Getting Started with Zend Framework

By Rob Allen, www.akrabat.com
Document Revision 1.5.2

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

NOTE: This tutorial has been tested on versions 1.5 of Zend Framework. It stands a very good chance of working with later versions in the 1.5.x series, but will not work with versions prior to 1.5.

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.

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

Zend Framework has the following requirements:

- PHP 5.1.4 (or higher)
- A web server supporting mod_rewrite functionality.

Tutorial Assumptions

I have assumed that you are running PHP 5.1.4 or higher with the Apache web server. Your Apache installation must have the mod_rewrite extension installed and configured.

You must also ensure that Apache is configured to support .htaccess files. This is usually done by changing the setting:

AllowOverride None

to

AllowOverride All

in your httpd.conf file. Check with your distribution's documentation for exact details. You will not be able to navigate to any page other than the home page in this tutorial if you have not configured mod_rewrite and .htacess usage correctly.

Getting the Framework

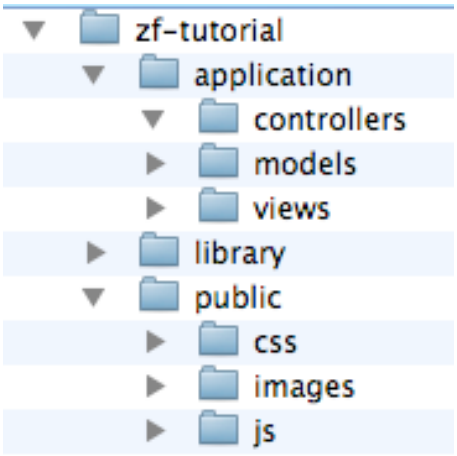
Zend Framework can be downloaded from http://framework.zend.com/download in either .zip or .tar.gz format.

Directory Structure

Whilst Zend Framework doesn't mandate a directory structure, the manual recommends a common directory structure, so this is what we will use. This structure assumes that you have complete control over your Apache configuration, so that you can keep most files outside of the web root directory.

Start by creating a directory in the web server's root directory called `zf-tutorial` and then create the following subdirectories to hold the website's files:

\



As you can see, we have separate directories for the model, view and controller files of our application. The `public/` directory is the root of the website, which means that the URL to get

to the application will be `http://localhost/zf-tutorial/public/`. This is so that most of the application's files are not accessible directly by Apache and so are more secure.

> **Note:**
> On a live website, you would create a virtual host for the website and set the document root directly to the public folder. For example you could create a virtual host called zf-tutorial.localhost that looked something like this:
>
> ```
> <VirtualHost *:80>
>     ServerName zf-tutorial.localhost
>     DocumentRoot /var/www/html/zf-tutorial/public
>     <Directory "/www/cs">
>         AllowOverride All
>     </Directory>
> </VirtualHost>
> ```
>
> The site would then be accessed using `http://zf-tutorial.localhost/` (make sure that you update your /etc/hosts or c\windows\system32\drivers\etc\hosts file so that zfia.localhost is mapped to 127.0.0.1)

Supporting images, JavaScript 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-1.5.0.zip in my case, to a temporary directory. All the files in the archive are placed into a subdirectory called `ZendFramework-1.5.0`. Copy the subdirectory `library/Zend` into `zf-tutorial/library/`. Your `zf-tutorial/library/` should now contain a sub-directory called `Zend`.

Bootstrapping

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. This is known as the Front Controller design pattern. 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/public` directory:

zf-tutorial/public/.htaccess

```
# Rewrite rules for Zend Framework
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule .* index.php

# Security: Don't allow browsing of directories
Options -Indexes

# PHP settings
php_flag magic_quotes_gpc off
php_flag register_globals off
php_flag short_open_tag on
```

The RewriteRule is very simple and can be interpreted as "for any url that doesn't map to a file that exists on disk, use index.php instead".

We also set a couple of PHP ini settings for security and sanity and set the short_open_tag setting to on for use in view scripts. 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 the make sure that your php.ini is correct.
Note that for .htaccess files be used by Apache, the configuration directive AllowOverride must be set to All within your httpd.conf file.

Bootstrap file: index.php

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

zf-tutorial/public/index.php

```
<?php

error_reporting(E_ALL|E_STRICT);
ini_set('display_errors', 1);
date_default_timezone_set('Europe/London');

// directory setup and class loading
set_include_path('.' . PATH_SEPARATOR . '../library/'
     . PATH_SEPARATOR . '../application/models'
     . PATH_SEPARATOR . get_include_path());
include "Zend/Loader.php";
Zend_Loader::registerAutoload();

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

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

Note that we do not put 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);
ini_set('display_errors', 1);
date_default_timezone_set('Europe/London');
```

These lines ensure that we will see any errors that we make. We also set up our current time zone as required by PHP 5.1+. Obviously, you should choose your own time zone.

```
// directory setup and class loading
set_include_path('.' . PATH_SEPARATOR . '../library/'
     . PATH_SEPARATOR . '../application/models'
     . PATH_SEPARATOR . get_include_path());
include "Zend/Loader.php";
Zend_Loader::registerAutoload();
```

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 gives us access to the `Zend_Loader` class and then call it's `registerAutoload()` member function in order to automatically load all Zend Framework files as we instantiate them.

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

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

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()`.

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/public/ to test, you should see a fatal error that starts with:

> **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:

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 Zend 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/public/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 `list`, `archived` and `view`. Zend Framework's MVC system also supports modules for grouping controllers together, but this application isn't big enough to worry about them!

By default, 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/public/news/` the `index` action within the news controller will be executed. There is also 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/public/` 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 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 `application/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 and again must be completely lowercase. Mixed case controller and action names are allowed, but have special rules that you must understand before you use them. Check the documentation first!

Thus our controller class is called `IndexController` which is defined in `zf-tutorial/application/controllers/IndexController.php`. Create this file to set up the skeleton:

zf-tutorial/application/controllers/IndexController.php

```
<?php

class IndexController extends Zend_Controller_Action
{
    function indexAction()
    {
    }

    function addAction()
    {
    }

    function editAction()
    {
    }

    function deleteAction()
    {
    }
}
```

We have now set up the four actions that we want to use. They won't work yet until we set up the views. The URLs for each action are:

URL	*Action*
http://localhost/zf-tutorial/public/	IndexController::indexAction()
http://localhost/zf-tutorial/public/index/add	IndexController::addAction()
http://localhost/zf-tutorial/public/index/edit	IndexController::editAction()
http://localhost/zf-tutorial/public/index/delete	IndexController::deleteAction()

We now have a working router and the actions are set up for each page of our application.

It's time to build the view.

Setting up the View

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('viewScipt.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 very boring "structural" 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 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 append it 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 and to prove that they work, we add some action specific content (the page title) into the controller actions.

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 view scripts or templates as noted above, 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 by navigating the urls shown earlier should display the four titles within the web browser.

Common HTML code

It very quickly becomes obvious that there is a lot of common HTML code in our views. This is a very common problem and the *Zend_Layout* component is designed to solve this problem. Zend_Layout allows us to move all the common header and footer code to a layout view script which then includes the view code that is specific to the action being executed.

The following changes are needed. Firstly we need to decide where to keep our layout view scripts. The recommended place is within the `application` directory, so create a directory called `layouts` within the `zf-tutorial/application` directory.

We need to tell start the Zend_Layout system in the bootstrap file, so we add to `public/index.php` like this:

zf-tutorial/public/index.php:

```
...
$frontController->throwExceptions(true);
$frontController->setControllerDirectory('../application/controllers');
Zend_Layout::startMvc(array('layoutPath'=>'../application/layouts'));

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

The `startMvc()` function does some work behind the scenes to set up a front controller plugin that will ensure that the Zend_Layout component renders the layout script with the action view scripts within it at the end of the dispatch process.

We now need a layout view script. By default, this is called layout.phtml and lives in the layouts directory. It looks like this:

zf-tutorial/application/layouts/layout.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">
    <h1><?php echo $this->escape($this->title); ?></h1>
    <?php echo $this->layout()->content; ?>
</div>
</body>
</html>
```

Note that we've made our code XHTML compliant and is standard HTML code for displaying a page. As the title of the page within the <h1> tag is displayed on all pages, we have moved this into the layout file and use the `escape()` view helper to ensure that it is properly encoded.

To get the view script for the current action to display, we echo out the content placeholder using the layout() view helper: `echo $this->layout()->content;` which does the work for us. This means that the view scripts for the action are run before the layout view script.

We can now clear out the 4 action scripts as we have nothing specific to put in them yet, so go ahead and empty the index.phtml, add.phtml, edit.phtml and delete.phtml files.

You can now test all 4 URLs again and should see no difference from last time you tested! The key difference is that this time, all the work is done in the layout.

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 create our own view helper, called `baseUrl()` that collects the information we require form the request object. This provides us with the bit of the URL that we don't know.

View helpers live in the `application/views/helpers` subdirectory and are named `{Helper name}.php` (the first letter must be uppercase) and the class inside must be called `Zend_Controller_Helper_{Helper name}` (again, uppercase first letter). There must be a function within the class called `{helper name}()` (lowercase first letter – don't forget!). In our case, the file is called `BaseUrl.php` and looks like this:

zf-tutorial/application/views/helpers/BaseUrl.php

```
<?php

class Zend_View_Helper_BaseUrl
{
    function baseUrl()
    {
        $fc = Zend_Controller_Front::getInstance();
        return $fc->getBaseUrl();
    }
}
```

Not a complicated function. We simply retrieve an instance to the front controller and return its `getBaseUrl()` member function.

We need to add the CSS file to the <head> section of the application/layouts/layout.phtml file:

zf-tutorial/application/layouts/layout.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();?>/css/site.css" />
</head>
...
```

Finally, we need some CSS styles:

zf-tutorial/public/css/site.css

```
body,html {
    margin: 0 5px;
    font-family: Verdana,sans-serif;
}
h1 {
    font-size:1.4em;
    color: #008000;
}
a {
    color: #008000;
}

/* Table */
th {
    text-align: left;
}
td, th {
    padding-right: 5px;
```

```
}

/* style form */
form dt {
    width: 100px;
    display: block;
    float: left;
    clear: left;
}
form dd {
    margin-left: 0;
    float: left;

}
form #submitbutton {
    margin-left: 100px;
}
```

This should make it look slightly prettier, but as you can tell, I'm not a designer!

The Database

Now that we have separated the control of the application from the 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 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.

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 called `config.ini` and store it in the `application/` directory:

zf-tutorial/application/config.ini

```
[general]
db.adapter = PDO_MYSQL
db.params.host = localhost
db.params.username = rob
db.params.password = 123456
db.params.dbname = zftest
```

Obviously you should use your username, password and database name, not mine! For larger applications with many config files you may decide to create a separate directory, such as `application/config` and store all your config files together, out of the way.

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->params->config`.

We will load our configuration file in our bootstrap file (public/index.php):

Relevant part of zf-tutorial/public/index.php

```
...
include "Zend/Loader.php";
Zend_Loader::registerAutoload();

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

// setup controller
$frontController = Zend_Controller_Front::getInstance();
...
```

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 a good idea as in a larger 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/public/index.php

```
...
$registry = Zend_Registry::getInstance();
$registry->set('config', $config);

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

// setup controller
$frontController = Zend_Controller_Front::getInstance();
...
```

As you can see, `Zend_Db_Table` has a static member function called `factory()` that interprets the data from the `$config->db` object and instantiates the correct database adapter for us.

Create the Table

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

```
CREATE TABLE albums (
  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 "Top Sellers" CDs from Amazon.co.uk:

```
INSERT INTO albums (artist, title)
VALUES
       ('Duffy', 'Rockferry'),
       ('Van Morrison', 'Keep It Simple');
```

(Nice to see that Van Morrison is still going…)

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 `Albums` as our table name is `albums`. 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 a file called Album.php within the applications/models directory:

zf-tutorial/application/models/Albums.php

```
<?php

class Albums extends Zend_Db_Table
{
    protected $_name = 'albums';
}
```

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 and we start by listing the albums in a table within the `indexAction()` function:

zf-tutorial/application/controllers/IndexController.php

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

    }
    ...
```

The `fetchAll()` function returns a `Zend_Db_Table_Rowset` which will allow us to iterate over the returned rows in the action's view script file. We can now fill in the index.phtml file:

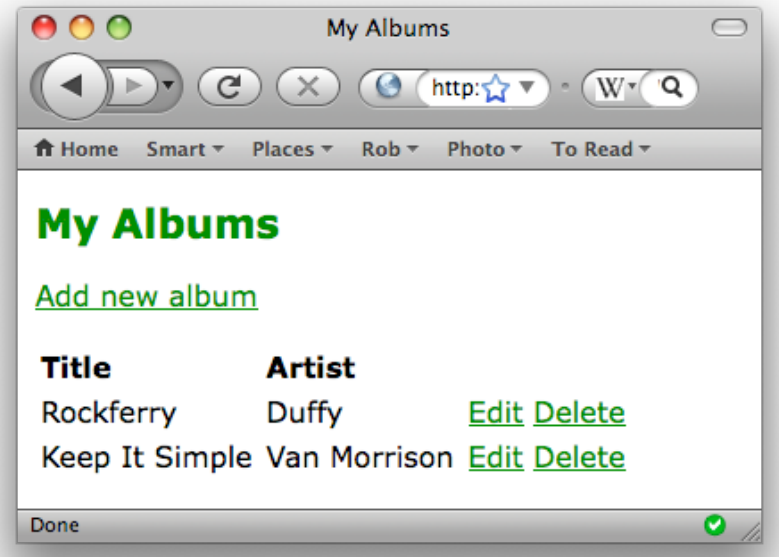
zf-tutorial/application/views/scripts/index/index.phtml

```
<p><a href="<?php echo $this->url(array('controller'=>'index',
        'action'=>'add'));?>">Add new album</a></p>
<table>
<tr>
    <th>Title</th>
    <th>Artist</th>
    <th> </th>
</tr>
<?php foreach($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 $this->url(array('controller'=>'index',
            'action'=>'edit', 'id'=>$album->id));?>">Edit</a>
        <a href="<?php echo $this->url(array('controller'=>'index',
            'action'=>'delete', 'id'=>$album->id));?>">Delete</a>
    </td>
</tr>
<?php endforeach; ?>
</table>
```

The first thing we do is to create a link to add a new album. The `url()` view helper is provided by the framework and helpfully creates links including the correct base URL. We simply pass in an array of the parameters we need and it will work out the rest as required.

We then create an html table to display each album's title, artist and provide links to allow for editing and deleting the record. A standard `foreach` loop is used to iterate over the list of albums, and we use the alternate form using a colon and `endforeach;` as it is easier to scan than to try and match up braces. Again, the `url()` view helper is used to create the edit and delete links.

http://localhost/zf-tutorial/ (or wherever you are following along from!) should now show a nice list of (two) albums, something like this:



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

We use *Zend_Form* to do this. The Zend_Form component allows us to create a form and validate the input. We create a new model class `AlbumForm` that extends from `Zend_Form` to define our form:

zf-tutorial/application/models/AlbumForm.php

```
<?php

class AlbumForm extends Zend_Form
{
    public function __construct($options = null)
    {
        parent::__construct($options);
        $this->setName('album');

        $id = new Zend_Form_Element_Hidden('id');

        $artist = new Zend_Form_Element_Text('artist');
        $artist->setLabel('Artist')
        ->setRequired(true)
        ->addFilter('StripTags')
        ->addFilter('StringTrim')
        ->addValidator('NotEmpty');

        $title = new Zend_Form_Element_Text('title');
        $title->setLabel('Title')
        ->setRequired(true)
        ->addFilter('StripTags')
        ->addFilter('StringTrim')
        ->addValidator('NotEmpty');

        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setAttrib('id', 'submitbutton');

        $this->addElements(array($id, $artist, $title, $submit));
    }
}
```

Within the constructor of `AlbumForm`, we create four form elements for the id, artist, title, and submit button. For each item we set various attributes, including the label to be displayed. For the text elements, we add two filters, `StripTags` and `StringTrim` to remove unwanted HTML and unnecessary whitespace. We also set them to be required and add a `NotEmpty` validator to ensure that the user actually enters the information we require.

We now need to get the form to display and then process it on submission. This is done within `addAction()`:

zf-tutorial/application/controllers/IndexController.php

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

        $form = new AlbumForm();
        $form->submit->setLabel('Add');
        $this->view->form = $form;

        if ($this->_request->isPost()) {
            $formData = $this->_request->getPost();
            if ($form->isValid($formData)) {
```

```
                $albums = new Albums();
                $row = $albums->createRow();
                $row->artist = $form->getValue('artist');
                $row->title = $form->getValue('title');
                $row->save();

                $this->_redirect('/');
            } else {
                $form->populate($formData);
            }
        }
    }
...
```

Let's examine this in a bit more detail:

```
$form = new AlbumForm();
$form->submit->setLabel('Add');
$this->view->form = $form;
```

We instantiate our `AlbumForm`, set the label for the submit button to "Add" and then assign to the view for rendering.

```
    if ($this->_request->isPost()) {
        $formData = $this->_request->getPost();
        if ($form->isValid($formData)) {
```

If the request object's `isPost()` method is true, then the form has been submitted and so we retrieve the form data from the request using `getPost()` and check to see if it is valid using the `isValid()` member function.

```
            $albums = new Albums();
            $row = $albums->createRow();
            $row->artist = $form->getValue('artist');
            $row->title = $form->getValue('title');
            $row->save();

            $this->_redirect('/');
```

If the form is valid, then we instantiate the `Albums` model class and use `createRow()` to retrieve an empty row and then populate the artist and title before saving. After we have saved the new album row, we redirect using the controller's `_redirect()` method to go back to the home page.

```
        } else {
            $form->populate($formData);
        }
```

If the form data is not valid, then we populate the form with the data that the user filled in and redisplay.

We now need to render the form in the `add.phtml` view script:

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

```
<?php echo $this->form ;?>
```

As you can see, rendering a form is very simple as the form knows how to display itself.

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";

    $form = new AlbumForm();
    $form->submit->setLabel('Save');
    $this->view->form = $form;

    if ($this->_request->isPost()) {
        $formData = $this->_request->getPost();
        if ($form->isValid($formData)) {
            $albums = new Albums();
            $id = (int)$form->getValue('id');
            $row = $albums->fetchRow('id='.$id);
            $row->artist = $form->getValue('artist');
            $row->title = $form->getValue('title');
            $row->save();

            $this->_redirect('/');
        } else {
            $form->populate($formData);
        }
    } else {
        // album id is expected in $params['id']
        $id = (int)$this->_request->getParam('id', 0);
        if ($id > 0) {
            $albums = new Albums();
            $album = $albums->fetchRow('id='.$id);
            $form->populate($album->toArray());
        }
    }
}
...
```

Let's look at the differences from adding an album. Firstly, when displaying the form to the user we need to retrieve the album's artist and title from the database and populate the form's elements with it:

```
        // album id is expected in $params['id']
        $id = (int)$this->_request->getParam('id', 0);
        if ($id > 0) {
            $albums = new Albums();
            $album = $albums->fetchRow('id='.$id);
            $form->populate($album->toArray());
        }
```

This is done if the request is not a POST and uses the model to retrieve the database row. The `Zend_Db_Table_Row` class has a member function called `toArray()` that we can use to populate the form directly.

Finally, we need to save the data back to the right database row. This is done by retrieving the data row and then saving it again:

```
            $albums = new Albums();
            $id = (int)$form->getValue('id');
            $row = $albums->fetchRow('id='.$id);
```

The view template is the same as add.phtml:

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

```
<?php echo $this->form ;?>
```

You should now be able to add and edit albums.

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 recall that you shouldn't do an irreversible action using GET and should use POST instead.

We shall show a confirmation form when the user clicks delete and if they then click "yes", we will do the deletion. As this form is trivial, we'll just create the form HTML in the view script.

Let's start with the action code:

zf-tutorial/application/controllers/IndexController.php

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

        if ($this->_request->isPost()) {
            $id = (int)$this->_request->getPost('id');
            $del = $this->_request->getPost('del');
            if ($del == 'Yes' && $id > 0) {
                $albums = new Albums();
                $where = 'id = ' . $id;
                $albums->delete($where);
            }
            $this->_redirect('/');
        } else {
            $id = (int)$this->_request->getParam('id');
            if ($id > 0) {
                $albums = new Albums();
                $this->view->album = $albums->fetchRow('id='.$id);
            }
        }
    }
    ...
```

We use the requests isPost() method to work out if we should display the confirmation form or if we should do a deletion, via the Album() class. The actual deletion is done via a call to delete() method of `Zend_Db_Table`. If the request is not a post, then we look for an id parameter and retrieve the correct database record and assign to the view.

The view script is a simple form:

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

```
<?php if ($this->album) :?>
<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>
<form action="<?php echo $this->url(array('action'=>'delete')); ?>"
method="post">
<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;?>
```

In this script, we display a confirmation message to the user and then a form with yes and no buttons. In the action, we checked specifically for the "Yes" value when doing the deletion.

That's it - you now have a fully working application.

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/public/index.php

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

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

Conclusion

This concludes our brief look at building a simple, but fully functional, MVC application using 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) and look at the 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) is worth a browse too...

Finally, if you prefer the printed page, then I'm currently in the process of writing a book called *Zend Framework in Action* which is available for pre-order. Further details are available at http://www.zendframeworkinaction.com. Check it out ☺