Getting Started with Zend Framework 2

By Rob Allen, www.akrabat.com
Document revision 0.3.6
Copyright © 2011, 2012

This tutorial is intended to give an introduction to using Zend Framework 2 by creating a simple database driven application using the Model-View-Controller paradigm.

Note: This tutorial has been tested on **beta 3 of Zend Framework 2**. It may work on more recent versions, but it won't work with any previous versions.

Requirements

Zend Framework has the following requirements:

- PHP 5.3.3 (or higher)
- A web server supporting mod_rewrite or similar functionality.

Some assumptions

I have assumed that you are running PHP 5.3.6 with the Apache web server and MySQL, accessible via the PDO extension. 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.

The tutorial application

The application that we are going to build is a simple inventory system to display which albums we own. The main page will list our collection and allow us to add, edit and delete CDs. We are going to need four pages in our website:

List of albums	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 started: A skeleton application

There are two ways to get started. If you're comfortable with `git`, then use *Getting started, if you have git*, otherwise use *Getting started, using zip file downloads*.

Getting using zip file downloads

In order to build our application, we are going to start with the ZendSkeletonApplication available from github. Go to https://github.com/zendframework/ZendSkeletonApplication and click the "Zip" button. This will download a file with a name like `zendframework-ZendSkeletonApplication-zf-release-2.0.0beta3-15-gb4b04cd.zip` or similar.

Unzip this file into the directory where you keep all your vhosts and rename the resultant directory to `zf2-tutorial`.

Zend Framework can be downloaded from https://github.com/zendframework/zf2 - look for the "Zip" button to download it. This will download a file with a name like `zendframework-zf2-release-2.0.0beta3-150-gb95fca2.zip` or similar.

Copy the `library/` directory from downloaded ZF2 zip file into your `zf2-tutorial/vendor/ZendFramework` directory, so that your `zf2-tutorial/vendor/ZendFramework/library` directory contains a sub-directory called `Zend`.

Getting started, if you have git

Alternatively, if you have `git` installed, you can do this:

```
git clone --recursive git://github.com/zendframework/ZendSkeletonApplication.git
```

This command will download the skeleton application and the Zend Framework 2 library at the same time.

Virtual host

You now need to create an Apache virtual host for the application and edit your hosts file so that http://zf2-tutorial.localhost will serve `index.php` from the `zf2-tutorial/public` directory.

Setting up the virtual host is usually done within `httpd.conf` or `extra/httpd-vhosts.conf`. (If you are using httpd-vhosts.conf, ensure that this file is included by your main `httpd.conf` file.)

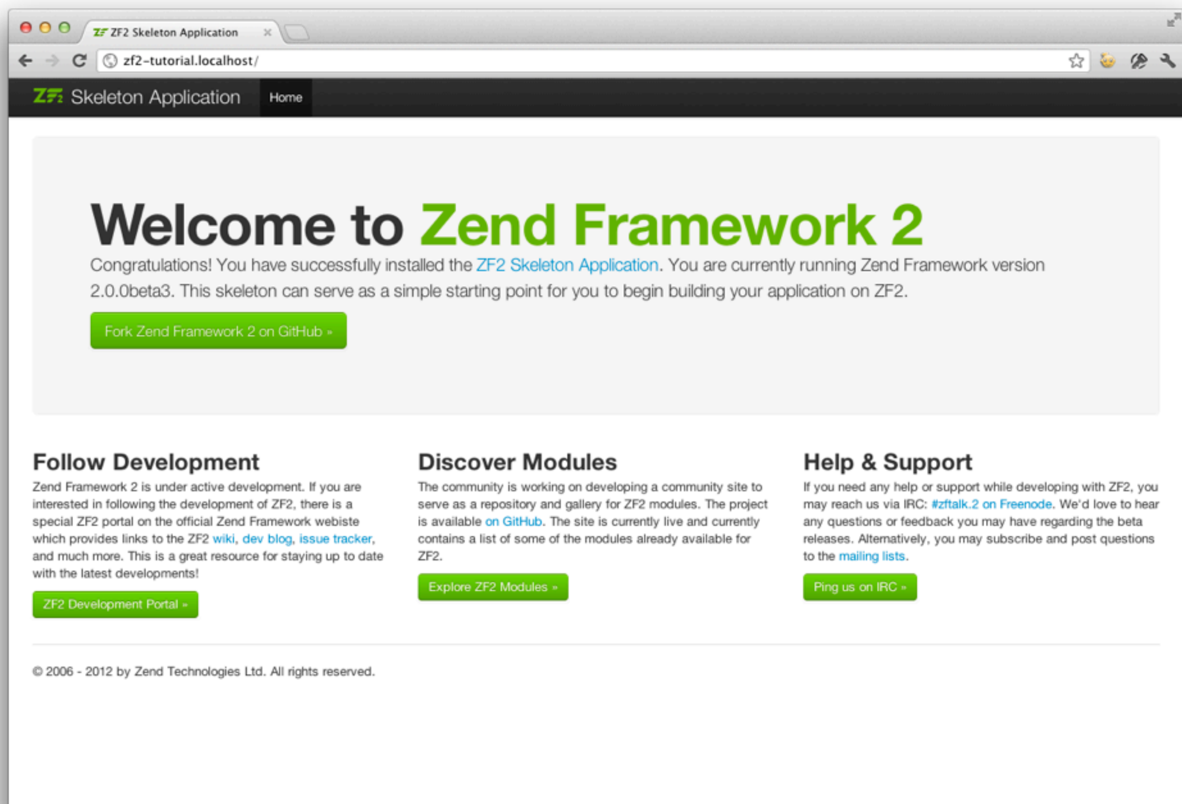
Ensure that `NameVirtualHost` is defined and set to "`*:80`" or similar and then define a virtual host along these lines:

```
<VirtualHost *:80>
    ServerName zf2-tutorial.localhost
```

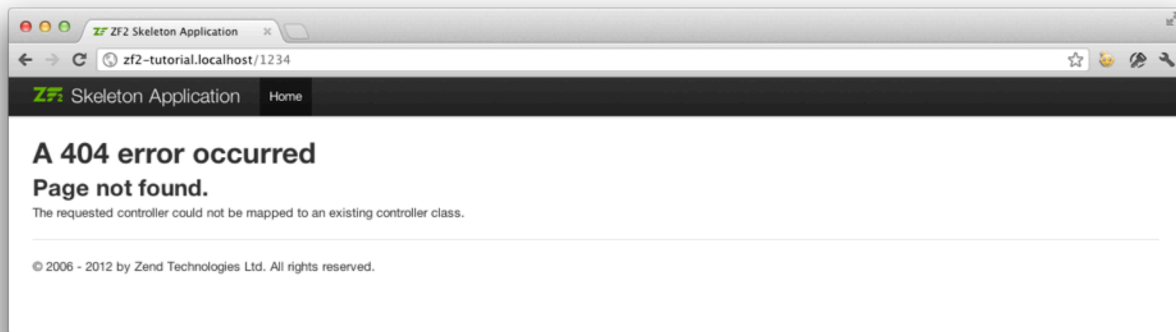
```
    DocumentRoot /path/to/zf-2tutorial/public
    SetEnv APPLICATION_ENV "development"
    <Directory /path/to/zf2-tutorial/public>
        DirectoryIndex index.php
        AllowOverride All
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

Make sure that you update your `/etc/hosts` or `c:\windows\system32\drivers\etc\hosts` file so that `zf2-tutorial.localhost` is mapped to `127.0.0.1`. The website can then be accessed using http://zf2-tutorial.localhost.

If you've done it right, you should see something like this:



To test that your `.htaccess` file is working, navigate to http://zf2-tutorial.localhost/1234 and you should see this:



If you see a standard Apache 404 error, then you need to fix `.htaccess` usage before continuing.

You now have a working skeleton application and we can start adding the specifics for our application.

Modules

Zend Framework 2 uses a module system and you organise your main application-specific code within each module. The Application module provided by the skeleton is used to provide bootstrapping, error and routing configuration to the whole application. It is usually used to provide application level controllers for, say, the home page of an application, but we are not going to use the default one provided in this tutorial as we want our album list to be the home page, which will live in our own module.

We are going to put all our code into the Album module which will contain our controllers, models, forms and views. We'll also need some configuration files. We start with the directories required.

Setting up the Album module

Start by creating a directory called `Album` under with the following subdirectories to hold the module's files:

```
zf2-tutorial/
    /module
        /Album
            /config
            /src
                /Album
                    /Controller
                    /Form
                    /Model
            /view
```

As you can see the `Album` module has separate directories for the different types of files we will have. The PHP files that contain classes within the `Album` namespace live in the `src/Album` directory so that we can have multiple namespaces within our module should we require it.

In order to load and configure a module, Zend Framework 2 has a `ModuleManager`. This will look for `Module.php` in the root of the module directory and expect to find a class called `Album\Module` within it. That is, the classes within a given module will have the namespace of the module's name, which is the directory name of the module.

Create Module.php in the Album module:

module/Album/Module.php

```
<?php

namespace Album;

use Zend\Module\Consumer\AutoloaderProvider;

class Module implements AutoloaderProvider
{
    public function getAutoloaderConfig()
    {
        return array(
            'Zend\Loader\ClassMapAutoloader' => array(
                __DIR__ . '/autoload_classmap.php',
            ),
            'Zend\Loader\StandardAutoloader' => array(
                'namespaces' => array(
                    __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
                ),
            ),
        );
    }

    public function getConfig()
    {
        return include __DIR__ . '/config/module.config.php';
    }
}
```

The ModuleManager will call `getAutoloaderConfig()` for us as we have implemented the `AutoloaderProvider` interface.

Autoloading files

Our `getAutoloaderConfig()` method returns an array that is compatible with ZF2's `AutoloaderFactory`. We configure it so that we add a classmap file to the `ClassmapAutoloader` and also add this module's namespace to the `StandardAutoloader`. The standard autoloader requires a namespace and the path where to find the files for that namespace. It is PSR-0 compliant and so classes map directly to files as per the PSR-0 rules[1].

As we are in development, we don't need to load files via the classmap, so we provide an empty array for the classmap autoloader. Create `autoload_classmap.php` with these contents:

module/Album/autoload_classmap.php:

```
<?php
return array();
```

As this is an empty array, whenever the autoloader looks for a class within the `Album` namespace, it will fall back to the to `StandardAutoloader` for us.

---

[1] https://gist.github.com/1293323

Configuration

Having registered the autoloader, let's have a quick look at the `getConfig()` method in `Album\Module`. This method simply loads the `config/module.config.php` file.

Create the following configuration file for the Album module:

module/Album/config/module.config.php:

```
<?php
return array(
    'di' => array(
        'instance' => array(
            'Zend\View\Resolver\TemplatePathStack' => array(
                'parameters' => array(
                    'paths'  => array(
                        'album' => __DIR__ . '/../view',
                    ),
                ),
            ),
        ),
    ),
);
```

The `di` config parameter is used to configure Zend Framework 2's dependency injection container. To start our configuration file off, we add our view directory to the `TemplatePathStack` resolver configuration. This will allow it to find the view scripts for the `Album` module that are stored in our `views` directory.

Informing the application about our new module

We now need to tell the ModuleManager that this new module exists. This is done in the application's `config/application.config.php` file which is provided by the skeleton application. Update this file so that its `modules` section contains the `Album` module as well so the file now looks like this:

(I've bolded the change required)

config/application.config.php:

```
<?php
return array(
    'modules' => array(
        'Application',
        'Album',
    ),
    'module_listener_options' => array(
        'config_cache_enabled' => false,
        'cache_dir'            => 'data/cache',
        'module_paths' => array(
            './module',
            './vendor',
        ),
    ),
);
```

As you can see, we have added our `Album` module into the list of `modules` after the `Application` module. We have now set up the module ready for putting our custom code into it.

The pages within the website

We are going to build a very simple inventory system to display our album collection. The home page will list our collection and allow us to add, edit and delete albums. Hence the following pages are required:

Home	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.

Before we set up our files, it's important to understand how the framework expects the pages to be organised. The default router configuration is provided in the skeleton application's `module.config.php` file.

Each page of the application is known as an "action" and actions are grouped into "controllers" within "modules". E.g. for a URL of the format `http://zf2-tutorial.localhost/news/view`, the controller is `news` and the action is `view`. The default route configuration provided in the skeleton application will look for the first controller within the list of modules that are registered.

Controllers allow for grouping of related actions. For instance, a `news` controller might have actions of `current`, `archived` and `view`. There is a special action called index which is treated as a default action. That is, for a url such as `http://zf2-tutorial.localhost/news/` the `index` action within the `news` controller will be executed.

As we have four pages that all apply to albums, we will group them in a single controller AlbumController within our `Album` module as four actions. The four actions will be:

Page	*Controller*	*Action*
Home	`AlbumController`	`index`
Add new album	`AlbumController`	`add`
Edit album	`AlbumController`	`edit`
Delete album	`AlbumController`	`delete`

Nice and simple!

Create the controller

We are now ready to set up our controller. In Zend Framework 2, the controller is a class that is generally called `{Controller name}Controller`. Note that `{Controller name}` must start with a capital letter. This class lives in a file called `{Controller name}Controller.php` within the `Controller` directory for the module. In our case that's is `module/Album/src/Album/Controller`. Each action is a public function within the controller class that is named `{action name}Action`. In this case `{action name}` should start with a lower case letter.

Note that this is by convention. Zend Framework 2 doesn't provide many restrictions on controllers other than that they must implement the `Zend\Stdlib\Dispatchable` interface. The framework provides two abstract classes that do this for us: `Zend\Mvc\Controller\ActionController` and `Zend\Mvc`

`\Controller\RestfulController`. We'll be using the standard `ActionController`, but if you're intending to write a RESTful web service, `RestfulController` may be useful.

Let's go ahead and create our controller class:

module/Album/src/Album/Controller/AlbumController.php:

```
<?php

namespace Album\Controller;

use Zend\Mvc\Controller\ActionController,
    Zend\View\Model\ViewModel;

class AlbumController extends ActionController
{
    public function indexAction()
    {
    }

    public function addAction()
    {
    }

    public function editAction()
    {
    }

    public function deleteAction()
    {
    }
}
```

Now that we have the controller class, we need to tell the system how to find it. This is done via a `di` alias in our module.config.php file. Update this file so that it now looks like this:

(I have bolded the part that is new.)

module/Album/config/module.config.php:

```
<?php
return array(
    'di' => array(
        'instance' => array(
            'alias' => array(
                'album' => 'Album\Controller\AlbumController',
            ),
            'Zend\View\Resolver\TemplatePathStack' => array(
                'parameters' => array(
                    'paths'  => array(
                        'album' => __DIR__ . '/../view',
                    ),
                ),
            ),
        ),
    ),
);
```

As you can see, we have added an `alias` section to the `di` config which maps '`album`' to the controller class we want to be loaded. Internally, the dispatcher loads the controller class by asking the dependency injection container (known as a service locator) for the controller name. By providing an alias, we have informed it which file we wish to be loaded when the `album` controller is needed. Be aware that this will probably change in later betas though.

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	*Method called*
http://zf2-tutorial.localhost/album	`Album\Controller\AlbumController::indexAction()`
http://zf2-tutorial.localhost/album/add	`Album\Controller\AlbumController::addAction()`
http://zf2-tutorial.localhost/album/edit	`Album\Controller\AlbumController::editAction()`
http://zf2-tutorial.localhost/album/delete	`Album\Controller\AlbumController::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 scripts

To integrate the view into our application all we need to do is create some view script files. These files will be executed by the `ViewListener` object within the `Application` module and be passed any variables that are returned from the controller action method. These view scripts are stored in our module's `views` directory within a directory named after the controller. Create these four empty files now:

- `module/Album/view/album/index.phtml`
- `module/Album/view/album/add.phtml`
- `module/Album/view/album/edit.phtml`
- `module/Album/view/album/delete.phtml`

We can now start filling everything in, starting with our database and models.

The database

Now that we have the `Album` module set up with controller action methods and view scripts, 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\TableGateway\TableGateway` which is used to find, insert, update and delete rows from a database table.

We are going to use MySQL, via PHP's PDO driver, so create a database called `zf2tutorial`, and run these SQL statements to create the `album` table with some data in it.

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

INSERT INTO album (artist, title)
      VALUES ('The Military Wives', 'In My Dreams');
INSERT INTO album (artist, title)
      VALUES ('Adele', '21');
INSERT INTO album (artist, title)
      VALUES ('Bruce Springsteen', 'Wrecking Ball (Deluxe)');
INSERT INTO album (artist, title)
```

```
      VALUES ('Lana Del Rey', 'Born To Die');
INSERT INTO album (artist, title)
      VALUES ('Gotye', 'Making Mirrors');
```

(The test data chosen, happens to be the Bestsellers on Amazon UK when I wrote this version of this tutorial!)

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

The model files

Zend Framework does not provide a `Zend\Model` component 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. Another is to use an ORM like Doctrine or Propel.

For this tutorial, we are going to create the most simple model that I can think of, by creating an `AlbumTable` class that extends `Zend\Db\TableGateway\TableGateway` where each album object is a `Zend\Db\ResultSet\Row`. This is an implementation of 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 limiting 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\AbstractTable`; *don't do this*!

We choose to extend `Zend\Db\TableGateway\TableGateway` and create our own `AlbumTable` class in the module's `Model` directory like this:

module/Album/src/Album/Model/AlbumTable.php:

```
<?php

namespace Album\Model;

use Zend\Db\TableGateway\TableGateway,
    Zend\Db\Adapter\Adapter,
    Zend\Db\ResultSet\ResultSet;

class AlbumTable extends TableGateway
{
    public function __construct(Adapter $adapter = null, $databaseSchema = null,
        ResultSet $selectResultPrototype = null)
    {
        return parent::__construct('album', $adapter, $databaseSchema,
            $selectResultPrototype);
    }

    public function fetchAll()
    {
        $resultSet = $this->select();
        return $resultSet;
    }

    public function getAlbum($id)
    {
        $id  = (int) $id;
        $rowset = $this->select(array('id' => $id));
        $row = $rowset->current();
        if (!$row) {
```

```
            throw new \Exception("Could not find row $id");
        }
        return $row;
    }

    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, array('id' => $id));
    }

    public function deleteAlbum($id)
    {
        $this->delete(array('id' => $id));
    }

}
```

We have created five helper methods that our application will use to interface to the database table. `fetchAll()` retrieves all albums rows from the database as a `ResultSet`, `getAlbum()` retrieves a single row as an `Row` object, `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.

Using DI to configure the database credentials and inject into the controller

In order to use our model class within our controller, we can take advantage of the fact that the dispatcher uses the dependency injection container to locate and instantiate our controller class. This means that we can use the DI container to create an instance of our `AlbumsTable` class and inject it into our controller. By virtue of the way this works, when instantiating the `AlbumsTable` class, we can also tell the DI container how to connect to the database. This means that we can explicitly state how our database is connected to our model and how our model is connected to our controller in our `module.config.php` file which will make maintenance easier if we ever need to change it.

We start by providing a method in our controller which the DI container can use to inject the `AlbumsTable` in it. This is called `setAlbumsTable()`, so add it now. `AlbumController` now looks like this:

module/Album/src/Album/Controller/AlbumController.php:

```
<?php

namespace Album\Controller;

use Zend\Mvc\Controller\ActionController,
    Zend\View\Model\ViewModel,
    Album\Model\AlbumTable;
```

```
class AlbumController extends ActionController
{
    protected $albumTable;

    public function indexAction()
    {

    }

    public function addAction()
    {

    }

    public function editAction()
    {

    }

    public function deleteAction()
    {

    }

    public function setAlbumTable(AlbumTable $albumTable)
    {
        $this->albumTable = $albumTable;
        return $this;
    }
}
```

As you can see, we have added a new method, `setAlbumTable()` that sets our `$albumTable` protected property and we have also updated the `use` list so that PHP knows about our `AlbumTable` class.

Now that there is a way to set an instance of `Album\Model\AlbumTable` within the `AlbumController`, we can now inform the DI container what needs to be done so that it can call `setAlbumTable()` for us. This is done within the '`di`' section of `module.config.php`. Update `module.config.php` with the following changes (in bold);

module/Album/config/module.config.php:

```
<?php
return array(
    'di' => array(
        'instance' => array(
            'alias' => array(
                'album' => 'Album\Controller\AlbumController',
            ),
            'Album\Controller\AlbumController' => array(
                'parameters' => array(
                    'albumTable' => 'Album\Model\AlbumTable',
                ),
            ),
            'Album\Model\AlbumTable' => array(
                'parameters' => array(
                    'adapter' => 'Zend\Db\Adapter\Adapter',
```

```
                )
            ),
            'Zend\Db\Adapter\Adapter' => array(
                'parameters' => array(
                    'driver' => array(
                        'driver' => 'Pdo',
                        'dsn' => 'mysql:dbname=zf2tutorial;hostname=localhost',
                        'username' => 'rob',
                        'password' => '123456',
                        'driver_options' => array(
                            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
                        ),
                    ),
                )
            ),
            'Zend\View\Resolver\TemplatePathStack' => array(
                'parameters' => array(
                    'paths'  => array(
                        'album' => __DIR__ . '/../view',
                    ),
                ),
            ),
        ),
    ),
);
```

Let's look at this change in a bit more detail.

```
           'Album\Controller\AlbumController' => array(
                'parameters' => array(
                    'albumTable' => 'Album\Model\AlbumTable',
                ),
            ),
```

This section tells the DI container that when it instantiates a new `Album\Controller\AlbumController` class, it needs to find a parameter called `albumTable` either in the constructor or in a `setXyz()` method and pass in an instance of `Album\Model\AlbumTable`. It's important to remember that the DI Container works with parameter names, so the actual name of the set method is unimportant. In this case we have called it `setAlbumTable()`, but we could have called it `setTheModel()` and the DI container would have found it as the parameter name is still `$albumTable`. By extension, if you use the same parameter name in two different set methods then both will get the same instance of the class you have specified in the `di` configuration.

```
            'Album\Model\AlbumTable' => array(
                'parameters' => array(
                    'config' => 'Zend\Db\Adapter\Adapter',
            )),
            'Zend\Db\Adapter\Adapter' => array(
                'parameters' => array(
                    'driver' => array(
                        'driver' => 'Pdo',
                        'dsn' => 'mysql:dbname=zf2tutorial;hostname=localhost',
                        'username' => 'rob',
                        'password' => '123456',
```

```
                        'driver_options' => array(
                            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
                        ),
                    ),
                ),
            ),
```

These two sections work the same way as for the controller. We tell the DI container that there's a `config` parameter to be passed into the `Album\Model\AlbumTable` constructor that is an instance of `Zend\Db\Adapter\Adapter` and we would like that instance to have this `driver` parameter (an array) passed to its constructor. In this way, we have configured the `Pdo` driver without having to pass configuration data down through the class hierarchy or use a registry or a static method.

Now that we have an instance of our model in the controller, we can provide a list of albums when the `index` action is called.

Listing albums

In order to list the albums, we need to retrieve them from the model and pass them to the view. To do this, we fill in `indexAction()` within `AlbumController`. Update the `AlbumController`'s `indexAction()` like this:

module/Album/src/Album/AlbumController.php:

```
/...
    public function indexAction()
    {
        return new ViewModel(array(
            'albums' => $this->albumTable->fetchAll(),
        ));
    }
/...
```

With Zend Framework 2, in order to set variables in the view, we return a `ViewModel` instance where the first parameter of the constructor is an array from the action containing data we need. These are then automatically passed to the view script. The `ViewModel` object also allows us to change the view script that is used, but the default is to use {controller name}/{action name}. We can now fill in the `index.phtml` view script:

module/Album/view/album/index.phtml:

```
<?php
$title = 'My albums';
$this->headTitle($title);
?>
<h1><?php echo $this->escape($title); ?></h1>
<p><a href="<?php echo $this->url('default', array('controller'=>'album',
        'action'=>'add'));?>">Add new album</a></p>

<table class="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('default',
            array('controller'=>'album','action'=>'edit'))
            . '?id=' . $album->id ;?>">Edit</a>
        <a href="<?php echo $this->url('default',
            array('controller'=>'album','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 browser's title bar. We then create a link to add a new album.

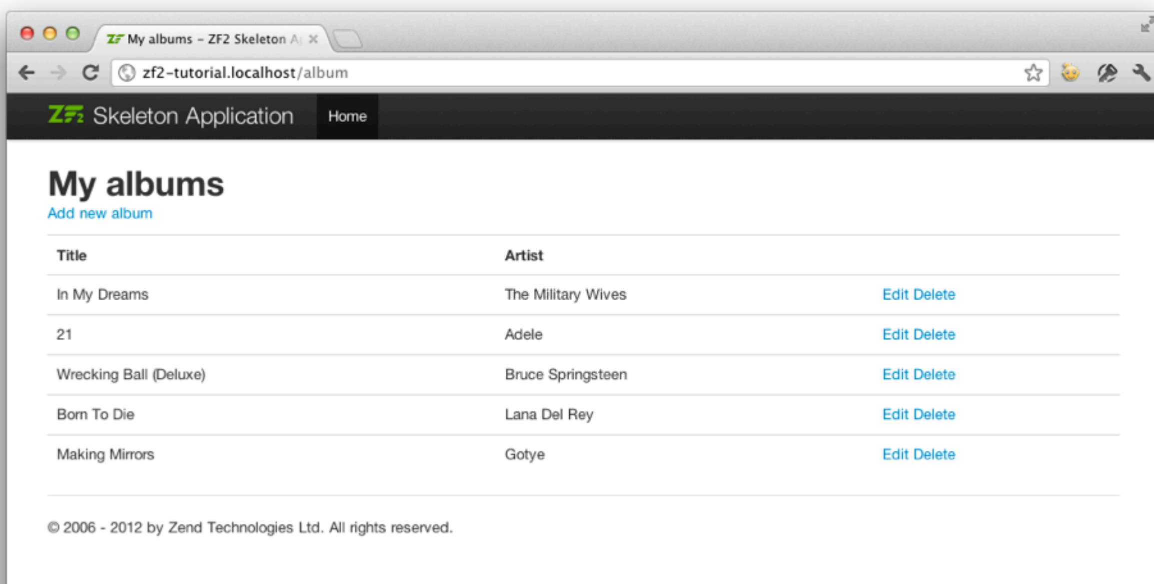
We iterate over the `$albums` that we assigned from the controller action. The Zend Framework 2 view system automatically ensures that these variables are extracted into the scope of the view script, so that we don't have to worry about prefixing them with `$this->` as we used to have to do with Zend Framework 1, however you can do so if you wish.

The `url()` view helper is provided by Zend Framework 2 and is used to create the links we need.

We then create a 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.

Note that we always use the `escape()` view helper to help protect ourselves from XSS vulnerabilities.

If you open http://zf2-tutorial.localhost/album you should see this:



Styling

We've picked up the SkeletonApplication's which is fine, but we need to change the title and and remove the copyright message. This is done in the `Application` module's `layout.phtml` view script:

module/Application/view/layouts/layout.phtml:

Look for:

```
<?php echo $this->headTitle('ZF2 Skeleton Application') ?>
```

and change to:

```
<?php echo $this->headTitle('ZF2 Tutorial') ?>
```

Also look for:

```
<a class="brand" href="<?php echo $this->url('home') ?>">Skeleton Application</
a>
```

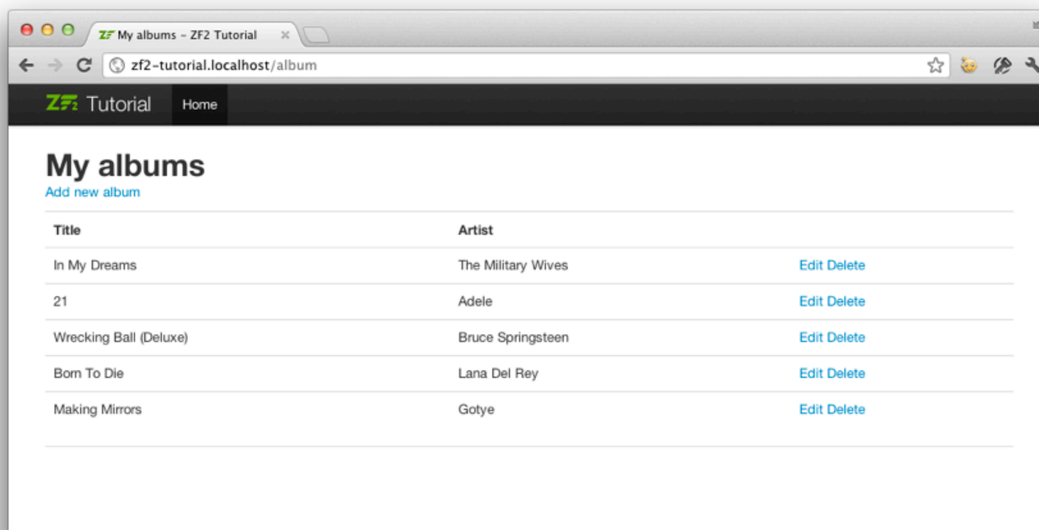
and change to:

```
<a class="brand" href="<?php echo $this->url('home') ?>">Tutorial</a>
```

Also, remove:

```
<p>&copy; 2006 - <?php echo date('Y') ?> by Zend Technologies Ltd. All rights
reserved.</p>
```

The page now looks ever so slightly better now!



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 `Album\Form\AlbumForm` that extends from Zend\Form to define our form. The class is stored in the `AlbumForm.php` file within the `module/Album/src/Album/Form` directory.

Note: The `Zend\Form` in beta 3 is just a name-spaced version of ZF1's `Zend_Form`. Expect it to change before the final release of Zend Framework 2.

Create this file file now:

module/Album/src/Album/Form/AlbumForm.php:

```
<?php
namespace Album\Form;

use Zend\Form\Form,
    Zend\Form\Element;

class AlbumForm extends Form
{
    public function init()
    {
        $this->setName('album');

        $id = new Element\Hidden('id');
        $id->addFilter('Int');

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

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

        $submit = new Element\Submit('submit');
        $submit->setAttrib('id', 'submitbutton');

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

Within the `init()` method 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 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 `AlbumController`'s `addAction()`:

module/Album/src/Album/AlbumController.php:

```
//...
use Zend\Mvc\Controller\ActionController,
    Zend\View\Model\ViewModel,
    Album\Model\AlbumTable,
    Album\Form\AlbumForm;
//...
    public function addAction()
    {
        $form = new AlbumForm();
        $form->submit->setLabel('Add');

        $request = $this->getRequest();
        if ($request->isPost()) {
            $formData = $request->post()->toArray();
            if ($form->isValid($formData)) {
                $artist = $form->getValue('artist');
                $title  = $form->getValue('title');
                $this->albumTable->addAlbum($artist, $title);

                // Redirect to list of albums
                return $this->redirect()->toRoute('default', array(
                    'controller' => 'album',
                    'action'     => 'index',
                ));
            }
        }

        return array('form' => $form);
    }
//...
```

After adding the `AlbumForm` to the `use` list, we implement `addAction()`. Let's look at the `addAction()` code in a little more detail:

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

We instantiate `AlbumForm` and set the label on the submit button to "Add". We do this here as we'll want to re-use the form when editing an album.

```
        $request = $this->getRequest();
        if ($request->isPost()) {
            $formData = $request->post()->toArray();
            if ($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 `post()->toArray()` and check to see if it is valid using the `isValid()` member function of the form.

```
                $artist = $form->getValue('artist');
                $title  = $form->getValue('title');
```

```
                $this->albumTable->addAlbum($artist, $title);
```

If the form is valid, then we grab the data from the form and store to the model using `addAlbum()`.

```
                // Redirect to list of albums
                return $this->redirect()->toRoute('default', array(
                    'controller' => 'album',
                    'action'     => 'index',
                ));
```

After we have saved the new album row, we redirect back to the list of albums using the `Redirect` controller plugin.

```
        return array('form' => $form);
```

Finally, we return the variables that we want assigned to the view. In this case, just the form object. Note that Zend Framework 2 also allows you to simply return an array containing the variables to be assigned to the view and it will create a ViewModel behind the scenes for you. This saves a little typing.

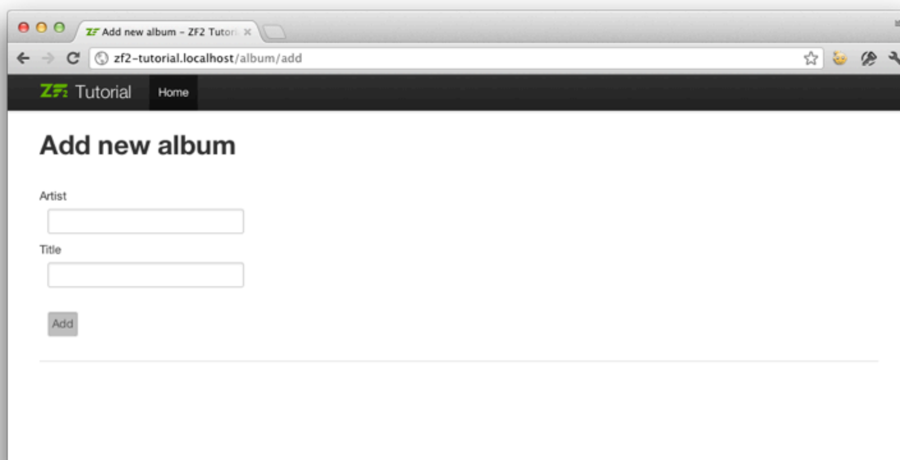
We now need to render the form in the `add.phtml` view script:

module/Album/view/album/add.phtml:

```
<?php
$title = 'Add new album';
$this->headTitle($title);
?>
<h1><?php echo $this->escape($title); ?></h1>

<?php
$form->setAction($this->url('default',
        array('controller' => 'album', 'action' => 'add'))
    )->setView($this);
echo $form;
```

Again, we display a title as before and then we render the 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. This time we use `editAction()` in the `AlbumController`:

module/Album/src/Album/AlbumController.php:

```
//...
    public function editAction()
    {
        $form = new AlbumForm();
        $form->submit->setLabel('Edit');

        $request = $this->getRequest();
        if ($request->isPost()) {
            $formData = $request->post()->toArray();
            if ($form->isValid($formData)) {
                $id     = $form->getValue('id');
                $artist = $form->getValue('artist');
                $title  = $form->getValue('title');

                if ($this->albumTable->getAlbum($id)) {
                    $this->albumTable->updateAlbum($id, $artist, $title);
                }

                // Redirect to list of albums
                return $this->redirect()->toRoute('default', array(
                    'controller' => 'album',
                    'action'     => 'index',
                ));
            }
        } else {
            $id = $request->query()->get('id', 0);
            if ($id > 0) {
                $album = $this->albumTable->getAlbum($id);
                $form->populate($album->getArrayCopy());
            }
        }

        return array('form' => $form);
    }
//...
```

This code should look comfortably familiar. 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 = $request->query()->get('id', 0);
             if ($id > 0) {
                 $album = $this->albumTable->getAlbum($id);
                 $form->populate($album->getArrayCopy());
             }
```

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 `query()->get()` 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 returns 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:

```
            if ($this->albumTable->getAlbum($id)) {
                $this->albumTable->updateAlbum($id, $artist, $title);
            }
```

Note that we check that an album exists with that id exists before calling `updateAlbum()`. Arguably, this check should be within the model method itself.

The view template, `edit.phtml`, looks very similar to the one for adding an album:

module/Album/view/album/edit.phtml:

```
<?php
$title = 'Edit album';
$this->headTitle($title);
?>
<h1><?php echo $this->escape($title); ?></h1>

<?php
$form->setAction($this->url('default',
        array('controller' => 'album', 'action' => 'edit'))
    )->setView($this);
echo $form;
```

The only changes are to use the 'Edit Album' title and set the form's action to the 'edit' action too.

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 `AlbumController::deleteAction()`:

module/Album/src/Album/AlbumController.php:

```
//...
    public function deleteAction()
    {
        $request = $this->getRequest();
        if ($request->isPost()) {
            $del = $request->post()->get('del', 'No');
            if ($del == 'Yes') {
                $id = $request->post()->get('id');
                $this->albumTable->deleteAlbum($id);
```

```
            }

            // Redirect to list of albums
            return $this->redirect()->toRoute('default', array(
                'controller' => 'album',
                'action'     => 'index',
            ));
        }

        $id = $request->query()->get('id', 0);
        return array('album' => $this->albumTable->getAlbum($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 model to actually delete the row using the `deleteAlbum()` method. If the request is not a POST, then as with edit, we look for an id parameter and retrieve the correct database record and assign to the view.

The view script is a simple form:

module/Album/view/album/delete.phtml:

```
<?php
$title = 'Delete album';
$this->headTitle($title);
?>
<h1><?php echo $this->escape($title); ?></h1>

<p>Are you sure that you want to delete
  '<?php echo $this->escape($album['title']); ?>' by
  '<?php echo $this->escape($album['artist']); ?>'?
</p>
<?php
$url = $this->url('default',
    array('controller' => 'album', 'action' => 'delete')); ?>
<form action="<?php echo $url; ?>" method="post">
<div>
  <input type="hidden" name="id" value="<?php echo (int)$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.

Ensuring that the home page displays the list of albums

One final point. At the moment, the home page, http://zf2-tutorial.localhost/ doesn't display the list of albums. This is due to a route set up in the `Application` module's `module.config.php`. To change it, open `module/Application/config/module.config.php` and find the `home` route:

```
        'home' => array(
            'type' => 'Zend\Mvc\Router\Http\Literal',
```

```
            'options' => array(
                'route'    => '/',
                'defaults' => array(
                    'controller' => 'Application\Controller\IndexController',
                    'action'     => 'index',
                ),
            ),
        ),
```

change the `controller` from `index` to `album`:

```
        'home' => array(
            'type' => 'Zend\Mvc\Router\Http\Literal',
            'options' => array(
                'route'    => '/',
                'defaults' => array(
                    'controller' => 'Album\Controller\AlbumController',
                    'action'     => 'index',
                ),
            ),
        ),
```

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 2. I hope that you found it useful. If you find anything that's wrong, please email me at rob@akrabat.com!

As Zend Framework 2 is still in beta, there aren't that many articles about it yet. You should however check out the manual at http://packages.zendframework.com/docs/latest/manual/en/index.html.

My website at http://akrabat.com contains many articles on Zend Framework and I have started covering Zend Framework 2 there too.