

Modular applications with Zend Framework

<http://blog.keppens.biz/2009/06/create-modular-application-with-zend.html>

I admit. I am a sucker for modules. I like my code nicely separated in manageable blocks, that I can reuse whenever I want. Needless to say, I'm a big fan of the modules in Zend Framework. It isn't always very easy to set it up though. I had to look long and hard for answers on several questions and I noticed a lot of people had the same ones. So, I decided to make an article on creating a modular application in Zend Framework.

Pre-requisites

I'm assuming you have already downloaded Zend Framework and have configured your system, so we can use Zend_Tool and the zf scripts to set up our application. If you haven't, you can find a link to my blog at the bottom of this article. Zend_Tool isn't really necessary when setting up an application, but it makes it very easy and assures that your application follows the default directory structure.

Setting up the application

Create a directory that will hold your application, on Debian this is under the /var/www directory by default:

```
dev:~# mkdir /var/www/amazium
dev:~# cd /var/www/amazium
dev:~# zf create project .
Creating project at /var/www/amazium
```

This creates the project and the initial structure (see figure 1).

Let's see what files are important to us.

First of all you have the public directory. This will not only hold the index.php file that will start your application, but also public files such as images, css files, javascript files, etc...

Then you have the library directory. If you have your own library, this is where you can put it. I will briefly touch how you can add it to your application.

The test directory will hold the unit tests. This can be used to test your models which should contain the business logic, but also the actual pages themselves.

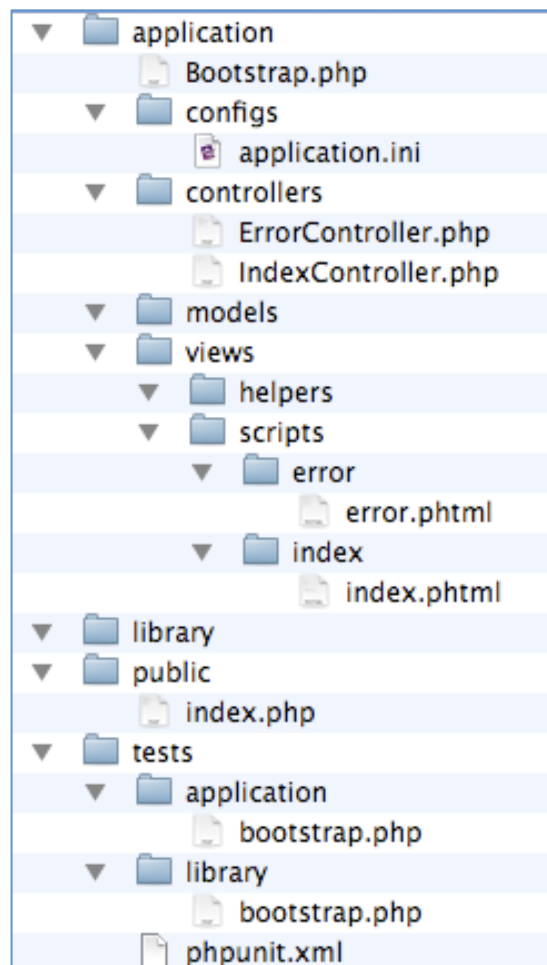


Figure 1: default project structure

Last but not least there is the application directory. This can be considered as the most important one. I must confess that in the past, I tried to put as many things like helpers, filters, models in my own library. But I've seen the light! It's actually far simpler to use the directory structure as defined by Zend Framework.

Digging into the application directory

configs

Inside the configs directory, you will find a file called application.ini. This file will be used to define your basic application configuration. We will see later how you can add your own resources to use this and to be auto loaded by your application. We will get back to this file on a number of occasions. Familiarize yourself with it, it will become one of your allies.

controllers

Inside the controller directory, you will find files for every controller. This is Zend Framework 101, so I'm not going into detail. When you get to modules, don't forget to prepend your module name to the class name (i.e. Admin_IndexController).

models

Inside the models directory, you can put your models. There is a lot of discussion to what a model is and here's my take: a model contains your business logic, but is not tied to a storage engine. This basically means that for me a model does not extend the Zend_Db_Table class. Under models you can put a directory DbTable and then put your data classes in there. It adds a level of abstraction and probably for a lot of people also an extra level of difficulty, but it's just good practice basically.

views

The views directory holds the view scripts, but also view related things such as filters and helpers. In the default directory layout the filters and layouts are not added, but we will do that manually later.

Adding your private library

My library is called Amz (short for Amazium) and contains some library classes I wrote in the past. Let's say I want to add this, I just add following entry in the config.ini:

```
autoloaderNamespaces.amz = "Amz_"  
autoloaderNamespaces.other = "AnotherLibrary_"
```

2 things to notice: since our application expects autoloaderNamespaces to be an array, you need add a key. This means you can add multiple custom libraries by writing multiple lines as above. The second thing to notice is to add an underscore at the end of your library prefix. If I would have only specified AnotherLibrary, this would actually have opened the door for classes such as AnotherLibraryThatIDontWant_Aclass).

Adding your first module

Ok, so that's it? Not really... you might have noticed we don't have a modules directory yet. Our application is still module unaware. So let's change that and add our first module, called "admin".

```
dev:~# zf create module admin
Creating the following module and artifacts:
/var/www/amazium/application/modules/admin/controllers
/var/www/amazium/application/modules/admin/models
/var/www/amazium/application/modules/admin/views
/var/www/amazium/application/modules/admin/views/scripts
/var/www/amazium/application/modules/admin/views/helpers
/var/www/amazium/application/modules/admin/views/filters
Updating project profile '/var/www/amazium/.zfproject.xml'
```

As you can see, it creates a new directory called "modules" in the application path. There it added the module directory "admin" with a similar structure for the module as we described above.

Let's create a new controller for the module so we have something to show.

```
dev:~# zf create controller index index-action-included=1 admin
Creating a controller at
/var/www/amazium/application/modules/admin/controllers/IndexController.php
Creating an index action method in controller index
Creating a view script for the index action method at
/var/www/amazium/application/modules/admin/views/scripts/index/index.phtml
Creating a controller test file at /var/www/amazium/tests/application/
modules/admin/controllers/IndexControllerTest.php
Updating project profile '/var/www/amazium/.zfproject.xml'
```

Now, if you try it in the browser, you get an error "Message: Invalid controller specified (admin)".¹

So, what did we do wrong? Nothing actually. You would expect your application is now module ready but it isn't. You created a first module, true... but you needed some extra configuration. So, let's open the application.ini we discussed earlier and add following lines to the end of the production block:

```
resources.frontController.moduleDirectory = APPLICATION_PATH "/modules"
resources.modules[]
```

Reload the page in the browser and you should get something...

Now, since we want to harness the full power of modules, we'll add a bootstrap first. The bootstrap will enable the use of class resources, but also make all models, filters, helpers, etc... available to your whole application.

¹ In all honesty I ran into a few problems the first time I tried because of bugs in Zend_Tool. I saw some patches in the bugtracker, so I assume they will be included in one of the upcoming minor releases (if they haven't been already): verify that 1/ the correct files were generated (index/index.phtml under admin/views/scripts), 2/ that the indexAction was added to the new IndexController (and not the default one) and 3/ that your controller's class name was prefixed by your module name.

Create the file Bootstrap.php under your admin directory and add this code:

```
<?php

class Admin_Bootstrap extends Zend_Application_Module_Bootstrap
{
}

```

Also modify the Bootstrap.php in your application path by adding this function:

```
protected function _initAppAutoload()
{
    $autoloader = new Zend_Application_Module_Autoloader(array(
        'namespace' => 'App',
        'basePath' => dirname(__FILE__),
    ));
    return $autoloader;
}

```

This will autoload to models, filters, helpers, etc... for our default module.

With both setup, we'll be able to access all these from our controllers later on.

Adding some layout

I bet you weren't really amazed by how it all looked? Well, that's because we didn't have a layout specified yet. Okay, the default index had some, but the layout should not be defined there.

Let us create a few directories first:

```
dev:~# cd /var/www/amazium
dev:~# mkdir public/images
dev:~# mkdir public/css
dev:~# mkdir public/js
dev:~# mkdir application/views/layouts

```

Now, create a file default.phtml and admin.phtml in the layouts directory and add your layout code to it. This is a very simple template, which just shows the content but is good to explain how to add the layouts. Replace the h1 title by the module name so you see a different template is loaded.

```
<?php echo $this->doctype() ?>
<html>

<head>
    <?php echo $this->headTitle() ?>
    <?php echo $this->headMeta() ?>
    <?php echo $this->headLink() ?>
    <?php echo $this->headStyle() ?>
    <?php echo $this->headScript() ?>
    <base href="/" />
</head>

<body id="page-home">
    <h1>Default</h1>
    <?php echo $this->layout()->content ?>
</body>

</html>

```

To get the layout working, you add the layout path and default layout to the application.ini file. Notice how we added a module resource entry for the layout as well. This will do nothing actually.... yet.

```
resources.layout.layoutPath = APPLICATION_PATH "/views/layouts"
resources.layout.layout     = default
admin.resources.layout.layout = admin
```

When you switch between your default en admin module, you'll see that it uses the same layout. One might think that prepending the second line with "default." will enable layouts specific to the module, but unfortunately it doesn't. It will use the last specified template, in our case the admin, as default. To mitigate this, we will use a controller plugin since we need access to the request object to decide which template to show.

I created Amz/Controller/Action/Helper/LayoutLoader.php and added following content:

```
<?php

class Amz_Controller_Action_Helper_LayoutLoader
extends Zend_Controller_Action_Helper_Abstract
{

    public function preDispatch()
    {
        $bootstrap = $this->getActionController()
                    ->getInvokeArg('bootstrap');
        $config = $bootstrap->getOptions();
        $module = $this->getRequest()->getModuleName();
        if (isset($config[$module]['resources']['layout']['layout'])) {
            $layoutScript =
                $config[$module]['resources']['layout']['layout'];
            $this->getActionController()
                ->getHelper('layout')
                ->setLayout($layoutScript);
        }
    }
}
}
```

Then I added a loader for it in the bootstrap:

```
protected function _initLayoutHelper()
{
    $this->bootstrap('frontController');
    $layout = Zend_Controller_Action_HelperBroker::addHelper(
        new Amz_Controller_Action_Helper_Layout());
}
```

If you reload your admin / default page, you should now see the module specific layout.

Spicing up the application with a model

For me a model is about business logic. It shouldn't about where you get data, but about what you do with it. Unfortunately there currently isn't a "Zend_Model" class you can abstract and use. So everyone basically invents his own system, or just plainly extend the DbTable. I've seen a very interesting implementation by Matthew Weier O'Phinney on the Dutch PHP Conference. The models had a data mapper which could be referenced for the data, while the

model itself did what it was there for. Unfortunately it is out of the scope of this article to go very deep on it, so I'll make a very simple model that doesn't do a lot at all: A color model. Basically it generates a hex color that can be used in html.

Create the file Color.php in your application/models directory:

```
<?php
class App_Model_Color
{
    public function getRandomHtmlColor()
    {
        $red    = rand(0,255);
        $green  = rand(0,255);
        $blue   = rand(0,255);
        return $this->getHtmlColor($red, $green, $blue);
    }

    public function getHtmlColor($red, $green, $blue)
    {
        $color = '#' . $this->_getHex($red)
                . $this->_getHex($green)
                . $this->_getHex($blue);
        return $color;
    }

    protected function _getHex($number, $digits = 2)
    {
        return substr(str_repeat('0', $digits) .
                    dechex($number), - $digits);
    }
}
```

As you can see it allows you to get a random color or to convert 3 integers to an html color. As I said.... nothing spectacular.

Modify the indexAction of your IndexController :

```
public function indexAction
{
    $model = new App_Model_Color();
    $this->view->color = $model->getRandomHtmlColor();
}
```

Lastly modify your view script (views/scripts/index/index.phtml) :

```
My random number is : <?php echo $this->color; ?>
```

When you refresh your default page, it should now show another html color each time you refresh. Now, that was easy wasn't it? But how come it recognized the model without any trouble? Well, the `_initAutoload` function that we specified for the bootstrap takes care of it. Since we specified App as the "prefix", our default modules will be available as `App_Model_*`. If you had created a `DbTable` directory, they would be available as `App_Model_DbTable_*`.

In the same way you can put models in the modules. You don't have to specify an autoloader there. The module bootstraps take care of everything for you. Instead of the App, the module name will be the prefix. In our example models would be available as `Admin_Model_*` and `Admin_Model_DbTable_*`.

Custom View Helpers

I tried the same with view helpers and you can get them autoloaded the same way. But there is a catch... apparently the view looks for `Zend_View_Helper_*` instead of the expected `App_View_Helper_*`. Since I wanted to use the `App` variant, I solved this by adding following line in the `config.ini` :

```
resources.view.helperPath.App_View_Helper = APPLICATION_PATH "/views/helpers"
```

Now, let's create a view helper that displays a text in a certain color.

Create a file `views/helpers/ColoredText.php` and add following code:

```
<?php

class App_View_Helper_ColoredText extends Zend_View_Helper_Abstract
{

    public function coloredText($text, $color)
    {
        $text = $this->view->escape($text);
        $ctext = '<span style="color: ' . $color . '">' .
                $text . '</span>';
        return $ctext;
    }
}
```

Change the view script by following code:

```
My random number is :
<?php echo $this->coloredText($this->color, $this->color); ?>
```

When you refresh the page now, you should see the color name displayed in the actual color it represents.

Conclusion

So, we have created a new modular application. We have tied in a library, we have tied in module specific layouts, we have worked with models and view helpers. There are a zillion other things you can do, but most of them are covered in the many tutorials online that don't specifically cover modules.

I hope this was useful for you, feel free to send in your comments.

See you in the PHP universe!

Jeroen Keppens

Links

My Blog

<http://blog.keppens.biz>

Article about setting up Zend Tool:

<http://blog.keppens.biz/2009/05/setting-up-new-zend-framework.html>