Getting Started with the Zend Framework

By Rob Allen, www.akrabat.com
Document Revision 1.0
Copyright © 2006

The Zend Framework is now at version 0.1.5, so it's time for me to have a go at writing a getting started guide! This tutorial is intended to give a very basic introduction to using the Zend Framework to write a very basic database driven application.

Model-View-Controller Architecture

The traditional way to build a PHP application is to do something like the following:

```
<?php
include "common-libs.php";
include "config.php";
mysql_connect($hostname, $username, $password);
mysql_select_db($database);
?>

<?php include "header.php"; ?>
<h1>Home Page</h1>

<?php
$sql = "SELECT * FROM news";
$result = mysql_query($sql);
?>
<table>
<?php
while ($row = mysql_fetch_assoc($result)) {
?>
<tr>
    <td><?php echo $row['date_created']; ?></td>
    <td><?php echo $row['title']; ?></td>
</tr>
<?php
}
?>
</table>
<?php include "footer.php"; ?>
```

Over the lifetime of an application this type of application becomes un-maintainable as the client keeps requesting changes which are hacked into the code-base in various places.

One method of improving the maintainability of the application is to separate out the code on the page into three distinct parts (and usually separate files):

Model	The model part the application is the part that is concerned with the specifics of the data to be displayed. In the above example code it is the concept of "news". Thus the model is generally concerned about the "business" logic part of the application and tends to load and save to databases.
View	The view consists of bits of the application that are concerned with the display to the user. Usually, this is the HTML.
Controller	The controller ties together the specifics of the model and the view to ensure that the correct data is displayed on the page.

The Zend Framework uses the Model-View-Controller (MVC) architecture. This is used to separate out the different parts of you application to make development and maintenance easier.

Requirements

The Zend Framework has the following requirements:

- PHP 5.1.4 (or higher)
- Apache web server supporting mod_rewrite

Getting the Framework

The Zend Framework can be downloaded from http://framework.zend.com/download in either .zip or .tar.gz format. At the time of writing, version 0.1.5 is the current version.

Directory Structure

Whilst the Zend Framework doesn't mandate a directory structure, the manual recommends a common directory structure. This structure assumes that you have complete control over your Apache configuration, however we want to make life a little easier, so will use a modification.

Start by creating a directory in the web server's root directory called zf-tutorial. This means that the url to get to the application will be http://localhost/zf-tutorial.

Create the following subdirectories to hold the application's files:

```
zf-tutorial/
    /application
           /controllers
           /models
           /views
    /library
    /public
           /images
           /scripts
           /styles
```

As you can see, we have separate directories for the model, view and controller files of our application. Supporting images, scripts and CSS files are stored in separate folders under the public directory. The downloaded Zend Framework files will be placed in the library folder. If we need to use any other libraries, they can also be placed here.

Extract the downloaded archive file, ZendFramework-0.1.5.zip in my case, to a temporary directory. All the files in the archive are placed into a subdirectory called `ZendFramework-0.1.5`. Copy the contents of `ZendFramework-0.1.5/library` into zf-tutorial/library. Your `zf-tutorial/library` should now contain a sub-directory called `Zend` and a file called `Zend.php`.

Bootstrapping

The Zend Framework's controller, Zend_Controller is designed to support websites with clean urls. To achieve this, all requests need to go through a single index.php file, known as the bootstrapper. This provides us with a central point for all pages of the application and ensures that the environment is set up correctly for running the application. We achieve this using an .htaccess file in the `zf-tutorial` root directory:

zf-tutorial/.htaccess

```
RewriteEngine on
RewriteRule .* index.php

php_flag magic_quotes_gpc off
php_flag register_globals off
```

The RewriteRule is very simple and can be interpreted as “for any url, redirect to index.php”. Note that we also set a couple of PHP ini settings. These should already be set correctly, but we want to make sure!

However, requests for images, JavaScript files and CSS files should not be redirected to our bootstrapper. By keeping all these files within the `pubic` subdirectory, we can easily configure Apache to serve these files directly with another .htaccess file in `zf-tutorial/public`:

zf-tutorial/public/.htaccess
```
RewriteEngine off
```

We also would like to protect our application and library directories so a couple more .htaccess files are required:

zf-tutorial/application/.htaccess
```
deny from all
```

zf-tutorial/library/.htaccess
```
deny from all
```

Note that for .htaccess files be used by Apache, the configuration directive AllowOverride must be set to All within your httpd.conf file. The idea presented here of using multiple .htaccess files is from Jayson Minard’s article “Blueprint for PHP Applications: Bootstrapping (Part 2) ”. I recommend reading the entire series of articles.

The Bootstrap File: index.php

`zf-tutorial/index.php` is our bootstrap file and we will start with the following code:

zf-tutorial/index.php
```
<?php
error_reporting(E_ALL|E_STRICT);
date_default_timezone_set('Europe/London');

set_include_path('.' . PATH_SEPARATOR . './library/'
     . PATH_SEPARATOR . './application/models');
include "Zend.php";

Zend::loadClass('Zend_Controller_Front');
Zend::loadClass('Zend_Controller_RewriteRouter');

// setup controller
$route = new Zend_Controller_RewriteRouter();
$route->addRoute('edit', ':controller/:action/id/:id',
    array('controller' => 'index', 'action' => 'index'));
$controller = Zend_Controller_Front::getInstance();
$controller->setRouter($route);

// run!
$controller->run('./application/controllers');
```

Note that we do not put in the `?>` at the end of the file as it is not needed and leaving it out ca prevent some hard-to-debug errors.

Let’s go through this file.

```
error_reporting(E_ALL|E_STRICT);
date_default_timezone_set('Europe/London');
```

These lines ensure that we will see any errors that we make (assuming you have the php ini setting `display_errors` set to on). We also set up our current time zone as required by PHP 5.1. Obviously, you should choose your own time zone.

```
set_include_path('.' . PATH_SEPARATOR . './library/'
     . PATH_SEPARATOR . './application/models');
include "Zend.php";
```

The Zend Framework is designed such that its files must be on the include path. We also place our models directory on the include path so that we can easily load our model classes later. To kick off we have to include the files `Zend.php` to gives us access to the `Zend` class which has the required static functions to enable us to load any other Zend Framework class

```
Zend::loadClass('Zend_Controller_Front');
Zend::loadClass('Zend_Controller_RewriteRouter');
```

Zend::loadClass loads the named class. This is achieved by converting the underscores in the class name to path separators and then adding .php to the end. Thus the class `Zend_Controller_Front` will be loaded from the file Zend/Controller/Front.php. If you follow the same naming convention for your own library classes, then you can utilise `Zend::loadClass()` to load them too.

We now set up the controller and add a route to allow for passing an id on the url. By default `Zend_Controller_RewriteRouter` only allows for /controller/action. We will need controller/action/id/{x}, so we add a new route allowing this. Note that this additional route will not be needed from version 0.1.6 onwards.

```
// setup controller
$route = new Zend_Controller_RewriteRouter();
$route->addRoute('edit', ':controller/:action/id/:id',
    array('controller' => 'index', 'action' => 'index'));
$controller = Zend_Controller_Front::getInstance();
$controller->setRouter($route);
```

Finally we get to the heart of the matter. We get an instance of the Zend Framework's controller, setup the required router and then run our application. Note that the `run()` function takes a path to our controllers directory.

```
// run!
$controller->run('./application/controllers');
```

If you go to http://localhost/zf_tutorial/ to test, you should see Fatal error similar to:

> **Fatal error**: Uncaught exception 'Zend_Exception' with message 'File "./application/controllers\IndexController.php" was not found.
> (etc.)

This is telling us that we haven't set up our application yet. Before we can do so, we had better discuss what we are going to build, so let's do that next.

The Website

We are going to build a very simple inventory system to display our CD collection. The main page will list our collection and allow us to add, edit and delete CDs. We are going to store our list in a database with a schema like this:

Fieldname	*Type*	*Null?*	*Notes*
id	Integer	No	Primary key, Autoincrement
artist	Varchar(100)	No	
title	Varchar(100)	No	

Required Pages
The following pages will be required.

Home page	This will display the list of albums and provide links to edit and delete them. Also, a link to enable adding new albums will be provided.
Add New Album	This page will provide a form for adding a new album
Edit Album	This page will provide a form for editing an album
Delete Album	This page will confirm that we want to delete an album and then delete it.

Organising the Pages
Before we set up our files, it's important to understand how the framework expects the pages to be organised. Each page of the application is known as an "action" and actions are grouped into "controllers". E.g. for a url of the format `http://localhost/zf-tutorial/news/view`, the controller is `news` and the action is `view`. This is to allow for grouping of related actions. For instance, a `news` controller might have actions of `current`, `archived` and `view`.

The Zend Framework's controller reserves a special action called `index` as a default action. That is, for a url such as `http://localhost/zf-tutorial/news/` the `index` action within the news controller will be executed. The Zend Framework's controller also reserves a default controller name should none be supplied. It should come as no surprise that this is also called `index`. Thus the url `http://localhost/zf-tutorial/` will cause the `index` action in the `index` controller to be executed.

As this is a simple tutorial, we are not going to be bothered with "complicated" things like logging in! That can wait for a separate tutorial…

As we have four pages that all apply to albums, we will group them in a single controller as four actions. We shall use the default controller and the four actions will be:

Page	*Controller*	*Action*
Home page	Index	Index
Add New Album	Index	Add
Edit Album	Index	Edit
Delete Album	Index	Delete

Nice and simple!

Setting up the Controller
We are now ready to set up our controller. In the Zend Framework, the controller is a class that must be called `{Controller name}Controller`. Note that `{Controller name}` must start with a capital letter. This class must live in a file called `{Controller name}Controller.php` within the specified controllers directory. Again `{Controller name}` must start with a capital letter and every other letter must be lowercase. Each action is a public function within the controller class that must be named `{action name}Action`. In this case `{action name}` should start with a lower case letter.

Thus our controller class is called `IndexController` which is defined in `zf-tutorial/application/controllers/IndexController.php`:

zf-tutorial/application/controllers/IndexController.php
```
<?php

class IndexController extends Zend_Controller_Action
{
    function IndexAction()
    {
        echo "<p>in IndexController::indexAction()</p>";
    }

    function addAction()
    {
        echo "<p>in IndexController::addAction()</p>";
    }

    function editAction()
    {
        echo "<p>in IndexController::editAction()</p>";
    }

    function deleteAction()
    {
        echo "<p>in IndexController::deleteAction()</p>";
    }
}
```

Initially, we've set it so that each controller prints out its name. Test this by navigating to the following urls:

URL	*Displayed text*
http://localhost/zf_tutorial/	in IndexController::indexAction()
http://localhost/zf_tutorial/index/add	in IndexController::addAction()
http://localhost/zf_tutorial/index/edit	in IndexController::editAction()
http://localhost/zf_tutorial/index/delete	in IndexController::deleteAction()

We now have a working router and the correct action is being executed for each page of our application.

It's time to build the view.

Setting up the View

The Zend Framework's view component is called, somewhat unsurprisingly, `Zend_View`. The view component will allow us to separate the code that displays the page from the code in the action functions.

The basic usage of `Zend_View` is:
```
$view = new Zend_View();
$view->setScriptPath('/path/to/view_files');
echo $view->render('view.php');
```

It can very easily be seen that if we were to put this skeleton directly into each of our action functions we will be repeating the setup code that is of no interest to the action. We would rather do the initialisation of the view somewhere else and then access our already initialised view object within each action function.

The designers of the Zend Framework foresaw this type of problem and provide a registry to enable storage and retrieval of objects. To register an object the code is:
```
Zend::register('obj', $object);
```
and to retrieve the object:
```
$object = Zend::registry('obj')
```

To integrate the view into our application we shall set up our view object and add it to the registry in our bootstrap file (zf-tutorial/index.php):

Relevant part of zf-tutorial/index.php

```
...
Zend::loadClass('Zend_Controller_Front');
Zend::loadClass('Zend_Controller_RewriteRouter');
Zend::loadClass('Zend_View');

// register the view we are going to use
$view = new Zend_View();
$view->setScriptPath('./application/views');
Zend::register('view', $view);

// setup controller
$route = new Zend_Controller_RewriteRouter();
$controller = Zend_Controller_Front::getInstance();
...
```

The changes to the file are in bold and should be self-explanatory. Note that we have to first load the `Zend_View` class using Zend::LoadClass(), before we can create an instance and set the script path to our views directory.

Having registered the view with the registry we now need to use it in the action files and move our test display code to the view files.

Change the `IndexController` as follows. Again, the changes are in bold:

zf-tutorial/application/controllers/IndexController.php

```
<?php

class IndexController extends Zend_Controller_Action
{
    function IndexAction()
    {
        $view = Zend::registry('view');
        $view->title = "My Albums";
        echo $view->render('indexIndex.tpl.php');
    }

    function addAction()
    {
        $view = Zend::registry('view');
        $view->title = "Add New Album";
        echo $view->render('indexAdd.tpl.php');
    }

    function editAction()
    {
        $view = Zend::registry('view');
        $view->title = "Edit Album";
        echo $view->render('indexEdit.tpl.php');
    }

    function deleteAction()
    {
        $view = Zend::registry('view');
        $view->title = "Delete Album";
        echo $view->render('indexDelete.tpl.php');
    }
}
```

In each function, we retrieve the view object from the registry and assign a title variable to it. We then display (render) the correct template. As should be obvious, we now need four view files to our application. These files are known as templates and, as a convention, I have named each template file after its action and used the extension .tpl.php to show that it is a template file.

zf-tutorial/application/views/indexIndex.tpl.php
```
<html>
<head>
    <title><?php echo $this->escape($this->title); ?></title>
</head>
<body>
    <h1><?php echo $this->escape($this->title); ?></h1>
</body>
</html>
```

zf-tutorial/application/views/addIndex.tpl.php
```
<html>
<head>
    <title><?php echo $this->escape($this->title); ?></title>
</head>
<body>
    <h1><?php echo $this->escape($this->title); ?></h1>
</body>
</html>
```

zf-tutorial/application/views/editIndex.tpl.php
```
<html>
<head>
    <title><?php echo $this->escape($this->title); ?></title>
</head>
<body>
    <h1><?php echo $this->escape($this->title); ?></h1>
</body>
</html>
```

zf-tutorial/application/views/deleteIndex.tpl.php
```
<html>
<head>
    <title><?php echo $this->escape($this->title); ?></title>
</head>
<body>
    <h1><?php echo $this->escape($this->title); ?></h1>
</body>
</html>
```

Testing each controller/action should display the four titles in bold.

Common HTML code

It very quickly becomes obvious that there is a lot of common HTML code in our views. We will factor out the html code that is common to all actions into a file called `site.tpl.php`. This we can use to provide the “outside” section of our page and then render the file that contains the specifics for each action from within `site.tpl.php`.

Again, our controller needs changing:

zf-tutorial/application/controllers/IndexController.php
```
<?php

class IndexController extends Zend_Controller_Action
{
    function IndexAction()
    {
        $view = Zend::registry('view');
        $view->title = "My Albums";
        $this->actionTemplate = 'indexIndex.tpl.php';
        echo $view->render('site.tpl.php');
    }

    function addAction()
    {
        $view = Zend::registry('view');
        $view->title = "Add New Album";
        $this->actionTemplate = 'indexAdd.tpl.php';
        echo $view->render('site.tpl.php');
    }

    function editAction()
    {
        $view = Zend::registry('view');
        $view->title = "Edit Album";
        $this->actionTemplate = 'indexEdit.tpl.php';
        echo $view->render('site.tpl.php');
    }

    function deleteAction()
    {
        $view = Zend::registry('view');
        $view->title = "Delete Album";
        $this->actionTemplate = 'indexDelete.tpl.php';
        echo $view->render('site.tpl.php');
    }
}
```

We have introduced a new variable for the view called actionTemplate and now render the same `site.tpl.php` template file in all cases.

The view files are as follows:

zf-tutorial/application/views/site.tpl.php
```
<html>
<head>
    <title><?php echo $this->escape($this->title); ?></title>
</head>
<body>
       <div id="content">
    <?php echo $this->render($this->actionTemplate); ?>
    </div>
</body>
</html>
```

zf-tutorial/application/views/indexIndex.tpl.php
```
<h1><?php echo $this->escape($this->title); ?></h1>
```

zf-tutorial/application/views/indexAdd.tpl.php
```
<h1><?php echo $this->escape($this->title); ?></h1>
```

zf-tutorial/application/views/indexEdit.tpl.php
```
<h1><?php echo $this->escape($this->title); ?></h1>
```

zf-tutorial/application/views/indexDelete.tpl.php
```
<h1><?php echo $this->escape($this->title); ?></h1>
```

Styling

Even though this is just a tutorial, we'll need a CSS file to make our application look a little bit presentable!

zf-tutorial/application/views/site.tpl.php

```
...
<head>
    <title><?php echo $this->escape($this->title); ?></title>
    <link rel="stylesheet" type="text/css" media="screen"
             href="/zf-tutorial/public/styles/site.css" />
</head>
...
```

zf-tutorial/public/styles/site.css

```
body,html {
    font-size:100%;
    margin: 0;
    font-family: Verdana,Arial,Helvetica,sans-serif;
    color: #000;
    background-color: #fff;
}

h1 {
    font-size:1.4em;
    color: #000080;
    background-color: transparent;
}

#content {
    width: 770px;
    margin: 0 auto;
}

label {
    width: 100px;
    display: block;
    float: left;
}

#formbutton {
    margin-left: 100px;
}
```

The Database

Now that we have separated the control of the application from the displayed view, it is time to look at the model section of our application. Remember that the model is the part that deals with the application's core purpose (the so-called "business rules") and hence, in our case, deals with the database. We will make use of the Zend Framework class `Zend_Db_Table` which is used to find, insert, update and delete rows from a database table.

Configuration

To use `Zend_Db_Table`, we need to tell it which database to use along with a username and password. As we would prefer not to hard-code this information into our application we will use a configuration file to hold this information.

The Zend Framework provides `Zend_Config` to provide flexible object oriented access to configuration files. At the moment, the configuration file can be a PHP array, an INI file or an XML file. We will use an INI file:

zf-tutorial/application/config.ini
```
[general]
db.adapter = PDO_MYSQL
db.config.host = localhost
db.config.username = rob
db.config.password = 123456
db.config.dbname = zftest
```

Obviously you should use your username, password and database name, not mine!

To use `Zend_Config` is very easy:
```
$cfg_array = Zend_Config_Ini::load('config.ini', 'section');
$config = new Zend_Config($cfg_array);
```

Note `Zend_Config_Ini` loads one section from the INI file, not every section. It supports a special keyword, `extends`, to allow loading of additional sections. `Zend_Config` also treats the “dot” in the parameter as hierarchical separators to allow for grouping of related configuration parameters. In our config.ini, the host, username, password and dbname parameters will be grouped under `$config->db->config`.

We will load our configuration file in our bootstrapper (index.php):

Relevant part of zf-tutorial/index.php
```
...
Zend::loadClass('Zend_Controller_Front');
Zend::loadClass('Zend_Controller_RewriteRouter');
Zend::loadClass('Zend_View');
Zend::loadClass('Zend_Config');
Zend::loadClass('Zend_Config_Ini');

// load configuration
$config = new Zend_Config(Zend_Config_Ini::load(
                  './application/config.ini', 'general'));
Zend::register('config', $config);

// register the view we are going to use
$view = new Zend_View();
$view->setScriptPath('./application/views');
...
```

The changes are in bold. We load the two classes we are going to use (`Zend_Config` and `Zend_Config_Ini`) and then load the ‘`general`’ section of `application/config.ini` into our `$config` object.

Note: In this tutorial, we don’t actually need to store `$config` to the registry, but it’s good practice as in a ‘real’ application you are likely to have more than just database configuration information in the INI file.

Setting up Zend_Db_Table

To use `Zend_Db_Table`, we need to tell it the database configuration information that we have just loaded. To do this we need to create an instance of `Zend_Db` and then register this with the static function `Zend_Db_Table:: setDefaultAdapter()`function. Again, we do this within the bootstrapper (additions in bold):

Relevant part of zf-tutorial/index.php

```
...
Zend::loadClass('Zend_Config');
Zend::loadClass('Zend_Config_Ini');
Zend::loadClass('Zend_Db');
Zend::loadClass('Zend_Db_Table);


// load configuration
$config = new Zend_Config(Zend_Config_Ini::load(
                         './application/config.ini', 'general'));
Zend::register('config', $config);

// setup database
$db = Zend_Db::factory($config->db->adapter,
                 $config->db->config->asArray());
Zend_Db_Table::setDefaultAdapter($db);

// register the view we are going to use
$view = new Zend_View();
$view->setScriptPath('./application/views');
Zend::register('view', $view);
...
```

Create the Table

I'm going to be using MySQL and so the SQL statement to create the database is:

```
CREATE TABLE album (
  id int(11) NOT NULL auto_increment,
  artist varchar(100) NOT NULL,
  title varchar(100) NOT NULL,
  PRIMARY KEY (id)
)
```

Run this statement in a MySQL client such as phpMyAdmin or the standard MySQL command-line client.

Insert Test Albums

We will also insert a couple of rows into the table so that we can test the retrieval functionality of the home page. I'm going to take the first two "Hot 100" CDs from Amazon.co.uk:

```
INSERT INTO album (artist, title)
VALUES
       ('Undiscovered', 'James Morrison'),
       ('Eyes Open', 'Snow Patrol');
```

The Model

`Zend_Db_Table` is an abstract class, so we have to derive our class that is specific to managing albums. By default, `Zend_Db_Table` expects that the class name is the same as the table name. Thus, our class will be called `Album` as our table name is `album`. Also, `Zend_Db_Table` assumes that your table has a primary key called `id` which is auto-incremented by the database. Both these conventions can be overridden if necessary.

We will store our Album table in the models directory:

zf-tutorial/application/models/Album

```
<?php

class Album extends Zend_Db_Table
{

}
```

Not very complicated is it?! Fortunately for us, our needs are very simple and `Zend_Db_Table` provides enough functionality directly. However if you need specific functionality to manage your model, then this is the class to put it in. Generally, the additional functions you would provide would be additional “find” type methods to enable collection of the exact data you are looking for.

Listing Albums

Now that we have set up configuration and database information, we can get onto the meat of the application and display some albums. This is done in the `IndexController` class.

Clearly every action within `IndexController` will be manipulating the `album` database using the `Album` class, so it makes sense to load the class in the constructor:

zf-tutorial/application/views/IndexController.php

```
<?php

class IndexController extends Zend_Controller_Action
{
    function __construct()
    {
        Parent::__construct();
        Zend::loadClass('Album');
    }

    function IndexAction()
    {

...
```

This is an example of using `Zend::loadClass()` to load our own classes and works because we put the models directory onto the php include path in `index.php`.

We are going to list the albums in a table within the `indexAction()`:

zf-tutorial/application/views/IndexController.php

```
...
function indexAction()
{
    $view = Zend::registry('view');
    $view->title = "My Albums";

    $album = new Album();
    $view->albums = $album->fetchAll();

    $view->actionTemplate = 'indexIndex.tpl.php';
    echo $view->render('site.tpl.php');
}
...
```

The function `Zend_Db_Table::fetchAll()` returns a `Zend_Db_Table_Rowset` which will allow us to iterate over the returned rows in the view template file:

zf-tutorial/application/views/indexIndex.tpl.php

```
<h1><?php echo $this->escape($this->title); ?></h1>
<p><a href="/zf-tutorial/index/add">Add new album</a></p>
<table>
<tr>
    <th>Title</th>
    <th>Artist</th>
    <th>&nbsp;</th>
</tr>
```

```
<?php foreach($this->albums as $album) : ?>
<tr>
    <td><?php echo $this->escape($album->title);?></td>
    <td><?php echo $this->escape($album->artist);?></td>
    <td>
      <a href="/zf-tutorial/index/edit/id/<?php echo $album->id;?>"
           >Edit</a>
      <a href="/zf-tutorial/index/delete/id/<?php echo $album->id;?>"
           >Delete</a>
    </td>
</tr>
<?php endforeach; ?>
</table>
```

http://localhost/zf-tutorial/ should now show a nice list of (two) albums.

Dealing with Post and Get Variables

In a traditional PHP application, the "magic" globals $_POST and $_GET are used to retrieve variables supplied from the user. The problem is that it is very easy to forget to validate that the data supplied for a given field is of the expected type. If validation isn't performed it is possible to introduce certain classes of security issues, or just break the application. The Zend Framework provides the `Zend_Filter_Input` class to make validation of user-supplied data easier.

To use `Zend_Filter_Input`:

```
$postArray = new Zend_Filter_Input($_POST);
$username = $postArray->testName('username');
if ($username !== false) {
  // $ username is a valid name
}
```

One key thing to remember about Zend_Filter_Input is that it wipes the input array. That is, after creating a Zend_Filter_Input for $_POST, $_POST is then set to null. Thus, we will set up out filters in the index.php bootstrap file and store them in the Zend::registry so that we can access them wherever we need to.

Relevant part of zf-tutorial/index.php

```
...
Zend::loadClass('Zend_Db');
Zend::loadClass('Zend_Db_Table');
Zend::loadClass('Zend_Filter_Input');

// register the input filters
Zend::register('post', new Zend_Filter_Input($_POST));
Zend::register('get', new Zend_Filter_Input($_GET));

// load configuration
...
```

Now we can get at a post variable with the following code:

```
$post = Zend::registry('post');
$myVar = $post->testAlpha(myVar');
```

Adding New Albums

Now that we have set up the Post input filter, we can code up the functionality to add new albums. There are two bits to this part:

- Display a form for user to provide details
- Process the form submission and store to database

This is done within `addAction()`:

zf-tutorial/application/views/IndexController.php

```
...
function addAction()
{
    $view = Zend::registry('view');
    $view->title = "Add New Album";

    if (strtolower($_SERVER['REQUEST_METHOD']) == 'post') {
        $post = Zend::registry('post');
        $artist = trim($post->noTags('artist'));
        $title = trim($post->noTags('title'));

        if ($artist != '' && $title != '') {
            $data = array(
                'artist' => $artist,
                'title'  => $title,
            );
            $album = new Album();
            $album->insert($data);

            $url = '/zf-tutorial/';
            $this->_redirect($url);
            return;
        }
    }

    // set up an "empty" album
    $view->album = new stdClass();
    $view->album->artist = '';
    $view->album->title = '';

    // additional view fields required by form
    $view->action = 'add';
    $view->buttonText = 'Add';

    $view->actionTemplate = 'indexAdd.tpl.php';
    echo $view->render('site.tpl.php');
}
...
```

Notice how we check the `$_SERVER['REQUEST_METHOD']` variable to see if the form has been submitted. If it has, we retrieve the artist and title from the post array using the `noTags()` function to ensure that no html is allowed and then assuming that they have been filled in, we utilise our model class, `Album()`, to insert the information into a new row in the database table.

Finally, we set up the view ready for the form we will use in the template. Looking ahead, we can see that the edit action's form will be very similar to this one, so we will use a common template file that is called from both indexAdd.tpl.php and indexEdit.tpl.php:

The templates for adding an album are:

zf-tutorial/application/views/indexAdd.tpl.php

```
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('_indexForm.tpl.php'); ?>
```

zf-tutorial/application/views/_indexForm.tpl.php

```
<form action="/zf-tutorial/index/<?php echo $this->action; ?>"
method="post">
<div>
    <label for="artist">Artist</label>
    <input type="text" class="input-large" name="artist"
        value="<?php echo $this->escape(trim($this->album->artist));?>"/>
</div>
```

```
<div>
    <label for="title">Title</label>
    <input type="text" class="input-large" name="title"
        value="<?php echo $this->escape($this->album->title);?>"/>
</div>

<div id="formbutton">
    <input type="hidden" name="id"
        value="<?php echo $this->album->id; ?>" />
    <input type="submit" name="add"
        value="<?php echo $this->escape($this->buttonText); ?>" />
</div>
</form>
```

This is fairly simple code. As we intend to use `_indexForm.tpl.php` for the edit action as well, we have used a variable to `$this->action` rather than hard coding the action attribute. Similarly, we use a variable for the text to be displayed on the submit button.

Editing an Album

Editing an album is almost identical to adding one, so the code is very similar:

zf-tutorial/application/views/IndexController.php

```
...
function editAction()
{
    $view = Zend::registry('view');
    $view->title = "Edit Album";
    $album = new Album();

    if (strtolower($_SERVER['REQUEST_METHOD']) == 'post') {
        $post = Zend::registry('post');
        $id = $post->testInt('id');
        $artist = trim($post->noTags('artist'));
        $title = trim($post->noTags('title'));

        if ($id !== false) {
            if ($artist != '' && $title != '') {
                $data = array(
                    'artist' => $artist,
                    'title'  => $title,
                );
                $where = 'id = ' . $id;
                $album->update($data, $where);

                $url = '/zf-tutorial/';
                $this->_redirect($url);
                return;
            } else {
                $view->album = $album->find($id);
            }
        }
    } else {
        // album id should be $params['id]
        $params = $this->_action->getParams();
        $id = 0;
        if (isset($params['id'])) {
            $id = (int)$params['id'];
        }
        if ($id > 0) {
            $view->album = $album->find($id);
       }
    }

    // additional view fields required by form
```

```
        $view->action = 'edit';
        $view->buttonText = 'Update';

        $view->actionTemplate = 'indexEdit.tpl.php';
        echo $view->render('site.tpl.php');
    }
    ...
```

Note that we retrieve the id parameter from the `$params` array when we are not in “post” mode as that is how we set up the route back in the `index.php` file.

The template is:

zf-tutorial/application/views/indexEdit.tpl.php

```
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('_indexForm.tpl.php'); ?>
```

Refactor!

It shouldn’t have escaped your notice that AddAction() and EditAction() are very similar and that the add and edit templates are identical. Some refactoring is in order!

I’ve left it as an exercise for you, dear reader…

Deleting an Album

To round out our application, we need to add deletetion. We have a Delete link next to each album on our list page and the naïve approach would be to do a delete when it’s clicked. This would be wrong. Remembering our HTTP spec, we would recall that you shouldn’t do an irreversible action using GET and should use POST instead. Google’s recent accelerator beta brought this point home to many people.

We shall show a confirmation form when the user clicks delete and if they then click “yes”, we will do the deletion.

The code looks somewhat similar to the add and edit actions:

zf-tutorial/application/views/IndexController.php

```
...
function deleteAction()
{
    $view = Zend::registry('view');
    $view->title = "Delete Album";

    $album = new Album();
    if (strtolower($_SERVER['REQUEST_METHOD']) == 'post') {
        $post = Zend::registry('post');
        $id = $post->getInt('id');
        if (strtolower($post->testAlpha('del')) == 'yes' && $id > 0) {
            $where = 'id = ' . $id;
            $album->delete($where);
        }
    } else {
        // album id should be $params['id]
        $params = $this->_action->getParams();
        if (isset($params['id'])) {
            $id = (int)$params['id'];
            if ($id > 0) {
               $view->album = $album->find($id);
               $view->actionTemplate = 'indexDelete.tpl.php';
               // only render if we have an id.
               echo $view->render('site.tpl.php');
               return;
```

```
                }
            }
        }
        // redirect back to the album list in all cases unless we are
        // rendering the template
        $url = '/zf-tutorial/';
        $this->_redirect($url);
    }
    ...
```

Again, we use the same trick of checking the request method to work out if we should display the confirmation form or if we should do a deletion, via the Album() class. Just like, insert and update, the actual deletion is done via a call to `Zend_Db_Table::delete()`.

Notice that we return immediately after calling the `render()` function. This is so that we can redirect back to the album list at the end of the function. Thus if any of the various sanity checks fail, we go back to the album list without having to call `_redirect()` multiple times within the function.

The template is a simple form:

zf-tutorial/application/views/indexDelete.tpl.php

```
<h1><?php echo $this->escape($this->title); ?></h1>
<?php if ($this->album) :?>
<form action="/zf-tutorial/index/delete" method="post">
    <p>Are you sure that you want to delete
           '<?php echo $this->escape($this->album->title); ?>' by
           '<?php echo $this->escape($this->album->artist); ?>'?</p>
    <div>
        <input type="hidden" name="id"
            value="<?php echo $this->album->id; ?>" />
        <input type="submit" name="del" value="Yes" />
        <input type="submit" name="del" value="No" />
    </div>
</form>
<?php else: ?>
<p>Cannot find album.</p>
<?php endif;?>
```

Conclusion

This concludes our brief look at building a simple, but fully functional, MVC application using the Zend Framework. I hope that you found it interesting and informative. If you find anything that's wrong, please let email me at rob@akrabat.com!

This tutorial has only looked at the basics of using the framework; there are many more classes to explore! You should really go and read the manual and look at the wiki to get more insights!

Last Thoughts

Whilst developing this tutorial, the most obvious thing missing to me was a better way to do models. I can see why the ActiveRecord pattern is currently very popular! There is a proposal in the Framework wiki for a `Zend_Db_Model`. The overview for `Zend_Db_Model` is: "An object that wraps a row in a database table or view, encapsulates the database access, and allows to adds domain logic on that data". Something along these lines would be very helpful for developing applications using the Zend Framework.

Overall, the Zend Framework is shaping up quite nicely.