

Getting Started with the Zend Framework

By Rob Allen, www.akrobat.com

Document Revision 1.4.2

Copyright © 2006, 2007

This tutorial is intended to give a very basic introduction to using the Zend Framework to write a basic database driven application.

NOTE: This tutorial has been tested on versions 1.0.0 of the Zend Framework. It stands a very good chance of working with later versions, but it's unlikely to work on versions prior to version 1.0.0.

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 of 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 your application to make development and maintenance easier.

Requirements

The Zend Framework has the following requirements:

- PHP 5.1.4 (or higher)
- A web server supporting `mod_rewrite` functionality. This tutorial assumes Apache.

Getting the Framework

The Zend Framework can be downloaded from <http://framework.zend.com/download/stable> in either `.zip` or `.tar.gz` format.

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  
      /filters  
      /helpers  
      /scripts  
  /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-1.0.0.zip` in my case, to a temporary directory. All the files in the archive are placed into a subdirectory called `ZendFramework-1.0.0`. Copy the contents of the subdirectory `library/Zend` into `zf-tutorial/library/`. Your `zf-tutorial/library` should now contain a sub-directory called `Zend`.

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, use index.php instead”.

We also set a couple of PHP ini settings for security and sanity. These should already be set correctly, but we want to make sure! Note that the php_flag settings in .htaccess only work if you are using mod_php. If you use CGI/FastCGI, then you need to make sure that your php.ini is correct.

However, requests for images, JavaScript files and CSS files should not be redirected to our bootstrapper. By keeping all these files within the `public` 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
```

Whilst not strictly necessary without current rewrite rules, we can add a couple more .htaccess files to ensure that our application and library directories are protected:

zf-tutorial/application/.htaccess

```
deny from all
```

zf-tutorial/library/.htaccess

```
deny from all
```

Note that for .htaccess files to 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 both 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/'
    . PATH_SEPARATOR . get_include_path());
include "Zend/Loader.php";

Zend_Loader::loadClass('Zend_Controller_Front');

// setup controller
$frontController = Zend_Controller_Front::getInstance();
$frontController->throwExceptions(true);
$frontController->setControllerDirectory('./application/controllers');

// run!
$frontController->dispatch();
```

Note that we do not put in the `?>` at the end of the file as it is not needed and leaving it out can prevent some hard-to-debug errors when redirecting via the `header()` function if additional whitespace occurs after the `?>`.

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/'
    . PATH_SEPARATOR . get_include_path());
include "Zend/Loader.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 file `Zend/Loader.php` to give us access to the `Zend_Loader` class which has the required static functions to enable us to load any other Zend Framework class

```
Zend_Loader::loadClass('Zend_Controller_Front');
```

`Zend_Loader::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_Loader::loadClass()` to load them too. The first class we need, is the front controller.

The front controller uses a router class to map the requested URL to the correct PHP function to be used for displaying the page. In order for the router to operate, it needs to work out which part of the URL is the path to our `index.php` so that it can look at the URI elements after that point. This is done by the Request object. It does a pretty good job of auto-detecting the correct base URL, but if it doesn't work for your set up, then you can override it using the function `$frontController->setBaseUrl()`.

We need to configure the front controller so that it knows which directory to find our controllers.

```
$frontController = Zend_Controller_Front::getInstance();
$frontController->setControllerDirectory('./application/controllers');
$frontController->throwExceptions(true);
```

As this is a tutorial and we are running on a test system, I've decided to instruct the front controller to throw all exceptions that occur. By default, the front controller will catch them for us and store them in the `_exceptions` property of the "Response" object that it creates. The response object holds all information about the response to the requested URL. This includes HTTP header, page content and exceptions. The front controller will automatically send the headers and display the page content just before it completes its work

This can be quite confusing for people new to the Zend Framework, so it is easier to just re-throw so that the exceptions are easily visible. Of course, on a production server, you shouldn't be displaying errors to the user anyway!

Finally we get to the heart of the matter and we run our application:

```
// run!
$frontController->dispatch();
```

If you go to <http://localhost/zf-tutorial/> to test, you should fatal error similar to:

Fatal error: Uncaught exception 'Zend_Controller_Dispatcher_Exception' with message 'Invalid controller specified (index)' in...

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:

| <i>Fieldname</i> | <i>Type</i> | <i>Null?</i> | <i>Notes</i> |
|------------------|--------------|--------------|----------------------------|
| 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 MVC system also supports modules for grouping controllers together, but this application isn't big enough to bother with them!

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:

| <i>Page</i> | <i>Controller</i> | <i>Action</i> |
|---------------|-------------------|---------------|
| 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 action prints out its name. Test this by navigating to the following URLs:

| <i>URL</i> | <i>Displayed text</i> |
|--|---|
| <code>http://localhost/zf-tutorial/</code> | <code>in IndexController::indexAction()</code> |
| <code>http://localhost/zf-tutorial/index/add</code> | <code>in IndexController::addAction()</code> |
| <code>http://localhost/zf-tutorial/index/edit</code> | <code>in IndexController::editAction()</code> |
| <code>http://localhost/zf-tutorial/index/delete</code> | <code>in IndexController::deleteAction()</code> |

We now have a working router and the correct action is being executed for each page of our application. If this does not work for you, check out the *Troubleshooting* section towards the end of this tutorial to see if that helps.

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 the solution is built into an "action helper" for us. `Zend_Controller_Action_Helper_ViewRenderer` takes care of initialising a view property (`$this->view`) for us to use and will render a view script too. For the rendering, it sets up the `Zend_View` object to look in `views/scripts/{controller name}` for the view scripts to be rendered and will (by default, at least) render the script that is named after the action with the extension `phtml`. That is, the view script rendered is `views/scripts/{controller name}/{action_name}.phtml` and the rendered contents are appended to the Response object's body. The response object is used to collate together all HTTP headers, body content and exceptions generated as a result of using the MVC system. The front controller then automatically sends the headers followed by the body content at the end of the dispatch.

To integrate the view into our application all we need to do is create some view files with test display code. No changes to the controller are required.

The changes to the `IndexController` follow (changes in bold):

zf-tutorial/application/controllers/IndexController.php

```
<?php

class IndexController extends Zend_Controller_Action
{
    function indexAction()
    {
        $this->view->title = "My Albums";
    }

    function addAction()
    {
        $this->view->title = "Add New Album";
    }

    function editAction()
    {
        $this->view->title = "Edit Album";
    }

    function deleteAction()
    {
        $this->view->title = "Delete Album";
    }
}
```

In each function, we assign a title variable to the view property and that's it! Note though that the actual display doesn't happen at this point – it is done by the front controller right at the end of the dispatch process.

We now need to add four view files to our application. These files are known as templates and the render() method expects that each template file is named after its action and has the extension .phtml to show that it is a template file. The file must be in a subdirectory that is named after the controller, so the four files are:

zf-tutorial/application/views/scripts/index/index.phtml

```
<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/scripts/index/add.phtml

```
<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/scripts/index/edit.phtml

```
<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/scripts/index/delete.phtml

```
<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 two files: header.phtml and footer.phtml within the scripts directory. We can then use them to hold the “common” HTML and just reference from the view templates.

The new files are:

zf-tutorial/application/views/scripts/header.phtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title><?php echo $this->escape($this->title); ?></title>
</head>
```

```
<body>
<div id="content">
```

(Note that we have corrected the HTML so we are now actually compliant too!)

zf-tutorial/application/views/scripts/footer.phtml

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

Again, our views need changing:

zf-tutorial/application/views/scripts/index/index.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('footer.phtml'); ?>
```

zf-tutorial/application/views/scripts/index/add.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('footer.phtml'); ?>
```

zf-tutorial/application/views/scripts/index/edit.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('footer.phtml'); ?>
```

zf-tutorial/application/views/scripts/index/delete.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('footer.phtml'); ?>
```

Styling

Even though this is just a tutorial, we'll need a CSS file to make our application look a little bit presentable! This causes a minor problem in that we don't actually know how to reference the CSS file because the URL doesn't point to the correct root directory. To solve this, we use the `getBaseUrl()` function that is part of the request and pass it to the view. This provides us with the bit of the URL that we don't know.

We use the `IndexController::init()` function for this code as `init()` is called by the constructor and so the property will be available in all actions.

zf-tutorial/application/controllers/IndexController.php

```
...
class IndexController extends Zend_Controller_Action
{
    function init()
    {
        $this->view->baseUrl = $this->_request->getBaseUrl();
    }

    function indexAction()
    {
        ...
    }
}
```

We need to add the CSS file to the `<head>` section of the `header.phtml`:

zf-tutorial/application/views/scripts/header.phtml

```
...
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title><?php echo $this->escape($this->title); ?></title>
    <link rel="stylesheet" type="text/css" media="screen"
```

```
                href="<?php echo $this->baseUrl;?>/public/styles/site.css" />
</head>
...
```

Finally, we need some CSS styles:

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: #800000;
    background-color: transparent;
}

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

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

#formbutton {
    margin-left: 100px;
}

a {
    color: #800000;
}
```

This should make it look slightly prettier!

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. The configuration file can either 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:

```
$config = new Zend_Config_Ini('config.ini', 'section');
```

Note in this case, `Zend_Config_Ini` loads one section from the INI file, not every section (though every section can be loaded if you wanted to). It supports a notation in the section name to allow loading of additional sections. `Zend_Config_Ini` 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_Loader::loadClass('Zend_Controller_Front');
Zend_Loader::loadClass('Zend_Config_Ini');
Zend_Loader::loadClass('Zend_Registry');

// load configuration
$config = new Zend_Config_Ini('./application/config.ini', 'general');
$registry = Zend_Registry::getInstance();
$registry->set('config', $config);

// setup controller
...

```

The changes are in bold. We load the classes we are going to use (`Zend_Config_Ini` and `Zend_Registry`) and then load the ‘general’ section of `application/config.ini` into our `$config` object. Finally we assign the `$config` object to the registry so that it can be retrieved elsewhere in the application.

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. Also, be aware that the registry is a bit like a global and causes dependencies between objects that shouldn’t depend on each other, if you aren’t careful.

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()`. Again, we do this within the bootstrapper (additions in bold):

Relevant part of zf-tutorial/index.php

```
...
Zend_Loader::loadClass('Zend_Controller_Front');
Zend_Loader::loadClass('Zend_Config_Ini');
Zend_Loader::loadClass('Zend_Registry');
Zend_Loader::loadClass('Zend_Db');
Zend_Loader::loadClass('Zend_Db_Table');

// load configuration
$config = new Zend_Config_Ini('./application/config.ini', 'general');
$registry = Zend_Registry::getInstance();
$registry->set('config', $config);

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

// setup controller
...
```

Create the Table

I'm going to be using MySQL and so the SQL statement to create the table 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
    ('James Morrison', 'Undiscovered'),
    ('Snow Patrol', 'Eyes Open');
```

The Model

`Zend_Db_Table` is an abstract class, so we have to derive our class that is specific to managing albums. It doesn't matter what we call our class, but it makes sense to call it the same as the database table. Thus, our class will be called `Album` as our table name is `album`. To tell `Zend_Db_Table` the name of the table that it will manage, we have to set the protected property `$_name` to the name of the table. Also, `Zend_Db_Table` assumes that your table has a primary key called `id` which is auto-incremented by the database. The name of this field can be changed too if required.

We will store our `Album` class in the models directory:

zf-tutorial/application/models/Album.php

```
<?php

class Album extends Zend_Db_Table
{
    protected $_name = 'album';
}
```

Not very complicated is it?! Fortunately for us, our needs are very simple and `Zend_Db_Table` provides all the functionality we need itself. 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. You can also tell `Zend_Db_Table` about related tables and it can fetch related data too.

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` table using the `Album` class, so it makes sense to load the album class when the controller is instantiated. This is done in the `init()` function:

zf-tutorial/application/controllers/IndexController.php

```
...
function init()
{
    $this->view->baseUrl = $this->_request->getBaseUrl();
    Zend_Loader::loadClass('Album');
}
...
```

Note: This is an example of using `Zend_Loader::loadClass()` to load our own classes and works because we have 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/controllers/IndexController.php

```
...
function indexAction()
{
    $this->view->title = "My Albums";
    $album = new Album();
    $this->view->albums = $album->fetchAll();
}
...
```

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/scripts/index/index.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<p><a href="<?php echo $this->baseUrl; ?>/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="<?php echo $this->baseUrl; ?>/index/edit/id/<?php
```

```

        echo $album->id;?>">Edit</a>
        <a href="<?php echo $this->baseUrl; ?>/index/delete/id/<?php
        echo $album->id;?>">Delete</a>
    </td>
</tr>
<?php endforeach; ?>
</table>
<?php echo $this->render('footer.phtml'); ?>

```

<http://localhost/zf-tutorial/> (or wherever you are following along from!) should now show a nice list of (two) albums.

Adding New Albums

We can now 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/controllers/IndexController.php

```

...
function addAction()
{
    $this->view->title = "Add New Album";

    if ($this->_request->isPost()) {
        Zend_Loader::loadClass('Zend_Filter_StripTags');
        $filter = new Zend_Filter_StripTags();

        $artist = $filter->filter($this->_request->getPost('artist'));
        $artist = trim($artist);
        $title = trim($filter->filter($this->_request->getPost('title')));

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

            $this->_redirect('/');
            return;
        }
    }

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

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

```

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 `Zend_Filter_StripTags` class to ensure that no html is allowed. 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.

After we have added the album, we redirect using the controller's `_redirect()` method to go back to the root of our application.

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 (`_form.phtml`) that is called from both `add.phtml` and `edit.phtml`:

The templates for adding an album are:

zf-tutorial/application/views/scripts/index/add.phtml

```
<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('index/_form.phtml'); ?>
<?php echo $this->render('footer.phtml'); ?>
```

zf-tutorial/application/views/scripts/index/_form.phtml

```
<form action="<?php echo $this->baseUrl ?>/index/<?php
    echo $this->action; ?>" method="post">
<div>
    <label for="artist">Artist</label>
    <input type="text" name="artist"
        value="<?php echo $this->escape(trim($this->album->artist));?>" />
</div>
<div>
    <label for="title">Title</label>
    <input type="text" 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 `_form.phtml` 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/controllers/IndexController.php

```
...
function editAction()
{
    $this->view->title = "Edit Album";
    $album = new Album();

    if ($this->_request->isPost()) {
        Zend_Loader::loadClass('Zend_Filter_StripTags');
        $filter = new Zend_Filter_StripTags();

        $id = (int)$this->_request->getPost('id');
        $artist = $filter->filter($this->_request->getPost('artist'));
        $artist = trim($artist);
        $title = trim($filter->filter($this->_request->getPost('title')));

        if ($id !== false) {
            if ($artist != '' && $title != '') {
                $data = array(
                    'artist' => $artist,
                    'title' => $title,
                );
            }
        }
    }
}
```

```

    );
    $where = 'id = ' . $id;
    $album->update($data, $where);

    $this->_redirect('/');
    return;
} else {
    $this->view->album = $album->fetchRow('id='.$id);
}
}
} else {
    // album id should be $params['id']
    $id = (int)$this->_request->getParam('id', 0);
    if ($id > 0) {
        $this->view->album = $album->fetchRow('id='.$id);
    }
}

// additional view fields required by form
$this->view->action = 'edit';
$this->view->buttonText = 'Update';
}
...

```

Note that when we are not in “post” mode, we retrieve the id parameter from the request’s params property using `getParam()`.

The template is:

zf-tutorial/application/views/scripts/index/edit.phtml

```

<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('index/_form.phtml'); ?>
<?php echo $this->render('footer.phtml'); ?>

```

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 deletion. 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/controllers/IndexController.php

```

...
function deleteAction()
{
    $this->view->title = "Delete Album";

    $album = new Album();
    if ($this->_request->isPost()) {

```

```

        Zend_Loader::loadClass('Zend_Filter_Alpha');
        $filter = new Zend_Filter_Alpha();

        $id = (int)$this->_request->getPost('id');
        $del = $filter->filter($this->_request->getPost('del'));

        if ($del == 'Yes' && $id > 0) {
            $where = 'id = ' . $id;
            $rows_affected = $album->delete($where);
        }
    } else {
        $id = (int)$this->_request->getParam('id');
        if ($id > 0) {
            // only render if we have an id and can find the album.
            $this->view->album = $album->fetchRow('id='.$id);

            if ($this->view->album->id > 0) {
                // render template automatically
                return;
            }
        }
    }

    // redirect back to the album list unless we have rendered the view
    $this->_redirect('/');
}
...

```

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 setting the response's body. 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/scripts/index/delete.phtml

```

<?php echo $this->render('header.phtml'); ?>
<h1><?php echo $this->escape($this->title); ?></h1>
<?php if ($this->album) :?>
<form action="<?php echo $this->baseUrl ?>/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;?>
<?php echo $this->render('footer.phtml'); ?>

```

Troubleshooting

If you are having trouble getting any other action other than index/index working, then the most likely issue is that the router is unable to determine which subdirectory your website is

in. From my investigations so far, this usually happens when the url to your website differs from the directory path from the web-root.

If the default code doesn't work for you, then you should set the \$baseUrl to the correct value for your server:

zf-tutorial/index.php

```
...
// setup controller
$frontController = Zend_Controller_Front::getInstance();
$frontController->throwExceptions(true);
$frontController->setBaseUrl('/mysubdir/zf-tutorial');
$frontController->setControllerDirectory('./application/controllers');
...
```

You would need to replace '/mysubdir/zf-tutorial/' with the correct URL path to index.php. For instance, if your URL to index.php is <http://localhost/~ralle/zf-tutorial/index.php> then the correct value of \$baseUrl is is '/~ralle/zf-tutorial/'.

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](http://framework.zend.com/manual) (<http://framework.zend.com/manual>) and look at the [wiki](http://framework.zend.com/wiki) (<http://framework.zend.com/wiki>) for more insights! If you are interested in the development of the framework, then the [development wiki](http://framework.zend.com/developer) (<http://framework.zend.com/developer>) is worth a browse...