Getting Started with Zend Framework

By Rob Allen, www.akrabat.com
Document Revision 1.6.2
Copyright © 2006, 2009

This tutorial is intended to give an introduction to using Zend Framework by creating a simple database driven application using the Model-View-Controller paradigm.

NOTE: This tutorial has been tested on **version 1.8** of Zend Framework. It stands a very good chance of working with later versions in the 1.x series, but will not work with versions prior to 1.8.

Requirements

Zend Framework has the following requirements:

- PHP 5.2.4 (or higher)
- A web server supporting mod_rewrite or similar functionality.

Tutorial assumptions

I have assumed that you are running PHP 5.2.4 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 .htaccess usage correctly.

Getting the framework

Zend Framework can be downloaded from http://framework.zend.com/download in either .zip or .tar.gz format. Look at the bottom of the page for direct links.

Setting up Zend_Tool

Zend Framework is supplied with a new command line tool. We start by setting it up.

Zend_Tool for Windows

- Create a new directory in `Program Files` called `ZendFrameworkCli`
- Double click the downloaded archive file, `ZendFramework-1.8.0-minimal.zip`.
- Copy the `bin` and `library` folders from within the `ZendFramework-1.8.0-minimal.zip` folder window to the `C:\Program Files\ZendFrameworkCli` folder. This folder should now have two sub folders: `bin` and `library`.
- Add the `bin` directory to your path:
  - Go to the System section of the Control Panel.
  - Choose Advanced and then press the Environment Variables button.
  - In the "System variables" list, find the `Path` variable and double click on it.
  - Add `;C:\Program Files\ZendFrameworkCli\bin` to the end of the input box and press okay. (The semicolon is important!)
  - Reboot.

Zend_Tool for OS X (Linux is similar)

- Extract the downloaded archive file, `ZendFramework-1.8.0b1-minimal.zip` in your `Downloads` directory by double clicking on it.
- Copy to `/usr/local/ZendFrameworkCli` by opening Terminal and typing:
  `sudo cp -r ~/Downloads/ZendFramework-1.8.0-minimal /usr/local/ZendFrameworkCli`
- Edit your bash profile to provide an alias:
  - From Terminal, type: `open ~/.bash_profile`
  - Add `alias zf=/usr/local/ZendFrameworkCli/bin/zf.sh` to the end of the file
  - Save and exit TextEdit.
  - Exit Terminal.

Testing Zend_Tool

You can test your installation of the Zend_Tool command line interface by opening a Terminal or Command Prompt and typing:

```
zf show version
```

If all has worked, you should see:

```
Zend Framework Version: 1.8.0
```

If not, then check you set up the path correctly and that the bin directory exists in the ZendFrameworkCli directory.

Getting our application off the ground

Now that all the pieces are in place, we can build a Zend Framework application. 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 very simple table schema like this:

Field name	Type	Null?	Notes
id	integer	No	Primary key, auto increment
artist	varchar(100)	No	
title	varchar(100)	No	

The following pages will be required.

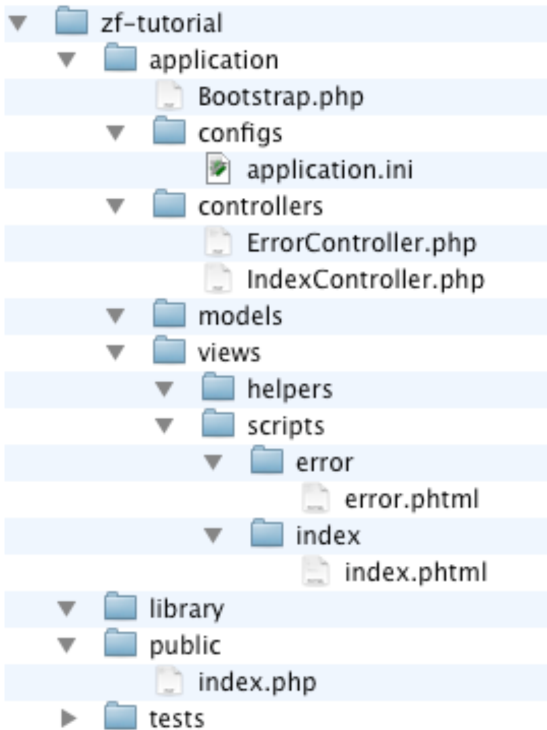
Home page	This will display the list of albums and provide links to edit and delete them. Also, a link to enable adding new albums will be provided.
Add New Album	This page will provide a form for adding a new album
Edit Album	This page will provide a form for editing an album
Delete Album	This page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

Create the project

Open Terminal or Command Prompt and type and change direcotry to root of your web server using the `cd` command. Ensure you have permissions to create files in this directory and that the webserver has read permissions. Type:

```
zf create project zf-tutorial
```

The ZF tool will create a directory called `zf-tutorial` and populate it with the recommended directory structure. 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. You should see the following files and directories:



(There is also a hidden .htaccess file in `public/`).

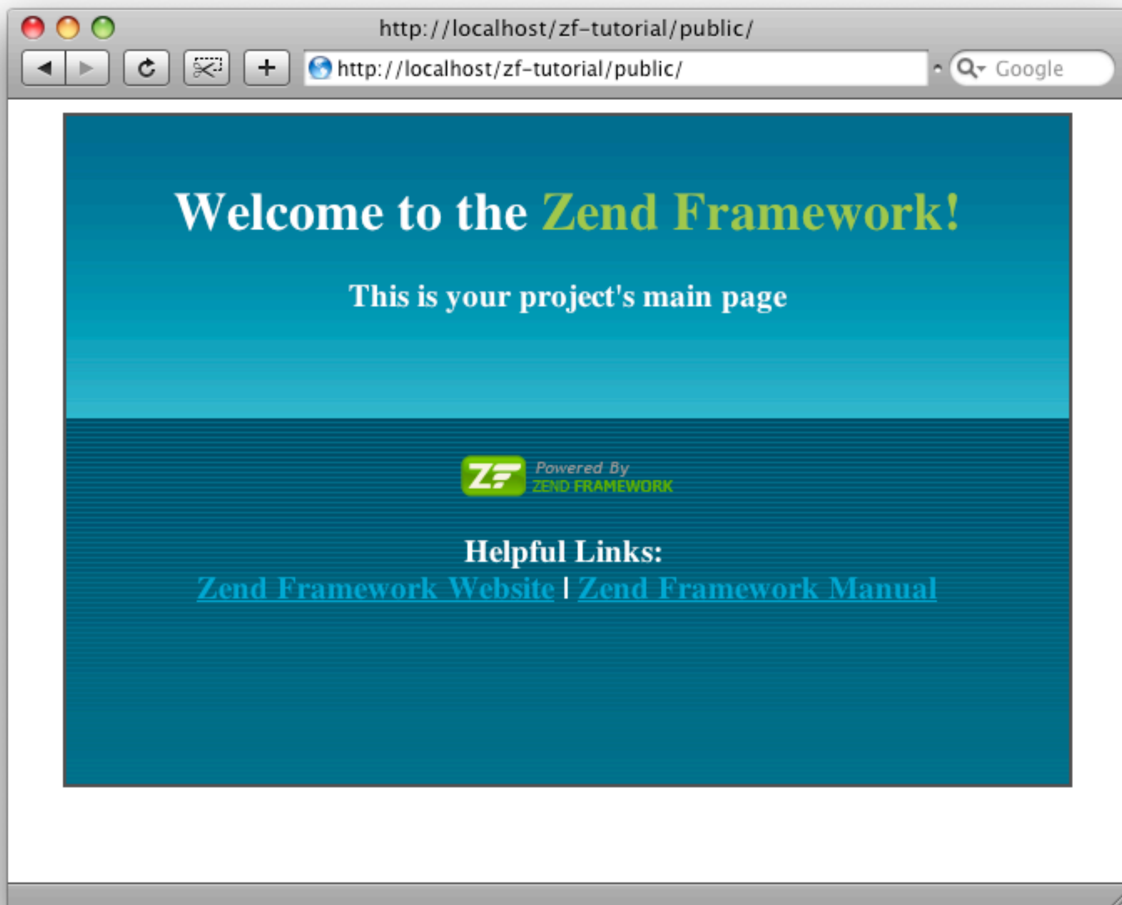
The `application/` directory is where the source code for this website lives. As you can see, we have separate directories for the model, view and controller files of our application. The `public/` directory is the public-facing root of the website, which means that the URL to get to the application 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` directory. 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/z-ftutorial/public
>     <Directory "/var/www/html/zf-tutorial/public">
>         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 zf-tutorial.localhost is mapped to 127.0.0.1). We will not be doing this in this tutorial though as it's just as easy to use a subdirectory for testing.

Supporting images, JavaScript and CSS files are stored in separate directories under the `public/` directory. The downloaded Zend Framework files will be placed in the `library/` directory. If we need to use any other libraries, they can also be placed here.

Copy the `library/Zend/` directory from downloaded archive file (`ZendFramework-1.8.0.zip`) into your `zf-tutorial/library/`, so that your `zf-tutorial/library/` contains a sub-directory called `Zend/`.

You can test that all is well by navigating to http://localhost/zftutorial/public. You should see something like this:



Bootstrapping background

Zend Framework's controller uses the Front Controller design pattern and routes all requests through a single `index.php` file. This ensures that the environment is set up correctly for running the application (known as bootstrapping). We achieve this using an `.htaccess` file in the `zf-tutorial/public` directory that is generated for us by `Zend_Tool` which redirects all requests to `public/index.php` which is also created by `Zend_Tool`.

The `index.php` file is the entry point to our application and is used to create an instance of `Zend_Application` to initialise our application and then run it. This file also defines two constants: `APPLICATION_PATH` and `APPLICATION_ENV` which define the path to the `application/` directory and the environment or mode of the application. The generated .htaccess file sets it to `development`.

The `Zend_Application` component is used to start up the application and is configured to use directives in the configuration file: `application/configs/application.ini`. This file is also auto-generated for us .

A Bootstrap class that extends Zend_Application_Bootstrap_Bootstrap is provided in application/Boostrap.php which can be used to execute any specific start-up code required.

Editing the application.ini file

Although Zend_Tool provides a decent default configuration file, we need to add our application specific items. Edit application/configs/application.ini and add:

```
phpSettings.date.timezone = "UTC"
```

after all the other phpSettings values in the [production] section. Obviously, you should probably use your own time zone.

Adding autoloading

We also need to create an autoloader that will enable us to automatically load resources within the application directory such as models and forms. The Zend_Application_Module_Autoloader class is used to do this and we implement it within in the Bootstrap class. The section you should add is in bold:

application/Bootstrap.php

```
<?php

class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initAutoload()
    {
        $moduleLoader = new Zend_Application_Module_Autoloader(array(
            'namespace' => '',
            'basePath'  => APPLICATION_PATH));
        return $moduleLoader;
    }
}
```

This method is called automatically by Zend_Application as it starts with _init. The remainder of the method name is up to you. The module autoloader will autoload classes with a certain prefix that are in certain directories within application/ as per this table:

Directory	Prefix	Example
api	Api_	Api_Rest
forms	Form_	Form_Login
models	Model_	Model_News
models/DbTable	Model_DbTable_	Model_DbTable_News
plugins	Plugin_	Plugin_

We are now in a position to add our application specific code.

The application specific code

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**. 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!

that this is also called `index` and so 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

As a site gets more complicated, additional controllers are needed and possibly you would even group of controllers into modules.

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 be within a file called `{Controller name}Controller.php` within the `application/controllers` directory. 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!

Our controller class is called `IndexController` which is defined in `application/controllers/IndexController.php` and has been automatically created for us by Zend_Tool. We just need to add our additional actions.

Adding additional controller actions is done using the `zf` command line tool. Open up Terminal or Command Prompt and change directory to your `zf-tutorial/` directory. Then type this:

```
zf create action add index
zf create action edit index
zf create action delete index
```

These commands create the xxxAction functions in `IndexController` and also create the appropriate view script files that we'll need later. We have now set up the four actions that we want to use.

The URLs for each action are:

URL	Action method
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()

You can test them and should see a message like this:

View script for controller **index** and script/action name **add**

It's time to look at the database.

The database

Now that we have the skeleton application done with controller action functions and view files ready, 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, 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.

Database configuration

To use `Zend_Db_Table`, we need to tell it which database to use along with a user name 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_Application` component is shipped with a database configuration resource, so all we need to do is set up the appropriate information in the `configs/application.ini` file and it will do the rest.

Open configs/application.ini and add the following to the end of the [production] section (i.e. above `[staging]`):

```
resources.db.adapter = PDO_MYSQL
resources.db.params.host = localhost
resources.db.params.username = rob
resources.db.params.password = 123456
resources.db.params.dbname = zf-tutorial
```

Obviously you should use your user name, password and database name, not mine! The database connection will now be made for us and `Zend_Db_Table`'s default adapter will be set.

Create the database 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 some rows into the table so that we can see the retrieval functionality of the home page. I'm going to take the first few "Top Sellers" CDs from Amazon UK. Run the following statement in your MySQL client:

```
INSERT INTO albums (artist, title)
VALUES
('Bob Dylan', 'Together Through Life'),
('Various Artists', 'Now That\'s what I Call Music! 72'),
('Lady Gaga', 'The Fame'),
('Lily Allen', 'It\'s Not Me, It\'s You'),
('Kings of Leon', 'Only By The Night');
```

We now have some data in a database and can write a very simple model for it

The Model

Zend Framework provides `Zend_Db_Table` which implements the Table Data Gateway design pattern to allow for interfacing with data in a database table. For more complicated projects, it is usually worth creating a model class that uses one or more `Zend_Db_Table` instances via protected member variables. For this tutorial however, we are going to create a model that extends `Zend_Db_Table`.

`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 after the database table. When we include the prefix, our class will be called `Model_DbTable_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 `Albums` class in a file called `Albums.php` within the `applications/models/DbTable` directory:. Create this file and enter the following code:

zf-tutorial/application/models/DbTable/Albums.php

```
<?php

class Model_DbTable_Albums extends Zend_Db_Table
{
    protected $_name = 'albums';

    public function getAlbum($id)
    {
        $id = (int)$id;
        $row = $this->fetchRow('id = ' . $id);
        if (!$row) {
            throw new Exception("Count not find row $id");
        }
        return $row->toArray();
    }

    public function addAlbum($artist, $title)
    {
        $data = array(
            'artist' => $artist,
            'title' => $title,
        );
        $this->insert($data);
    }

    function updateAlbum($id, $artist, $title)
    {
        $data = array(
            'artist' => $artist,
            'title' => $title,
        );
        $this->update($data, 'id = '. (int)$id);
    }

    function deleteAlbum($id)
    {
        $this->delete('id =' . (int)$id);
    }
}
```

We create four helper methods that our application will use to interface to the database table. `getAlbum()` retrieves a single row as an array, `addAlbum()` creates a new row in the database, `updateAlbum()` updates an album row and `deleteAlbum()` removes the row completely. The code for each of these

methods is self-explanatory. Whilst not needed in this tutorial, you can also tell Zend_Db_Table about related tables and it can fetch related data too.

We need to fill in the controllers with data from the model and get the view scripts to display it, however, before we can do that, we need to understand how Zend Framework's view system works.

Layouts and views

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/scripts');
echo $view->render('script.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. Zend Framework provides an Action Helper for us called the ViewRenderer. This takes care of initialising a view property in the controller ($this->view) for us to use and will also render a view script after the action has been dispatched.

For the rendering, the ViewRenderer 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.

This is all set up for us by Zend_Tool when we create the project and add controllers and actions using the zf creation controller and zf create action commands.

Common HTML code: Layouts

It also very quickly becomes obvious that there will be a lot of common HTML code in our views for the header and footer sections at least and maybe a side bar or two also. 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, footer and other code to a layout view script which then includes the specific view code for the action being executed.

The default place to keep our layouts is application/layouts/ and there is a resource available for Zend_Application that will configure Zend_Layout for us.

Firstly, create the application/layouts/ directory and then add this line to the end of the [production] section of the configs/applications.ini file.

```
resources.layout.layoutpath = APPLICATION_PATH "/layouts"
```

We also need to set up the global settings for our view in the Bootstrap class. Again, we use an _init method which we'll call _initViewHelpers(), so edit application/Bootstrap.php and add the following underneath the _initAutoload() method:

application/Bootstrap.php

```
    ...
    function _initViewHelpers()
    {
        $this->bootstrap('layout');
        $layout = $this->getResource('layout');
        $view = $layout->getView();

        $view->doctype('XHTML1_STRICT');
        $view->headMeta()->appendHttpEquiv('Content-Type', 'text/html;charset=utf-8');
        $view->headTitle()->setSeparator(' - ');
        $view->headTitle('Zend Framework Tutorial');
    }
    ...
```

We use the `bootstrap()` member variable to ensure that the layout resource is initialised and then we retrieve the `Zend_Layout` object using `getResource()` and then retrieve the view using the layout's `getView()` method.

Once we have the `$view` instance, We can call various helper methods to set up ready for rendering later. The `doctype()` view helper is used to set the doc-type that we want. This is used by various other view helpers to ensure that the correct HTML code is generated. We use `headMeta()` to set the content type meta tagand then the `headTitle()` view helper sets the last part of the `<title>` element along with the separator between each part.

When dispatching, `Zend_Layout` will now look for a layout view script called `layout.phml` in the `application/layouts` directory, so we'd better write one! This is the file:

zf-tutorial/application/layouts/layout.phtml

```
<?php echo $this->doctype('XHTML1_TRANSITIONAL'); ?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <?php echo $this->headMeta(); ?>
    <?php echo $this->headTitle(); ?>
</head>
<body>
<div id="content">
    <h1><?php echo $this->escape($this->title); ?></h1>
    <?php echo $this->layout()->content; ?>
</div>
</body>
</html>
```

The layout file contains the "outer" HTML code which is all pretty standard. As it is a normal PHP file, we can use PHP within it, There is a variable, `$this`, available which is an instance of the view object that was created during bootstrapping. We can use this to retrieve data that has been assigned to the view and also to call methods. The methods (known as view helpers) return a string that we can then echo.

Firstly we echo out the view helpers that we set up in `Bootstrap::_initViewHelpers()` which will create the correct code for the `<head>` section. Within the `<body>`, we create a div with an `<h1>` containing the title. 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.

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 `{HelperName}.php` (the first letter must be uppercase) and there is an expected naming convention for the class inside; it must be called `Zend_Controller_Helper_{HelperName}` (again, with an uppercase first letter). There must be a function within the class called `{helperName}()` (lowercase first letter this time – 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();
    }
}
```

This is 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 and again we use a view helper, headLink():

zf-tutorial/application/layouts/layout.phtml
```
...
<head>
    <?php echo $this->HeadMeta(); ?>
    <?php echo $this->headTitle(); ?>
    <?php echo $this->headLink()->prependStylesheet($this->baseUrl().'/css/site.css'); ?>
</head>
...
```

By using `headLink()`, we allow for additional, more specific, CSS files to be added within the controller view scripts which will be rendered to the <head> section after `site.css`.

Finally, we need some CSS styles, so create a `css` directory within `public/`:

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!

We can now clear out the four action scripts that were auto generated for us ready for filling up, so go ahead and empty the `index.phtml`, `add.phtml`, `edit.phtml` and `delete.phtml` files which, as you'll no doubt remember, are in the `application/views/scripts/index` directory.

Listing albums

Now that we have set up configuration, database information and our view skeletons, 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";
        $this->view->headTitle($this->view->title, 'PREPEND');

        $albums = new Model_DbTable_Albums();
        $this->view->albums = $albums->fetchAll();
    }
    ...
```

Firstly we set the title for the page and then prepend it to the head title for display in the brower's title bar.

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 associated view script, `index.phtml`.

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

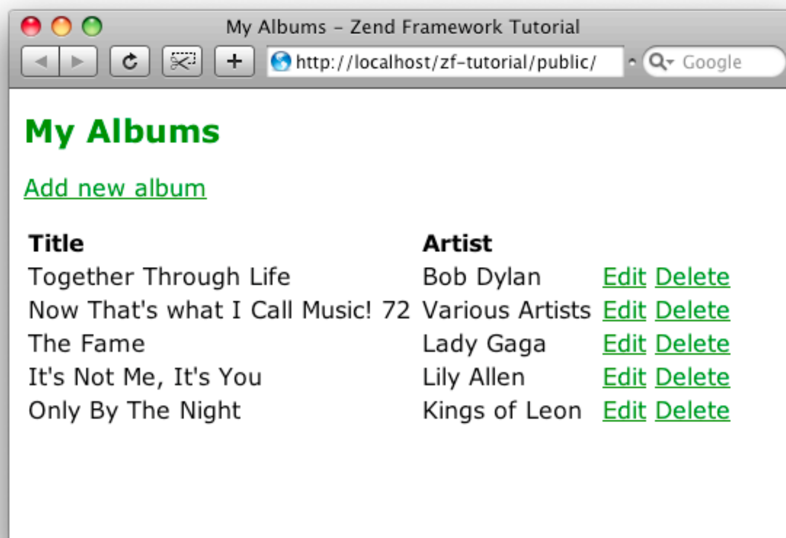
```
<p><a href="<?php echo $this->url(array('controller'=>'index',
        'action'=>'add'));?>">Add new album</a></p>
<table>
<tr>
    <th>Title</th>
    <th>Artist</th>
    <th> </th>
</tr>
<?php foreach($this->albums as $album) : ?>
<tr>
    <td><?php echo $this->escape($album->title);?></td>
    <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 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 class `Form_Album` that extends from `Zend_Form` to define our form. As we are using the module autoloader, the class is stored in the `Album.php` file within the `forms` directory:

zf-tutorial/application/forms/Album.php

```
<?php

class Form_Album 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 `Form_Album`, 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 white space. 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 the `IndexController`'s `addAction()`:

zf-tutorial/application/controllers/IndexController.php

```
    ...
    function addAction()
    {
        $this->view->title = "Add new album";
        $this->view->headTitle($this->view->title, 'PREPEND');

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

        if ($this->getRequest()->isPost()) {
            $formData = $this->getRequest()->getPost();
            if ($form->isValid($formData)) {
                $artist = $form->getValue('artist');
                $title = $form->getValue('title');
                $albums = new Model_DbTable_Albums();
                $albums->addAlbum($artist, $title);

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

Let's examine it in a bit more detail:

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

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

```
if ($this->getRequest()->isPost()) {
    $formData = $this->getRequest()->getPost();
    if ($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.

```
        $artist = $form->getValue('artist');
        $title = $form->getValue('title');
        $albums = new Model_DbTable_Albums();
        $albums->addAlbum($artist, $title);
```

If the form is valid, then we instantiate the `Model_DbTable_Albums` model class and use `addAlbum()` method that we created earlier to create a new record in the database.

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

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";
    $this->view->headTitle($this->view->title, 'PREPEND');

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

    if ($this->getRequest()->isPost()) {
```

```
            $formData = $this->getRequest()->getPost();
            if ($form->isValid($formData)) {
                $id = (int)$form->getValue('id');
                $artist = $form->getValue('artist');
                $title = $form->getValue('title');
                $albums = new Model_DbTable_Albums();
                $albums->updateAlbum($id, $artist, $title);

                $this->_redirect('/');
            } else {
                $form->populate($formData);
            }
        } else {
            $id = $this->_getParam('id', 0);
            if ($id > 0) {
                $albums = new Model_DbTable_Albums();
                $form->populate($albums->getAlbum($id));
            }
        }
    }
    ...
```

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. This is at the bottom of the method:

```
           $id = $this->_getParam('id', 0);
            if ($id > 0) {
                $albums = new Model_DbTable_Albums();
                $form->populate($albums->getAlbum($id));
            }
```

This is done if the request is not a POST. We retrieve the id from request using the `_getParam()` method. We then use the model to retrieve the database row and populate the form with the row's data directly..

After validating the form , we need to save the data back to the correct database row. This is done using our model's updateAlbum() method:

```
                $id = (int)$form->getValue('id');
                $artist = $form->getValue('artist');
                $title = $form->getValue('title');
                $albums = new Model_DbTable_Albums();
                $albums->updateAlbum($id, $artist, $title);
```

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 the form is trivial, we'll code it directly into our view.

Let's start with the action code in `IndexController::deleteAction()`:

zf-tutorial/application/controllers/IndexController.php

```
...
 public function deleteAction()
{
    $this->view->title = "Delete album";
    $this->view->headTitle($this->view->title, 'PREPEND');

    if ($this->getRequest()->isPost()) {
        $del = $this->getRequest()->getPost('del');
        if ($del == 'Yes') {
            $id = $this->getRequest()->getPost('id');
            $albums = new Model_DbTable_Albums();
            $albums->deleteAlbum($id);
        }
        $this->_redirect('/');
    } else {
        $id = $this->_getParam('id', 0);
        $albums = new Model_DbTable_Albums();
        $this->view->album = $albums->getAlbum($id);
    }
}
...
```

As with add and edit, we use the request's `isPost()` method to work out if we should display the confirmation form or if we should do a deletion. We use `Model_DbTable_Albums()` to actually delete the row using the `deleteAlbum()` method. 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

```
<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>
```

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.

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 looked at the basics of using the framework; there are many more components to explore! You should read the manual (http://framework.zend.com/manual) too! If you are interested in the development of the framework, then read the development wiki (http://framework.zend.com/developer) too...

Finally, if you prefer the printed page, then I have written a book called *Zend Framework in Action* which is available to purchase. Further details are available at http://www.zendframeworkinaction.com. Check it out ☺