Getting Started with Zend Framework

By Rob Allen, www.akrabat.com
Document Revision 1.7.7
Copyright © 2006, 2011

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 1.10.1 through to 1.11.4** 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 1.10.1.

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/latest in either .zip or .tar.gz format. Look at the bottom of the page for direct links. The "Minimal" version is what you need.

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-1.11.10-minimal.zip`.
- Copy the `bin` and `library` folders from within the `ZendFramework-1.11.10-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 leading semicolon is important!)

- Reboot.

Zend_Tool for OS X (Linux is similar)

- Extract the downloaded archive file, `ZendFramework-1.11.10-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-1.11.10-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: 1.11.10
```

If not, then check you set up the path correctly and that the bin directory exists in the ZendFrameworkCli directory. Once the `zf` tool is working, `zf --help` will show you all the commands available.

Note: If your PHP distribution ships with Zend Framework, please check that it isn't using ZF 1.9 as this tutorial will not work. At the time of writing, the xxamp distribution did this.

The tutorial application

Now that all the pieces are in place that we can build a Zend Framework application, let's look at the application that we are going to build. 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. As with any software engineering, it helps if we do a little pre-planning. We are going to need four pages in our website:

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.

We will also need to store our data into a database. We will only need one table with these fields in it:

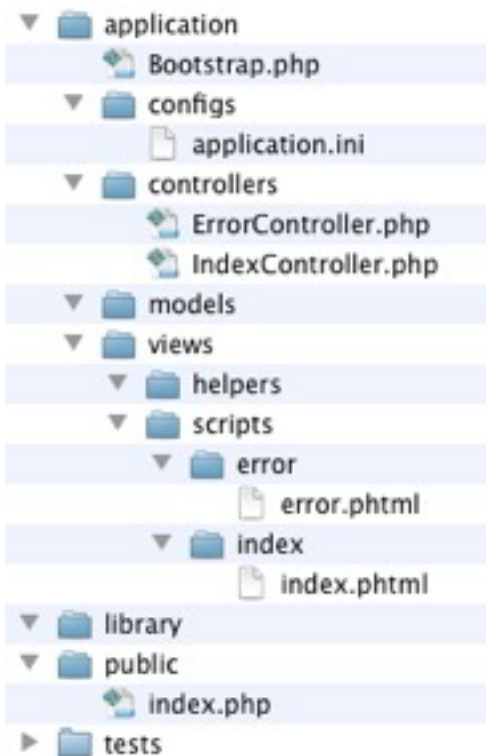| Field name	Type	Null?	Notes
id	integer	No	Primary key, auto increment
artist	varchar(100)	No	
title	varchar(100)	No	

Getting our application off the ground

Let's start building our application. Where possible, we will make use of the `zf` command line tool as it saves us time and effort. The first job is to create the project skeleton files and directories.

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

```
▼ 📁 application
      🔷 Bootstrap.php
   ▼ 📁 configs
         📄 application.ini
   ▼ 📁 controllers
         🔷 ErrorController.php
         🔷 IndexController.php
   ▼ 📁 models
   ▼ 📁 views
      ▼ 📁 helpers
      ▼ 📁 scripts
         ▼ 📁 error
               📄 error.phtml
         ▼ 📁 index
               📄 index.phtml
▼ 📁 library
▼ 📁 public
      🔷 index.php
▶ 📁 tests
```

(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/zf-tutorial/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-1.10.6-minimal.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/zf-tutorial/public. You should see something like this:
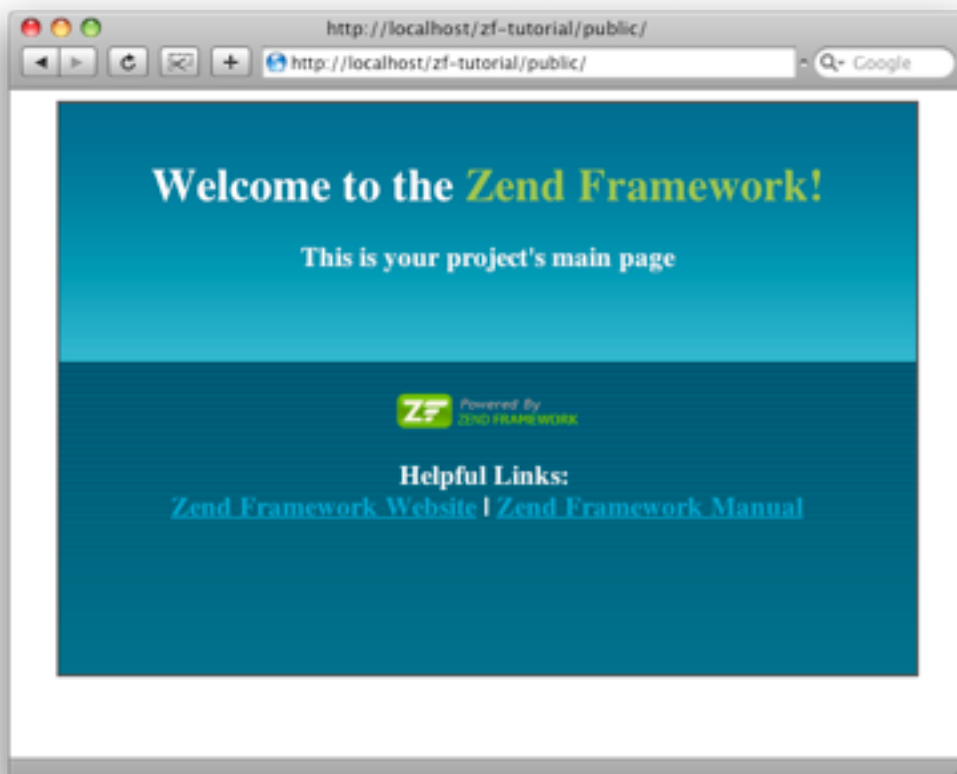
Bootstrapping background

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 default is set to `production` in `index.php`, but you should set it to `development` in the `.htaccess` file by adding this line:

```
SetEnv APPLICATION_ENV 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/Bootstrap.php` which can be used to execute any specific start-up code required.

The, `application.ini`, that is stored in the `application/configs` directory is loaded using the `Zend_Config_Ini` component. `Zend_Config_Ini` understands the concept of inheritance of sections which are denoted by using a colon on the section name. For example:

```
[staging : production]
```

This means that the `staging` section inherits all the `production` section's settings. The `APPLICATION_ENV` constant defines which section is loaded. Obviously, whilst developing, the `development` section is best and when on the live server, the `production` sections should be used. We will put all changes we make to `application.ini` within the `production` section so that all configurations will load the changes we make.

Editing the application.ini file

The first change we need to make is to add is our timezone information for PHP's date and time functionality. Edit `application/configs/application.ini` and add:

```
phpSettings.date.timezone = "Europe/London"
```

after all the other `phpSettings` values in the `[production]` section. Obviously, you should probably use your own time zone. We are now in a position to add our application specific code.

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/zf-tutorial/public/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. This is for cases like `http://localhost/zf-tutorial/public/news/` where the `index` action within the `News` controller will be executed. There is also a default controller name, which 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 (or you can read about it in *Zend Framework in Action*!)

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

Page	Controller	Action
Edit album	Index	edit
Delete album	Index	delete

As a site gets more complicated, additional controllers are needed and you can even group controllers into modules if needed.

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}` starts 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. It also contains the first action method, `indexAction()`. We just need to add our additional actions.

Adding additional controller actions is done using the `zf` command line tool's `create action` command. Open up Terminal or Command Prompt and change directory to your `zf-tutorial/` directory. Then type these three commands:

```
zf create action add Index
zf create action edit Index
zf create action delete Index
```

These commands create three new methods: `addAction`, `editAction` and `deleteAction` in `IndexController` and also create the appropriate view script files that we'll need later. We now have all 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 the three new actions and should see a message like this:

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

Note: If you get a 404 error, then you have not configured Apache with the mod_rewrite module or have not set up the AllowOverride correctly in your Apache config files so that the `.htaccess` file in the `public/` folder is actually used.

The database

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

```
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 automatically and `Zend_Db_Table`'s default adapter will be set. You can read about the other available resource plugins here: http://framework.zend.com/manual/en/zend.application.available-resources.html.

Create the database table

As noted in the initial planning, we are going to be using a database to store our album data. 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 data

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 "Bestsellers" CDs from Amazon UK. Run the following statement in your MySQL client:

```
INSERT INTO albums (artist, title)
VALUES
('Paolo Nutine', 'Sunny Side Up'),
('Florence + The Machine', 'Lungs'),
('Massive Attack', 'Heligoland'),
('Andre Rieu', 'Forever Vienna'),
('Sade', 'Soldier of Love');
```

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

The Model

Zend Framework does not provide a `Zend_Model` class as the model is your business logic and it's up to you to decide how you want it to work. There are many components that you can use for this depending on your needs. One approach is to have model classes that represent each entity in your application and then

use mapper objects that load and save entities to the database. This approach is documented in the Zend Framework QuickStart here: http://framework.zend.com/manual/en/learning.quickstart.create-model.html.

For this tutorial, we are going to create a model that extends `Zend_Db_Table` and uses `Zend_Db_Table_Row`. Zend Framework provides `Zend_Db_Table` which implements the Table Data Gateway design pattern to allow for interfacing with data in a database table. Be aware though that the Table Data Gateway pattern can become limited in larger systems. There is also a temptation to put database access code into controller action methods as these are exposed by Zend_Db_Table.

`Zend_Db_Table_Abstract` is an abstract class, from which we 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. Our project has a default autoloader instantiated by `Zend_Application` which maps the resource classes within under a module to the directory where it is defined. For the main `application/` folders we use the prefix `Application_`.

The autoloader maps resources to directories using this mapping:

Prefix	Directory
Form	forms
Model	models
Model_DbTable	models/DbTable
Model_Mapper	models/mappers
Plugin	plugins
Service	services
View_Filter	views/filters
View_Helper	views/helpers

As we are naming after the database table, albums and will use `Zend_Db_Table` our class will be called `Application_Model_DbTable_Albums` which will be stored in `applications/models/DbTable/Albums.php`.

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 can use the zf command line tool to do some of the work, so run the following command from the command line:

```
zf create db-table Albums albums
```

The command line tool has now created the file `Albums.php` in the `application/models/DbTable` folder. Inside this file is a class called `Application_Model_DbTable_Albums` within which is set the name of the database table that this class communicates with.

We now need to add some functionality so edit application/models/DbTable/Albums.php and add the methods `getAlbum()`, `addAlbum()`, `updateAlbum()` and `deleteAlbum()` and so that it now looks like this:

zf-tutorial/application/models/DbTable/Albums.php
```php
<?php

class Application_Model_DbTable_Albums extends Zend_Db_Table_Abstract
{
    protected $_name = 'albums';
```

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

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

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

    public 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:

```php
$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 code 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 create 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. We use `Zend_Tool` to create the layout view script file and update application.ini appropriately. Again, from Terminal or the Command Prompt, type the following in your `zf-tutorial` directory.

```
zf enable layout
```

`Zend_Tool` has now created the folder `application/layouts/scripts` and placed a view script called `layout.phtml` into it. It has also updated `application.ini` and added the line `resources.layout.layoutPath = APPLICATION_PATH "/layouts/scripts/"` to the `[production]` section.

At the end of the dispatch cycle, after the controller action methods have finished, `Zend_Layout` will render our layout. `Zend_Tool` provides a very basic layout file that just displays the content of the action's view script. We will augment this with the HTML required for this website. Open `layouts.phtml` and replace the code in there with this:

zf-tutorial/application/layouts/scripts/layout.phtml
```php
<?php
$this->headMeta()->appendHttpEquiv('Content-Type', 'text/html;charset=utf-8');
$this->headTitle()->setSeparator(' - ');
$this->headTitle('Zend Framework Tutorial');

echo $this->doctype(); ?>
<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 configure some view helpers for the head section of the web page and then echo out the correct doctype. 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.

We need to set the doctype for the webpage before we render any view scripts. As the action view scripts are rendered earlier and may need to know which doctype is in force. This is especially true for Zend_Form.

To set the doctype we add another line to our application.ini, in the [production] section:

```
resources.view.doctype = "XHTML1_STRICT"
```

The doctype() view helper will now output the correct doctype and components like Zend_Form will generate compatible HTML.

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. Fortunately, a view helper called baseUrl() is available. This helper collects the information we require from the request object and provides us with the bit of the URL that we don't know.

We can now add the CSS file to the <head> section of the application/layouts/scripts/ layout.phtml file and again we use a view helper, headLink():

zf-tutorial/application/layouts/scripts/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()'s prependStylesheet() method, we allow for additional, more specific, CSS files to be added within the controller view scripts which will be rendered within the <head> section after site.css.

Finally, we need some CSS styles, so create a css directory within public/ and add site.css with this code:

zf-tutorial/public/css/site.css

```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 {
```

```css
    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
```php
    ...
    function indexAction()
    {
        $albums = new Application_Model_DbTable_Albums();
        $this->view->albums = $albums->fetchAll();
    }
    ...
```

We instantiate an instance of our table data gateway based model. 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
```php
<?php
$this->title = "My Albums";
$this->headTitle($this->title);
?>
<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 set the title for the page (used in the layout) and also set the title for the `<head>` section using the `headTitle()` view helper which will display in the brower's title bar. We then 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.

If you open http://localhost/zf-tutorial/public/ (or wherever you are following along from!) then you should now see a nice list of albums, something like this:
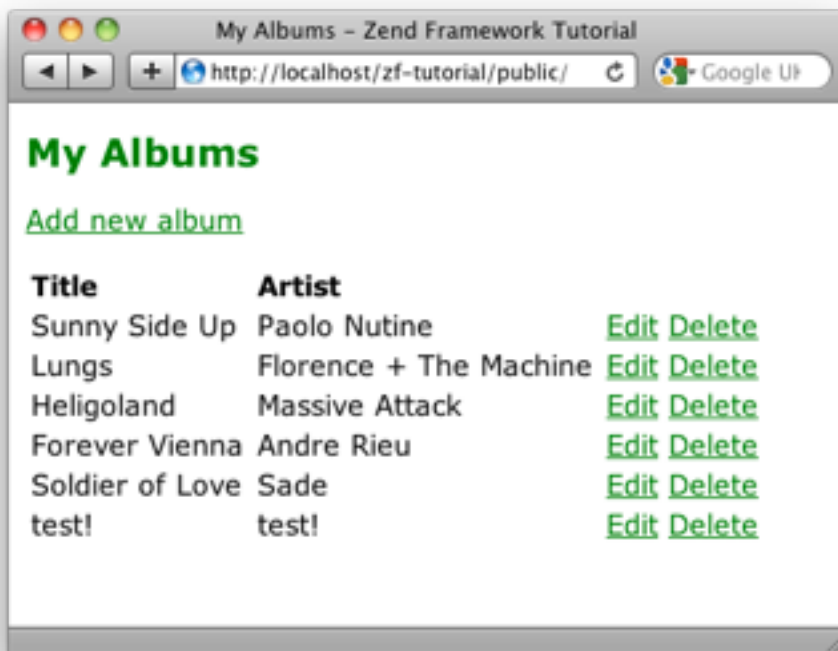
Adding new albums

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 this an

application resource, the class is stored in the `Album.php` file within the `forms` directory. We start by using the `zf` command line script to create the correct file:

```
zf create form Album
```

This creates the file `Album.php` in `application/forms` and includes an `init()` method where we can set up the form and add the elements that we need. Edit `application/forms/Album.php` and remove the comment in the `init()` method and add the following:

zf-tutorial/application/forms/Album.php

```php
<?php

class Application_Form_Album extends Zend_Form
{

    public function init()
    {
        $this->setName('album');

        $id = new Zend_Form_Element_Hidden('id');
        $id->addFilter('Int');

        $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 `init()` method of `Application_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 id, we want to ensure that it is only an integer to prevent potential SQL injection issues. The `Int` filter will do this for us.

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. (the `NotEmpty` validator isn't technically required as it will be automatically added by the system as `setRequired()` has been set to true; it exists here as a demonstration of how to add a validator.)

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

```php
    ...
    function addAction()
    {
        $form = new Application_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 Application_Model_DbTable_Albums();
                $albums->addAlbum($artist, $title);

                $this->_helper->redirector('index');
            } else {
                $form->populate($formData);
            }
        }
    }
    ...
```

Let's examine it in a bit more detail:

```
    $form = new Application_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 Application_Model_DbTable_Albums();
            $albums->addAlbum($artist, $title);
```

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

```
            $this->_helper->redirector('index');
```

After we have saved the new album row, we redirect using the `Redirector` action helper to return to the `index` action (i.e we 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
$this->title = "Add new album";
```

```
$this->headTitle($this->title);
echo $this->form ;
?>
```

As you can see, rendering a form is very simple - we just echo it out, as the form knows how to display itself. You should now be able to use the "Add new album" link on the home page of the application to add a new album record.

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()
{
    $form = new Application_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 Application_Model_DbTable_Albums();
            $albums->updateAlbum($id, $artist, $title);

            $this->_helper->redirector('index');
        } else {
            $form->populate($formData);
        }
    } else {
        $id = $this->_getParam('id', 0);
        if ($id > 0) {
            $albums = new Application_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 Application_Model_DbTable_Albums();
            $form->populate($albums->getAlbum($id));
        }
```

Note that this is done if the request is not a POST, as a POST implies we have filled out the form and want to process it. For the initial display of the form, 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. (Now you know why the `getAlbum()` method in the model returned an array!)

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 = $form->getValue('id');
            $artist = $form->getValue('artist');
            $title = $form->getValue('title');
            $albums = new Application_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
<?php
$this->title = "Edit album";
$this->headTitle($this->title);

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

You should now be able to 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 (Zend_Form is, after all, optional!).

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

zf-tutorial/application/controllers/IndexController.php
```php
    ...
    public function deleteAction()
    {
        if ($this->getRequest()->isPost()) {
            $del = $this->getRequest()->getPost('del');
            if ($del == 'Yes') {
                $id = $this->getRequest()->getPost('id');
                $albums = new Application_Model_DbTable_Albums();
                $albums->deleteAlbum($id);
            }
            $this->_helper->redirector('index');
        } else {
            $id = $this->_getParam('id', 0);
            $albums = new Application_Model_DbTable_Albums();
            $this->view->album = $albums->getAlbum($id);
        }
    }
    ...
```

As with add and edit, we use the Request's isPost() method to determine if we should display the confirmation form or if we should do a deletion. We use the Application_Model_DbTable_Albums model 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
```php
<?php
$this->title = "Delete album";
$this->headTitle($this->title);
```

```
?>
<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>
```

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 email me at rob@akrabat.com!

This tutorial has looked at the basics of using the framework; there are many more components to explore! I have also skipped over a lot of explanation. My website at http://akrabat.com contains many articles on Zend Framework and you should read the manual at http://framework.zend.com/manual 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 ☺