Posted on 26 January, 2013 by Masud Alam

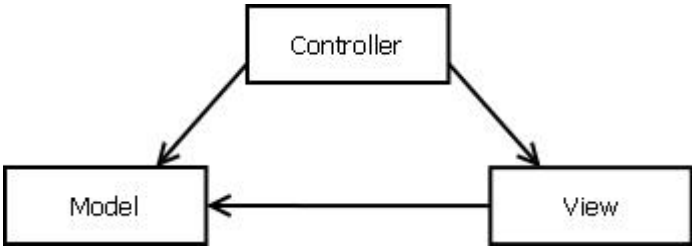
CRUD with PHP OOP and MVC Design Pattern

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.

- **Model** – represents applications behavior and data. It is also commonly known as the domain. The domain represents the problem you are trying to solve.
- **View** – represents the presentation. It can query the model about its state and generate the presentation of the model objects.
- **Controller** – represents applications workflow, it processes the request parameters and decides what to do based on them. This, usually, involves manipulating the model and displaying the data in the selected view.



As I think about the MVC pattern I can see only one negative aspect and several positive aspects. The negative is that the MVC pattern introduces complexity into the project which can be a burden for simple applications, but as your application grows this negative aspect is over weighted by positive aspects. I am going to describe few positive aspects I can think of.

MVC separates the model from the view, that is, the data from its representation. The separation of a model from its representation is one of the basic rules of a good software design. When you separate a model from its representation you can easily add many different representations of the same model. For example, a web application typically has HTML representation, used by web browsers, and JSON representation used by JavaScript AJAX clients.

A distinctive MVC role allows us to better distribute manpower on a big project. For example, domain and database experts can work on a model while web designers work on a view. Because each part is developed by specialized developers, an application can be of better quality. This also affects development time, projects can be build quicker because specialized

developers can work simultaneously, each in their area of expertise, without affecting other developers work (to much).

Maintenance and upgrades of a complex application are easier because a developer can look at the application as a series of "modules" each consisting of Model, View and the Controller part. A developer can modify one "module" and does not need to worry that the changes he introduced will affect other parts of the system (I believe that this is called separation of concerns). Also, when adding new functionality to the application, a developer can simply create a new "module".

Example application

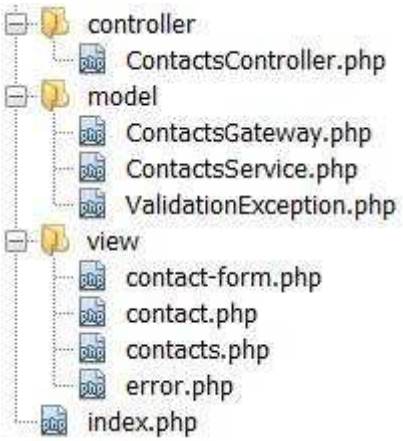
For this tutorial I created simple contacts manager which supports basic CRUD operations. User can add a new contact, delete it, display a contact detail and display list of all contacts. I will refer to these operations as actions. Therefore, this application has four different actions: *add contact, delete contact, show contact* and *list contacts*. You can see the result of the *list contacts* action in the screenshot below:

Add new contact

Name	Phone	Email	Address	
Liton Saha	011876543587	liton@sslwireless.com	Dhaka,Bangladesh	delete
Masud Alam	01722817591	masud.eden@gmail.com	Dhaka, Bangladesh	delete
Razi Uddin Razu	01877666666	razi.uddin@gmail.com	Comilla, Bangladesh	delete
Sahid	01887686687	sahid.hassan@gmail.com	Blab Blab Blab	delete
Sohel Alam	0122298765	sohel.alam@gmail.com	Kazir Haat, Noakhali, Bangladesh	delete

The implementation

The implementation consists of the *index.php* file and several files placed in the *model, view* and *controller* directories:



The ***index.php*** script is central access point, all requests go through it.

The controller is defined in the *controller directory:*

1. ***ContactsController.php* file.**

Application views are defined in the *view* directory:

1. *contact-form.php* is responsible for displaying “Add new contact” form to the user
2. *contact.php* is responsible for displaying contact details, *contacts.php* is responsible for displaying the contacts list
3. *error.php* is responsible for diplaying errors.

The model is defined in the *model* directory:

It consists of three parts:

1. ***ContactsGateway*.php** is a table data gateway to the database table I’ll show you later, the
2. ***ContactsService*.php** object defines the model API that is used by the controller.
3. ***ValidationException*.php** is an exception thrown from the model and catched by the controller in case of any validation errors. Using the *ValidationException* the model can alert the controller about validation errors and the controller can pass them to the view so they can be displayed.

The model

Before I explain source code, you must know something about the model. The model has a single *entity* – Contact which is persisted in the *contacts* table. The Contact has no behavior so I used SQL table structure to define it:

```
1 CREATE TABLE `contacts` (
2
3     `id` int(11) NOT NULL AUTO_INCREMENT,
4
5     `name` varchar(128) NOT NULL,
6
7     `phone` varchar(64) DEFAULT NULL,
8
9      `email` varchar(255) DEFAULT NULL,
10
11     `address` varchar(255) DEFAULT NULL,
12
13     PRIMARY KEY (`id`)
14
15 )
```

There is no class in the model that represents the Contact entity, instead I used standard PHP objects automatically created from the database record (I know, it is not very OO but it was quick).

What can we do with that incredible model? Here is the public interface of the *ContactsService* class where you can see all the features of the model:

```
1 class ContactsService {
2
3 private $contactsGateway  = NULL;
4
5 private function openDb() {
6 if (!mysql_connect("localhost", "root", "")) {
7 throw new Exception("Connection to the database server failed!");
8 }
9 if (!mysql_select_db("mvc-crud")) {
10                 throw new Exception("No mvc-crud database found on
                    database server.");
11 }
12 }
13
14 private function closeDb() {
15 mysql_close();
16 }
17
18 public function __construct() {
19 $this->contactsGateway = new ContactsGateway();
20 }
21
22 public function getAllContacts($order) {
23 try {
24 $this->openDb();
25 $res = $this->contactsGateway->selectAll($order);
26 $this->closeDb();
27 return $res;
28 } catch (Exception $e) {
29 $this->closeDb();
30 throw $e;
31 }
32 }
33
34 public function getContact($id) {
35 try {
36 $this->openDb();
37 $res = $this->contactsGateway->selectById($id);
38 $this->closeDb();
39 return $res;
40 } catch (Exception $e) {
41 $this->closeDb();
42 throw $e;
43 }
44 return $this->contactsGateway->find($id);
```

```
45 }
46
47 private function validateContactParams( $name, $phone, $email, $address )
   {
48 $errors = array();
49 if ( !isset($name) || empty($name) ) {
50 $errors[] = 'Name is required';
51 }
52 if ( empty($errors) ) {
53 return;
54 }
55 throw new ValidationException($errors);
56 }
57
58 public function createNewContact( $name, $phone, $email, $address ) {
59 try {
60 $this->openDb();
61 $this->validateContactParams($name, $phone, $email, $address);
62 $res = $this->contactsGateway->insert($name, $phone, $email, $address);
63 $this->closeDb();
64 return $res;
65 } catch (Exception $e) {
66 $this->closeDb();
67 throw $e;
68 }
69 }
70
71 public function deleteContact( $id ) {
72 try {
73 $this->openDb();
74 $res = $this->contactsGateway->delete($id);
75 $this->closeDb();
76 } catch (Exception $e) {
77 $this->closeDb();
78 throw $e;
79 }
80 }
81
82 }
```

ContactsService object does not work with the database directly; instead it uses the *ContactsGateway* object which in return issues queries to the database. This technique is called *Table Data Gateway*.

The source

First let’s look at the *index.php* source:

```
1 <?php
2
3 require_once 'controller/ContactsController.php';
4
5 $controller = new ContactsController();
6
7 $controller->handleRequest();
8
9 ?>
```

This script has simple role, it instantiates the controller object and hands it control over the application via the *handleRequest* method.

All requests must go through the *index.php* script because MVC requires that all requests are handled by the controller which is called from the *index.php* script. Web MVC applications usually redirects all requests to go through the *index.php* which can be done in server configuration.

Let’s look at the *handleRequest* method of the controller.

```
1 class ContactsController {
2
3 ...
4
5     public function handleRequest() {
6
7        $op = isset($_GET['op'])?$_GET['op']:NULL;
8
9         try {
10
11            if ( !$op || $op == 'list' ) {
12
13                $this->listContacts();
14
15            } elseif ( $op == 'new' ) {
16
17                $this->saveContact();
18
19            } elseif ( $op == 'delete' ) {
20
21                $this->deleteContact();
22
```

```
23             } elseif ( $op == 'show' ) {
24
25                 $this->showContact();
26
27             } else {
28
29                 $this->showError("Page not found", "Page for operation
".$op." was not found!");
30
31             }
32
33         } catch ( Exception $e ) {
34
35             $this->showError("Application error", $e->getMessage());
36
37         }
38
39     }
40
41 ...
42
43 }
```

The *handleRequest* method acts as a dispatcher for the actions. The method decides which action should it invoke based on a value of the HTTP GET *"op"* parameter and invokes the method that implements the action. If any exception is thrown from the action methods the *handleRequest* method catches them and prints the error message.

Now, let's look at the action methods:

```
1 class ContactsController {
2
3 private $contactsService = NULL;
4
5 public function __construct() {
6 $this->contactsService = new ContactsService();
7 }
8
9  public function redirect($location) {
10 header('Location: '.$location);
11 }
12
13 public function handleRequest() {
14 $op = isset($_GET['op'])?$_GET['op']:NULL;
```

```
15 try {
16 if ( !$op || $op == 'list' ) {
17 $this->listContacts();
18 } elseif ( $op == 'new' ) {
19 $this->saveContact();
20 } elseif ( $op == 'delete' ) {
21 $this->deleteContact();
22 } elseif ( $op == 'show' ) {
23 $this->showContact();
24 } else {
25 $this->showError("Page not found", "Page for operation ".$op." was not found!");
26 }
27 } catch ( Exception $e ) {
28 // some unknown Exception got through here, use application error page to display it
29 $this->showError("Application error", $e->getMessage());
30 }
31 }
32
33 public function listContacts() {
34 $orderby = isset($_GET['orderby'])?$_GET['orderby']:NULL;
35 $contacts = $this->contactsService->getAllContacts($orderby);
36 include 'view/contacts.php';
37 }
38
39 public function saveContact() {
40
41 $title = 'Add new contact';
42
43 $name = '';
44 $phone = '';
45 $email = '';
46 $address = '';
47
48 $errors = array();
49
50 if ( isset($_POST['form-submitted']) ) {
51
52 $name    = isset($_POST['name']) ?   $_POST['name']  :NULL;
53 $phone   = isset($_POST['phone'])?   $_POST['phone'] :NULL;
54 $email   = isset($_POST['email'])?   $_POST['email'] :NULL;
55 $address  = isset($_POST['address'])? $_POST['address']:NULL;
56
```

```
57 try {
58 $this->contactsService->createNewContact($name, $phone, $email,
   $address);
59 $this->redirect('index.php');
60 return;
61 } catch (ValidationException $e) {
62 $errors = $e->getErrors();
63 }
64 }
65
66 include 'view/contact-form.php';
67 }
68
69 public function deleteContact() {
70 $id = isset($_GET['id'])?$_GET['id']:NULL;
71 if ( !$id ) {
72 throw new Exception('Internal error.');
73 }
74
75 $this->contactsService->deleteContact($id);
76
77 $this->redirect('index.php');
78 }
79
80 public function showContact() {
81 $id = isset($_GET['id'])?$_GET['id']:NULL;
82 if ( !$id ) {
83 throw new Exception('Internal error.');
84 }
85 $contact = $this->contactsService->getContact($id);
86
87 include 'view/contact.php';
88 }
89
90 public function showError($title, $message) {
91 include 'view/error.php';
92 }
93
94 }
```

First, we have the *listContacts* method which has a simple workflow, it reads a parameter required for sorting the contacts, gets sorted contacts from the model, stores it in the *contacts* variable and, finally, includes the view.

Second, we have the *deleteContact* method which reads an id of the contact and tells the model to delete it. Finally, it redirects the user back to the *index.php* script which in return invokes the *list contacts* action. The *delete contact* action can not work without the *id* parameter, so, the method will throw an exception if the id of a contact is not set.

Third method is the *showContact* method which is similar to the *deleteContact* method so I will not waste space explaining it.

Finally, the *saveContact* method displays the "Add new contact" form, or, processes the data passed from the form if it was submitted. If any *ValidationException* occurred in the model, errors are collected from the exception and passed to the view. Ideally the view should display it (and indeed it does, as you will see soon).

Now let's look at some views, first I want to show you the *contacts.php* view which is straightforward:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
5 <title>Contacts</title>
6 <style type="text/css">
7 table.contacts {
8 width: 100%;
9 }
10
11 table.contacts thead {
12 background-color: #eee;
13 text-align: left;
14 }
15
16 table.contacts thead th {
17 border: solid 1px #fff;
18 padding: 3px;
19 }
20
21 table.contacts tbody td {
22 border: solid 1px #eee;
23 padding: 3px;
24 }
25
26 a, a:hover, a:active, a:visited {
27 color: blue;
28 text-decoration: underline;
29 }
30 </style>
31 </head>
```

```
32 <body>
33 <div><a href="index.php?op=new">Add new contact</a></div>
34 <table border="0" cellpadding="0" cellspacing="0">
35 <thead>
36 <tr>
37 <th><a href="?orderby=name">Name</a></th>
38 <th><a href="?orderby=phone">Phone</a></th>
39 <th><a href="?orderby=email">Email</a></th>
40 <th><a href="?orderby=address">Address</a></th>
41 <th> </th>
42 </tr>
43 </thead>
44 <tbody>
45 <?php foreach ($contacts as $contact): ?>
46 <tr>
47 <td><a href="index.php?op=show&id=<?php print $contact->id; ?>"><?php
   print htmlentities($contact->name); ?></a></td>
48 <td><?php print htmlentities($contact->phone); ?></td>
49 <td><?php print htmlentities($contact->email); ?></td>
50 <td><?php print htmlentities($contact->address); ?></td>
51 <td><a href="index.php?op=delete&id=<?php print $contact->id;
   ?>">delete</a></td>
52 </tr>
53 <?php endforeach; ?>
54 </tbody>
55 </table>
56 </body>
57 </html>
```

The *contacts.php* view, which is used by the *list contacts* action, requires the *contacts* variable to be filled with contact objects. The variable is filled in the *listContacts* method and passed to the view using the PHP scoping rules and then the view uses the data from it to display the contacts as an HTML table.

Scripts like the *contacts.php* script are called templates. With a template developer defines a fixed structure that is filled with variable values from the application (at run-time) and then presented to the user. Templates contain only presentation logic, because of this, they are understandable by non-programmers (designers, …) which makes them quite usefull and frequently used in web applications.

Now let's take a look at more complex view, the *contact-form.php* view:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
5 <title>
6 <?php print htmlentities($title) ?>
```

```
7 </title>
8 </head>
9  <body>
10 <?php
11 if ( $errors ) {
12 print '<ul>';
13 foreach ( $errors as $field => $error ) {
14 print '<li>'.htmlentities($error).'</li>';
15 }
16 print '</ul>';
17 }
18 ?>
19 <form method="POST" action="">
20 <label for="name">Name:</label><br/>
21 <input type="text" name="name" value="<?php print htmlentities($name)
?>"/>
22 <br/>
23
24 <label for="phone">Phone:</label><br/>
25 <input type="text" name="phone" value="<?php print htmlentities($phone)
?>"/>
26 <br/>
27 <label for="email">Email:</label><br/>
28 <input type="text" name="email" value="<?php print htmlentities($email)
?>" />
29 <br/>
30 <label for="address">Address:</label><br/>
31 <textarea name="address"><?php print htmlentities($address)
?></textarea>
32 <br/>
33 <input type="hidden" name="form-submitted" value="1" />
34 <input type="submit" value="Submit" />
35 </form>
36
37 </body>
38 </html>
```

This view, also an template, is used by the *add contact* action and it displays the “Add new contact” form to the user. All variables required by the template are filled in the *saveContact* action method including the *errors* variable which, if set, is used by the template to display errors that occurred in the model while trying to create a new contact so the user can fix invalid data causing them.

This template shows us usefulness of controller-view separation, because, without it we would put controller logic from the *saveContact* method into the *contact-form.php* script. I say script because if a template contains any controller logic in it, it is no more a template.

Conclusion

I hope you understood the MVC and that you i it, try to implement some sample application using your own simple MVC framework, but do not reinvent the wheel because there are many excellent MVC frameworks out there, for any programming language. Just google it up.

As always if you have any questions or suggestions don’t hesitate to write them in the comment form below.

Download Complete Project: