# Getting Started with Zend Framework 2

*By Rob Allen, www.akrabat.com*

*Document revision 0.6.0*

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. By the end you will have a working ZF2 application and you can then poke around the code to find out more about how it all works and fits together.

**Note:** This tutorial has been tested on **RC1 of Zend Framework 2**. It may work on more recent versions, but it won't work with any previous versions.

## Some assumptions

This tutorial assumes that you are running PHP 5.3.10 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 | |

| Field name | Type | Null? | Notes |
|---|---|---|---|
| title | varchar(100) | No | |

## Getting started: A skeleton application

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-2.0.0beta5-2-gc2c7315.zip` or similar.

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

ZendSkeletonApplication is set up to use Composer (http://getcomposer.org) to resolve its dependencies. In this case, the dependency is Zend Framework 2 itself.

To install Zend Framework 2 into our application we simply type:

```
php composer.phar self-update
php composer.phar install
```

from the `zf2-tutorial` folder. This takes a while. You should see an output like:

```
Installing dependencies from lock file
  - Installing zendframework/zendframework (dev-master)
    Cloning 18c8e223f070deb07c17543ed938b54542aa0ed8

Generating autoload files
```

We can now move on to the virtual host.

## 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.

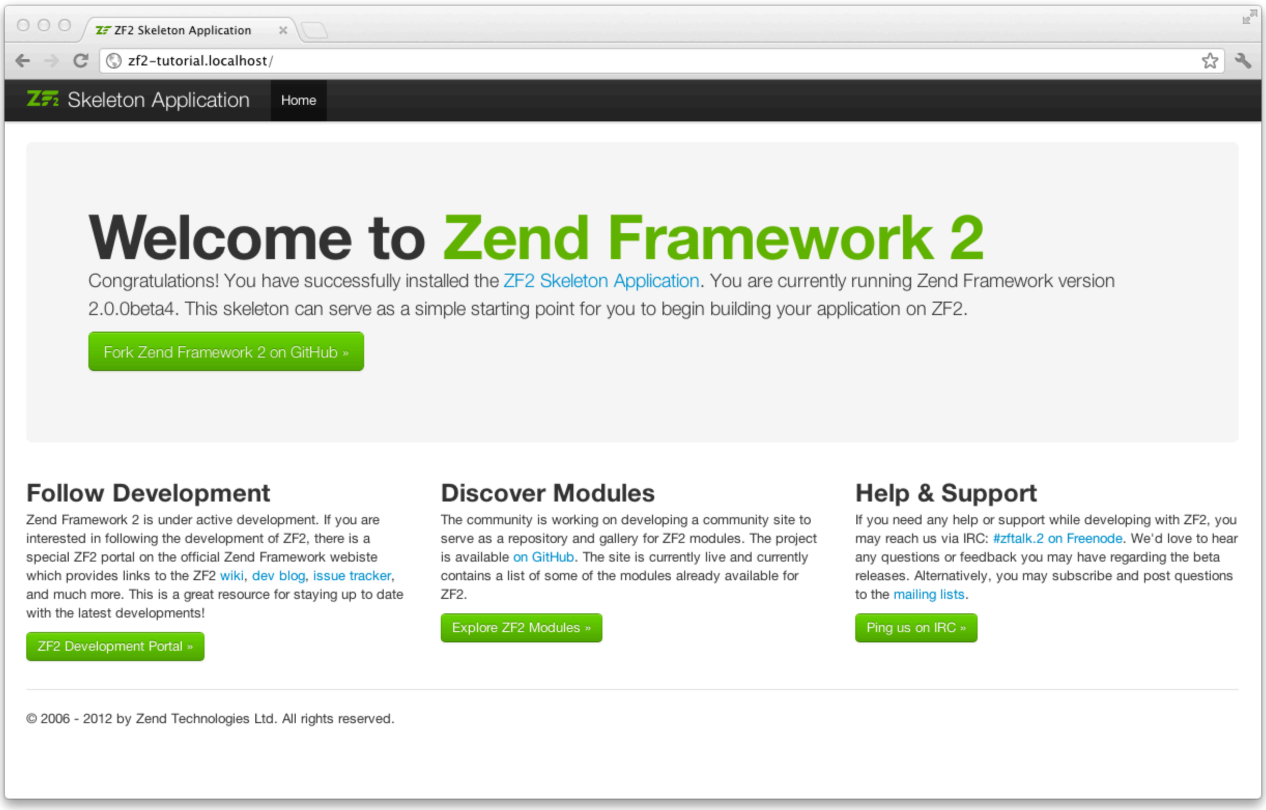
Setting up the virtual host is usually done within `httpd.conf` or `extra/httpd-vhosts.conf`. (If you are using httpd-vhosts.conf, ensure that this file is included by your main `httpd.conf` file.)

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:



ZF2 Skeleton Application

zf2-tutorial.localhost/

Skeleton Application Home

Welcome to Zend Framework 2

Congratulations! You have successfully installed the ZF2 Skeleton Application. You are currently running Zend Framework version 2.0.0beta4. This skeleton can serve as a simple starting point for you to begin building your application on ZF2.

Fork Zend Framework 2 on GitHub »

Follow Development

Zend Framework 2 is under active development. If you are interested in following the development of ZF2, there is a special ZF2 portal on the official Zend Framework webiste which provides links to the ZF2 wiki, dev blog, issue tracker, and much more. This is a great resource for staying up to date with the latest developments!

ZF2 Development Portal »

Discover Modules

The community is working on developing a community site to serve as a repository and gallery for ZF2 modules. The project is available on GitHub. The site is currently live and currently contains a list of some of the modules already available for ZF2.

Explore ZF2 Modules »

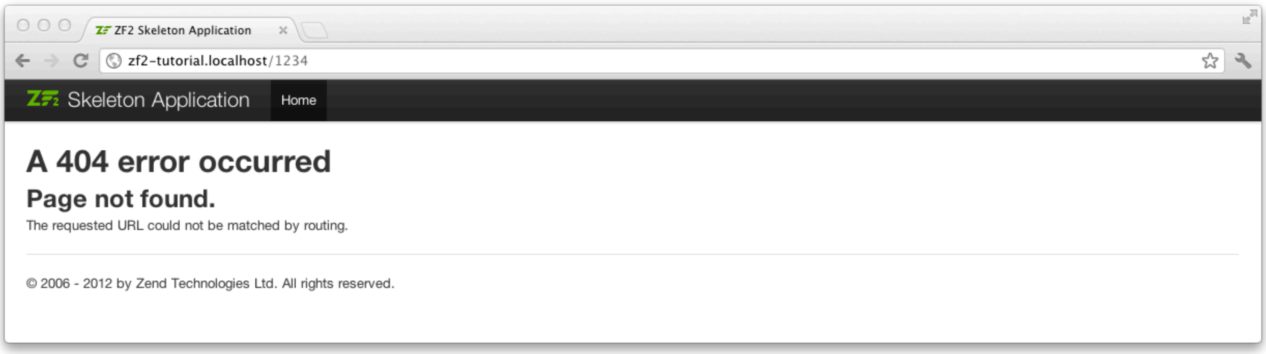
Help & Support

If you need any help or support while developing with ZF2, you may reach us via IRC: #zftalk.2 on Freenode. We'd love to hear any questions or feedback you may have regarding the beta releases. Alternatively, you may subscribe and post questions to the mailing lists.

Ping us on IRC »

© 2006 - 2012 by Zend Technologies Ltd. All rights reserved.

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



ZF2 Skeleton Application

zf2-tutorial.localhost/1234

Skeleton Application Home

A 404 error occurred

Page not found.

The requested URL could not be matched by routing.

© 2006 - 2012 by Zend Technologies Ltd. All rights reserved.

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, along with configuration. We'll also tweak the Application module as required.

Let's 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
                    /album
                        /album
```

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. The `view` directory also has a sub-folder called `album` for our module's view scripts.

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 (`module/Album`) 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;

class Module
{
    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()` and `getConfig()` automatically for us.

**Autoloading files**

Our `getAutoloaderConfig()` method returns an array that is compatible with ZF2's `AutoloaderFactory`. We configure it so that we add a class map 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.

*Note* that as we are using Composer, as an alternative, you could not implement `getAutoloaderConfig()` and instead add `"Application": "module/Application/src"` to the `psr-0` key in `composer.json`. If you go this way, then you need to run `php composer.phar update` to update the composer autoloading files.

## 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(
    'controllers' => array(
        'invokables' => array(
            'Album\Controller\Album' => 'Album\Controller\AlbumController',
        ),
    ),

    'view_manager' => array(
        'template_path_stack' => array(
            'album' => __DIR__ . '/../view',
        ),
    ),
);
```

The  config information passed to the relevant components by the `ServiceManager`. We need two initial sections: `controller` and `view_manager`. The `controller` section provides a list of all the controllers provided by the module. We will need one controller, `AlbumController`, which we'll reference as `Album\Controller\Album`. We call it this as the key must be unique across all modules, so we prefix with our module name.

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

and within the `view_manager` section we add our view directory to the `TemplatePathStack` configuration. This will allow it to find the view scripts for the `Album` module that are stored in our `views/album` 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_glob_paths'    => array(
            'config/autoload/{,*.}{global,local}.php',
        ),
        '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. Each page of the application is known as an action and actions are grouped into controllers within modules. Hence, you would generally group related actions into a controller; for instance, a `news` controller might have actions of `current`, `archived` and `view`.

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` |

The mapping of a URL to a particular action is done using routes that are defined in the module's `module.config.php` file. We will add a route for our album actions. This is the updated config file with the new code added in bold.

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

```
<?php
return array(
    'controllers' => array(
        'invokables' => array(
            'Album\Controller\Album' => 'Album\Controller\AlbumController',
        ),
    ),

    'router' => array(
        'routes' => array(
            'album' => array(
                'type'    => 'segment',
                'options' => array(
                    'route'    => '/album[/:action][/:id]',
                    'constraints' => array(
                        'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
                        'id'     => '[0-9]+',
                    ),
                    'defaults' => array(
                        'controller' => 'Album\Controller\Album',
                        'action'     => 'index',
                    ),
                ),
            ),
        ),
    ),

    'view_manager' => array(
        'template_path_stack' => array(
            'album' => __DIR__ . '/../view',
        ),
    ),
);
```

The name of the route is 'album' and has a type of segment. The segment route allows us to specify placeholders in the URL pattern (route) that will be mapped to named parameters in the matched route. In this case, the route is **`'/album[/:action][/:id]'`** which will match any URL that starts with /album. The next segment will be an optional action name and then finally the next segment will be mapped to an optional id. The square brackets indicate that a segment is optional. The constraints section allows us to ensure that the characters within a segment are as expected, so we have limited actions to starting with a letter and then subsequent characters only being alphanumeric, underscore or hyphen. We also limit the id to a number.

This route allows us to have the following URLS:

| *URL* | *Page* | *Action* |
|---|---|---|
| /album | Home (list of albums) | `index` |
| /album/add | Add new album | `add` |
| /album/edit/2 | Edit album with an id of 2 | `edit` |
| /album/delete/4 | Delete album with an id of 4 | `delete` |

## 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\AbstractActionController;
use Zend\View\Model\ViewModel;

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

    public function addAction()
    {
    }

    public function editAction()
    {
    }
    public function deleteAction()
    {
    }
}
```

Note that we have already informed the module about our controller in the 'controller' section of `config/module.config.php`.

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 and the model layer.

## Initialise 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/album/index.phtml`
- `module/Album/view/album/album/add.phtml`
- `module/Album/view/album/album/edit.phtml`
- `module/Album/view/album/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 a very simple model by creating an `AlbumTable` class that extends `Zend\Db\TableGateway\TableGateway` where each album object is an `Album` object (known as an *entity*). 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\TableGateway\AbstractTableGateway`. *Don't do this*!

Let's start with our `Album` entity class within the `Model` directory:

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

```
<?php

namespace Album\Model;

class Album
{
    public $id;
    public $artist;
    public $title;

    public function exchangeArray($data)
    {
        $this->id     = (isset($data['id'])) ? $data['id'] : null;
        $this->artist = (isset($data['artist'])) ? $data['artist'] : null;
        $this->title  = (isset($data['title'])) ? $data['title'] : null;
    }
}
```

Our Album entity object is a simple PHP class. In order to work with `Zend\Db`'s `AbstractTableGateway` class, we need to implement the exchangeArray() method. This method simply copies the data from the passed in array to our entity's properties. We will add an input filter for use with our form later.

Next, we extend `Zend\Db\TableGateway\AbstractTableGateway` and create our own `AlbumTable` class in the module's `Model` directory like this:

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

```php
<?php

namespace Album\Model;

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

class AlbumTable extends AbstractTableGateway
{
    protected $table ='album';

    public function __construct(Adapter $adapter)
    {
        $this->adapter = $adapter;
        $this->resultSetPrototype = new ResultSet();
        $this->resultSetPrototype->setArrayObjectPrototype(new Album());

        $this->initialize();
    }

    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 saveAlbum(Album $album)
    {
        $data = array(
            'artist' => $album->artist,
            'title'  => $album->title,
        );

        $id = (int)$album->id;
        if ($id == 0) {
            $this->insert($data);
        } else {
            if ($this->getAlbum($id)) {
                $this->update($data, array('id' => $id));
            } else {
                throw new \Exception('Form id does not exist');
            }
        }
    }

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

}
```

There's a lot going on here. Firstly we set the protected property `$table` to the name of the database table, 'album' in this case. We then write a constructor that takes a database adapter as its only parameter and assigns it to the `adapter` property of our class. We then need to tell the table gateway's result set that whenever it creates a new row object, it should use an `Album` object to do so. The TableGateway classes use the prototype pattern for creation of result sets and entities. This means that instead of instantiating when required, the system clones a previously instantiated object. See http://ralphschindler.com/2012/03/09/php-constructor-best-practices-and-the-prototype-pattern for more details.

We then create some helper methods that our application will use to interface with the database table. `fetchAll()` retrieves all albums rows from the database as a `ResultSet`, `getAlbum()` retrieves a single row as an `Album` object, `saveAlbum()` either creates a new row in the database or updates a row that already exists and `deleteAlbum()` removes the row completely. The code for each of these methods is, hopefully, self-explanatory.

## Using ServiceManager to configure the database credentials and inject into the controller

In order to always use the same instance of our `AlbumTable`, we will use the `ServiceManager` to define how to create one. This is most easily done in the `Module` class where we create a method called `getServiceConfi()` which is automatically called by the `ModuleManager` and applied to the `ServiceManager`. We'll then be able to retrieve it in our controller when we need it.

To configure the `ServiceManager` we can either supply the name of the class to be instantiated or a factory (closure or callback) that instantiates the object when the `ServiceManager` needs it. We start by implementing `getServiceConfig()` to provide a factory that creates an `AlbumTable`. Add this method to the bottom of the `Module` class.

**module/Album/Module.php:**

```
<?php

namespace Album;

use Album\Model\AlbumTable;

class Module
{
    // getAutoloaderConfig() and getConfig() methods here

    public function getServiceConfig()
    {
        return array(
            'factories' => array(
                'Album\Model\AlbumTable' =>  function($sm) {
                    $dbAdapter = $sm->get('Zend\Db\Adapter\Adapter');
                    $table = new AlbumTable($dbAdapter);
                    return $table;
                },
            ),
        );
    }
}
```

This method returns an array of `factories` that are all merged together by the `ModuleManager` before passing to the `ServiceManager`. We also need to configure the Service Manager so that it knows how to get a `Zend\Db\Adapter\Adapter`. This is done using a factory called `Zend\Db\Adapter\AdapterServiceFactory` which we can configure within the merged config system. Zend Framework 2's `ModuleManager` merges all the configuration from each module's `module.config.php` file and then merges in the files in `config/autoload` (*.global.php and then *.local.php files). We'll add our database configuration information to `global.php` which you should commit to your version control system.You can use local.php (outside of the VCS) to store the credentials for your database if you want to.

**config/autoload/global.php:**

```
return array(
    'db' => array(
        'driver' => 'Pdo',
        'dsn'            => 'mysql:dbname=zf2tutorial;hostname=localhost',
        'driver_options' => array(
            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
        ),
    ),
    'service_manager' => array(
        'factories' => array(
            'Zend\Db\Adapter\Adapter'
                    => 'Zend\Db\Adapter\AdapterServiceFactory',
        ),
    ),
);
```

You should put your database credentials in `config/autoloader/local.php` so that they are not in the git repository (as local.php is ignored):

**config.autoload/local.php:**

```
return array(
    'db' => array(
        'username' => 'YOUR USERNAME HERE',
        'password' => 'YOUR PASSWORD HERE',
    ),
);
```

Now that the ServiceManager can create an `AlbumTable` instance for us, we can add a method to the controller to retrieve it. Add `getAlbumTable()` to the `AlbumController` class:

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

```
    public function getAlbumTable()
    {
        if (!$this->albumTable) {
            $sm = $this->getServiceLocator();
            $this->albumTable = $sm->get('Album\Model\AlbumTable');
        }
        return $this->albumTable;
    }
```

You should also add:

```
    protected $albumTable;
```

to the top of the class.

We can now call `getAlbumTable()` from within our controller whenever we need to interact with our model. Let's start with 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/Controller/AlbumController.php:**

```
/...
    public function indexAction()
    {
        return new ViewModel(array(
            'albums' => $this->getAlbumTable()->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/album/index.phtml:**

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

<table class="table">
<tr>
    <th>Title</th>
    <th>Artist</th>
    <th>&nbsp;</th>
</tr>
<?php foreach($albums as $album) : ?>
<tr>
    <td><?php echo $this->escapeHtml($album->title);?></td>
    <td><?php echo $this->escapeHtml($album->artist);?></td>
    <td>
        <a href="<?php echo $this->url('album',
            array('action'=>'edit', 'id' => $album->id));?>">Edit</a>
        <a href="<?php echo $this->url('album',
            array('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.

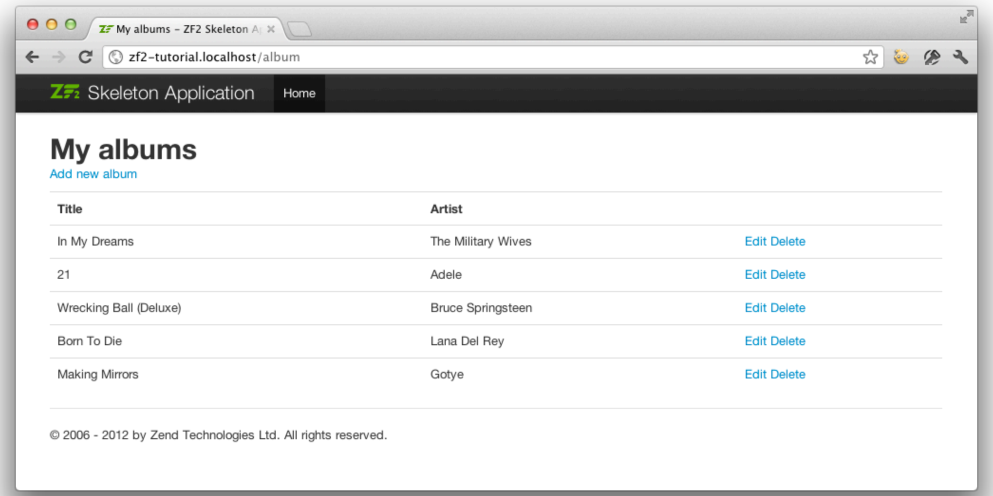
The `url()` view helper is provided by Zend Framework 2 and is used to create the links we need. The first parameter to `url()` is the route name we wish to use for construction of the URL and then the second parameter is an array of all the variables to fit into the placeholders to use. In this case we use our 'album' route which is set up to accept two placeholder variables: `action` and `id`.

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.

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 `escapeHtml()` view helper to help protect ourselves from XSS vulnerabilities.

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



My albums

Add new album

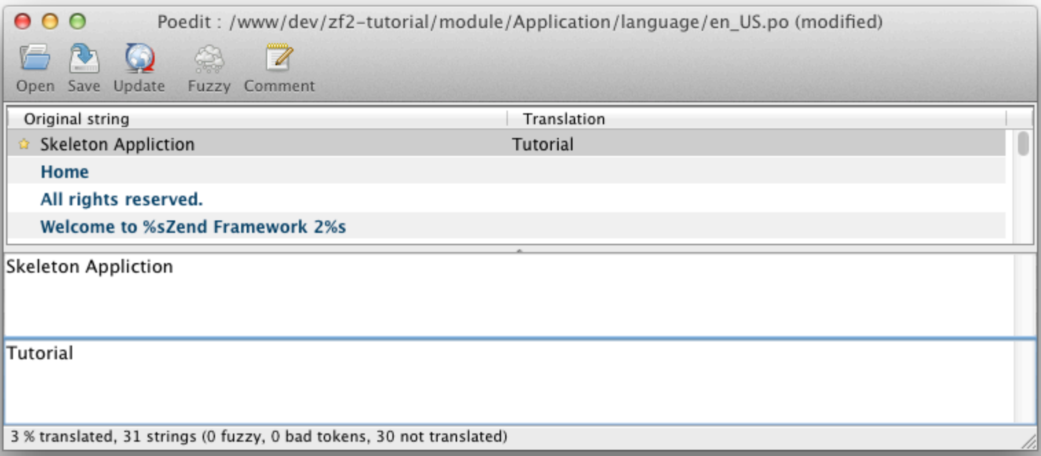
| Title | Artist | |
|---|---|---|
| In My Dreams | The Military Wives | Edit Delete |
| 21 | Adele | Edit Delete |
| Wrecking Ball (Deluxe) | Bruce Springsteen | Edit Delete |
| Born To Die | Lana Del Rey | Edit Delete |
| Making Mirrors | Gotye | Edit Delete |

© 2006 - 2012 by Zend Technologies Ltd. All rights reserved.

## Styling

We've picked up the SkeletonApplication's which is fine, but we need to change the title and and remove the copyright message.

The ZendSkeletonApplication is set up to use `Zend\I18n`'s translation functionality for all the text. It uses .po files that live in application/langauge and you need to use poedit (http://www.poedit.net/download.php/) to change the text. Start poedit and open `application/language/en_US.po`. Click on "Skeleton Application" in the list of Original strings and then type in "Tutorial" as the translation.



Poedit : /www/dev/zf2-tutorial/module/Application/language/en_US.po (modified)

| Original string | Translation |
|---|---|
| Skeleton Appliction | Tutorial |
| Home | |
| All rights reserved. | |
| Welcome to %sZend Framework 2%s | |

Skeleton Appliction

Tutorial

3 % translated, 31 strings (0 fuzzy, 0 bad tokens, 30 not translated)

Press Save in the toolbar and poedit will create an en_US.mo file for us.

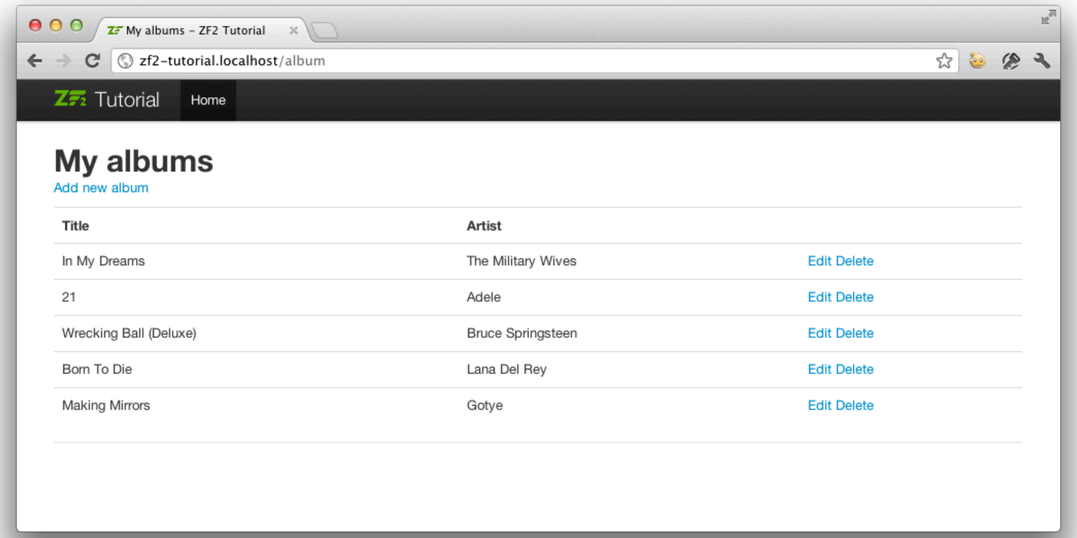
To remove the copyright message, we need to edit the `Application` module's `layout.phtml` view script:

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

Remove this line:

```
<p>&copy; 2005 - 2012 by Zend Technologies Ltd. <?php echo $this->translate('All
rights reserved.') ?></p>
```

The page now looks ever so slightly better now!



My albums

Add new album

| Title | Artist | |
|---|---|---|
| In My Dreams | The Military Wives | Edit Delete |
| 21 | Adele | Edit Delete |
| Wrecking Ball (Deluxe) | Bruce Springsteen | Edit Delete |
| Born To Die | Lana Del Rey | Edit Delete |
| Making Mirrors | Gotye | Edit Delete |

## 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 manages the form and for validation, we add a `Zend\InputFilter` to our `Album` entity. We start by creating a new class `Album\Form\AlbumForm` that extends from `Zend\Form\Form` to define our form. The class is stored in the `AlbumForm.php` file within the `module/Album/src/Album/Form` directory.

Create this file file now:

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

```
<?php
namespace Album\Form;

use Zend\Form\Form;

class AlbumForm extends Form
{
    public function __construct($name = null)
    {
        // we want to ignore the name passed
        parent::__construct('album');

        $this->setAttribute('method', 'post');
        $this->add(array(
            'name' => 'id',
            'attributes' => array(
                'type'  => 'hidden',
            ),
        ));

        $this->add(array(
            'name' => 'artist',
            'attributes' => array(
                'type'  => 'text',
            ),
            'options' => array(
                'label' => 'Artist',
            ),
        ));

        $this->add(array(
            'name' => 'title',
            'attributes' => array(
                'type'  => 'text',
            ),
            'options' => array(
                'label' => 'Title',
            ),
        ));

        $this->add(array(
            'name' => 'submit',
            'attributes' => array(
                'type'  => 'submit',
                'value' => 'Go',
                'id' => 'submitbutton',
            ),
        ));

    }
}
```

Within the constructor of `AlbumForm`, we set the name when we call the parent's constructor and then set the method and then create four form elements for the id, artist, title, and submit button. For each item we set various attributes and options, including the label to be displayed.

We also need to set up validation for this form. In Zend Framework 2 is this done using an input filter which can either be standalone or within any class that implements `InputFilterAwareInterface`, such as a model entity. We are going to add the input filter to our `Album` entity:

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

```
<?php

namespace Album\Model;

use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Factory as InputFactory;
use Zend\InputFilter\InputFilterAwareInterface;
use Zend\InputFilter\InputFilterInterface;

class Album implements InputFilterAwareInterface
{
    public $id;
    public $artist;
    public $title;

    protected $inputFilter;

    public function exchangeArray($data)
    {
        $this->id     = (isset($data['id'])) ? $data['id'] : null;
        $this->artist = (isset($data['artist'])) ? $data['artist'] : null;
        $this->title  = (isset($data['title'])) ? $data['title'] : null;
    }

    public function setInputFilter(InputFilterInterface $inputFilter)
    {
        throw new \Exception("Not used");
    }

    public function getInputFilter()
    {
        if (!$this->inputFilter) {
            $inputFilter = new InputFilter();

            $factory = new InputFactory();

            $inputFilter->add($factory->createInput(array(
                'name'     => 'id',
                'required' => true,
                'filters'  => array(
                    array('name' => 'Int'),
                ),
            )));

            $inputFilter->add($factory->createInput(array(
                'name'     => 'artist',
                'required' => true,
                'filters'  => array(
                    array('name' => 'StripTags'),
                    array('name' => 'StringTrim'),
                ),
                'validators' => array(
                    array(
                        'name'    => 'StringLength',
                        'options' => array(
                            'encoding' => 'UTF-8',
                            'min'      => 1,
                            'max'      => 100,
                        ),
                    ),
                ),
            )));

            $inputFilter->add($factory->createInput(array(
                'name'     => 'title',
                'required' => true,
                'filters'  => array(
                    array('name' => 'StripTags'),
                    array('name' => 'StringTrim'),
                ),
                'validators' => array(
                    array(
                        'name'    => 'StringLength',
                        'options' => array(
                            'encoding' => 'UTF-8',
                            'min'      => 1,
                            'max'      => 100,
                        ),
                    ),
                ),
            )));

            $this->inputFilter = $inputFilter;
        }

        return $this->inputFilter;
    }
}
```

The `InputFilterAwareInterface` defines two methods: `setInputFilter()` and `getInputFilter()`. We only need to implement `getInputFilter()` so we simply throw an exception in `setInputFilter()`.

Within `getInputFilter()`, we instantiate an `InputFilter` and then add the inputs that we require. We add one input for each property that we wish to filter or validate. For the `id` field we add an `Int` filter to as we only need integers. 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 `StringLength` validator to ensure that the user doesn't enter more characters than we can store into the database.

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/Controller/AlbumController.php:**

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

        $request = $this->getRequest();
        if ($request->isPost()) {
            $album = new Album();
            $form->setInputFilter($album->getInputFilter());
            $form->setData($request->getPost());
            if ($form->isValid()) {
                $album->exchangeArray($form->getData());
                $this->getAlbumTable()->saveAlbum($album);

                // Redirect to list of albums
                return $this->redirect()->toRoute('album');
            }
        }

        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->setAttribute('value', '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 and will use a different label.

```
        $request = $this->getRequest();
        if ($request->isPost()) {
            $album = new Album();
            $form->setInputFilter($album->getInputFilter());
            $form->setData($request->getPost());
            if ($form->isValid()) {
```

If the Request object's `isPost()` method is `true`, then the form has been submitted and so we set the form's input filter from an album instance. We then set the posted data to the form and check to see if it is valid using the `isValid()` member function of the form.

```
                $album->exchangeArray($form->getData());
                $this->getAlbumTable()->saveAlbum($album);
```

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

```
                // Redirect to list of albums
                return $this->redirect()->toRoute('album');
```

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:

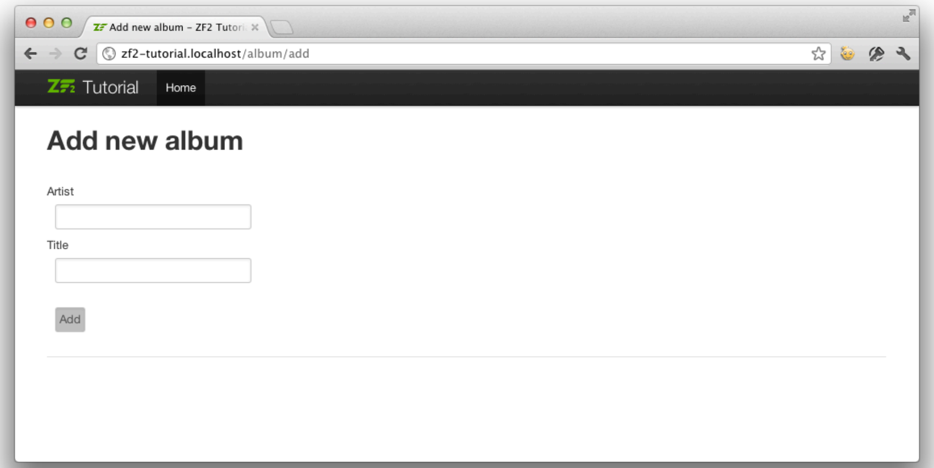
**module/Album/view/album/album/add.phtml:**

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

<?php
$form = $this->form;
$form->setAttribute('action', $this->url('album', array('action' => 'add')));
$form->prepare();

echo $this->form()->openTag($form);
echo $this->formHidden($form->get('id'));
echo $this->formRow($form->get('title'));
echo $this->formRow($form->get('artist'));
echo $this->formInput($form->get('submit'));
echo $this->form()->closeTag($form);
```

Again, we display a title as before and then we render the form. Zend Framework provides some view helpers to make this a little easier. The `form()` view helper has an `openTag()` and `closeTag()` method which we use to open and close the form. Then for each element with a label, we can use `formRow()`, but for the two elements that are standalone, we use `formHidden()` and `formInput()` You can use `formInput()`, `formLabel()` and `formElementErrors()` for all fields and create specific 'decoration' HTML if you wanted to though.



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()
    {
        $id = (int)$this->params('id');
        if (!$id) {
            return $this->redirect()->toRoute('album', array('action'=>'add'));
        }
        $album = $this->getAlbumTable()->getAlbum($id);

        $form = new AlbumForm();
        $form->bind($album);
        $form->get('submit')->setAttribute('value', 'Edit');

        $request = $this->getRequest();
        if ($request->isPost()) {
            $form->setData($request->getPost());
            if ($form->isValid()) {
                $this->getAlbumTable()->saveAlbum($album);

                // Redirect to list of albums
                return $this->redirect()->toRoute('album');
            }
        }

        return array(
            'id' => $id,
            'form' => $form,
        );
    }
```

```
//...
```

This code should look comfortably familiar. Let's look at the differences from adding an album. Firstly, we look for the id that is in the matched route and use it to load the album to be edited:

```
        $id = (int)$this->params('id');
        if (!$id) {
            return $this->redirect()->toRoute('album', array('action'=>'add'));
        }
        $album = $this->getAlbumTable()->getAlbum($id);
```

`params` is a controller plugin that provides a convenient way to retrieve parameters from the matched route. We use it to retrieve the id from the route we created in the modules' `module.config.php`. If the id is zero, then we redirect to the `add` action, other wise, we continue by getting the album entity from the database.

```
        $form = new AlbumForm();
        $form->bind($album);
        $form->get('submit')->setAttribute('value', 'Edit');
```

The form's `bind()` method attaches the model to the form. This is used in two ways:

1. When displaying the form, the initial values for each element are extracted from the model.
2. After successful validation in `isValid()`, the data from the form is put back into the model.

These operations are done using a hydrator object. There are a number of hydrators, but the default one is `Zend\Stdlib\Hydrator\ArraySerializable` which expects to find two methods in the model: `getArrayCopy()` and `exchangeArray()`. We have already written `exchangeArray()` in our Album entity, so just need to write `getArrayCopy()`:

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

```
...

    public function exchangeArray($data)
    {
        $this->id     = (isset($data['id'])) ? $data['id'] : null;
        $this->artist = (isset($data['artist'])) ? $data['artist'] : null;
        $this->title  = (isset($data['title'])) ? $data['title'] : null;
    }

    public function getArrayCopy()
    {
        return get_object_vars($this);
    }

...
```

As a result of using `bind()` with its hydrator, we do not need to populate the form's data back into the `$album` as that's already been done, so we can just call the mappers' `saveAlbum()` to store the changes back to the database.

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

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

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

<?php
$form = $this->form;
$form->setAttribute('action',
    $this->url('album', array('action' => 'edit', 'id'=>$this->id)));
$form->prepare();

echo $this->form()->openTag($form);
echo $this->formHidden($form->get('id'));
echo $this->formRow($form->get('title'));
echo $this->formRow($form->get('artist'));
echo $this->formInput($form->get('submit'));
echo $this->form()->closeTag($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()
    {
        $id = (int)$this->params('id');
        if (!$id) {
            return $this->redirect()->toRoute('album');
        }

        $request = $this->getRequest();
        if ($request->isPost()) {
            $del = $request->getPost()->get('del', 'No');
            if ($del == 'Yes') {
                $id = (int)$request->getPost()->get('id');
                $this->getAlbumTable()->deleteAlbum($id);
            }
            // Redirect to list of albums
            return $this->redirect()->toRoute('album');
        }

        return array(
            'id' => $id,
            'album' => $this->getAlbumTable()->getAlbum($id)
        );
    }
//...
```

As before, we get the id from the matched route,and check the request object's `isPost()` to determine whether to show the confirmation page or to delete the album. We use the table object to delete the row using the `deleteAlbum()` method and then redirect back the list of albums. If the request is not a POST, then we retrieve the correct database record and assign to the view, along with the id.

The view script is a simple form:

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

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

<p>Are you sure that you want to delete
  '<?php echo $this->escapeHtml($album->title); ?>' by
  '<?php echo $this->escapeHtml($album->artist); ?>'?
</p>
<?php
$url = $this->url('album', array('action' => 'delete', 'id'=>$this->id)); ?>
<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\Index',
                    'action'     => 'index',
                ),
            ),
        ),
```

change the `controller` from `Application\Controller\Index` to `Album\Controller\Album`:

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