
}
```

Now our corresponding view i.e. views/products/page.php will be something like below.

```
<?php foreach ($products as $product):?>
<div>
<?php echo $product['Product']['name']?>
</div>
<?php endforeach?>

<?php for ($i = 1; $i <= $totalPages; $i++):?>
<div>
<?php if ($i == $currentPageNumber):?>
<?php echo $currentPageNumber?>
<?php else: ?>
<?php echo $html->link($i,'products/page/'.$i)?>
<?php endif?>
</div>
<?php endfor?>
```

Thus with a few lines of code we have enabled pagination on our products page with pretty URLs! (/products/page/1, products/page/2 …)

If you look at the totalPages() function on line 384-397, you will see that it takes the existing query and strips of the LIMIT condition and returns the count. This count/limit gives us the totalPages.

Now suppose that we are in the categories controller and we want to implement a query on the products table. One way to implement this is using a custom query i.e. $this->Category->custom('select * from products where ….');

Alternatively, we can use the performAction function (line 49-57 in shared.php) to call an action of another controller. An example call in categoriescontroller.php would be something like below.

```
function view($categoryId = null, $categoryName = null) {
  $categories = performAction('products','findProducts',array($categoryId,$categoryName));
}
```

And the corresponding code in productscontroller.php should be as below.

```
function findProducts ($categoryId = null, $categoryName = null) {
  $this->Product->where('category_id',$categoryId);
  $this->Product->orderBy('name');
  return $this->Product->search();
}
```

The above more or less sums up all the functionality of the SQLQuery class. You can easily extend this as per your requirements e.g. to cache results, add conditions to hasMany, hasOne, hasMABTM queries also etc.

I have also laid the foundation for implementing user registration functionality by specifying a beforeAction and afterAction function which will be executed for each controller action. This will enable us to call a function which checks whether the user cookie is set and is a valid user.

One more feature that I have implemented is to have an administration centre. For this purpose we create a dummy model called models/admin.php and set a variable called $abstract = true. This will tell our VanillaModel to not look for a corresponding table in the database. As stated before I have created a routing for admin/X/Y as admin/X_Y. Thus we can have URLs like admin/categories/delete/15 which will actually call the categories_delete(15) function in the controllers/admincontroller.php file.

Where do we go from here?

I hope this tutorial is not too complicated. I have tried to break it down as much as possible. A couple of functions that we need to implement include -> redirectAction which will be similar to performAction, but instead redirect. performAction can only return results. redirectAction will perform only the redirected action and display that corresponding view only. This will be helpful during user authentication. If the user is not logged in, then we call a function like -> redirectAction('user','login',array($returnUrl));

Various classes are in their infancy like the cache class, HTML class etc. These classes can be expanded to add more functionality.

More scripts can be added like dbimport, clearcache etc. can be added. Currently there is only one script dbexport.php which dumps the entire database. Classes for managing sessions, cookies etc. also need to be developed.