

Self-Contained Reusable Zend Framework Modules With Standardised Configurators

It was during last week, while writing out a draft chapter for [Zend Framework: Survive The Deep End](#), that I found myself hitting a conceptual wall. If you are familiar with Zend Framework, you likely understand the concept of a Module in some detail. A Module is, in theory, a reusable collection of controllers, views and other classes which is packaged in its own directory for simpler copying or separate treatment in a version control system like git or subversion.

The problem I had lay in demonstrated this fabled reusability. The more I tried to, the more I found myself throwing out cautions, warnings and advice on what to avoid doing. When it came to using Zend_Application, the trend continued since Zend_Application (a great component otherwise!) is just badly documented and explained. So off went another section just to try and explain its often confusing terminology. If you read the source code it all makes sense but if you don't the disconnect between the explanations and a user's expectations is obvious.

Reusability Rules

Zend Framework developers have, for better or worse, been ignoring the potential of modules for an interminably long time. It's not that big of a surprise given the focus of the framework has always been to present a use-at-will architecture which relies on loose coupling and independent components. Tight integration through overarching features (which don't break the framework's impressive orthogonality) like a command line tool or initialisation tools has long been neglected until very recently. Zend Framework 1.8 saw the long needed introduction of Zend_Application which offers standardised bootstrapping. Zend_Tool is another ongoing effort on the command line side.

The most typical example of a module in the literature is also the worst. An administration backend. It's a logical module since it's a completely separate system to the frontend, but it's the worst example because it is so very rarely reusable. Not every logical separation is reusable - they are mutually exclusive concepts. You could equally have a logical module which itself is comprised of several reusable modules and one non-reusable module. By definition, an administration backend is closely tied to CRUD operations against the application's domain model (at least to start with). Since each application will be different, the administration backend will also.

A far better example of a reusable module is something much narrower and focused. Consider a module dealing with User Management, or Paypal IPN integration, or implementing a blog aggregator. These are each common needs which, depending on the application, may require little change from implementation to implementation. Drop them in, configure them, integrate them, and you can have them working with few issues. Unfortunately, we keep focusing our module efforts on obviously non-reusable things like administration backends. Losing sight of the potential reuse of smaller subsystems will lead us to repeatedly developing them over and over again without even noticing this as a problem.

For the Zend Framework, this would be a big win. Rather than having developers re-implement commonly used web application systems it would encourage the distribution of third-party modules which would benefit from open source licensing and feedback. Imagine your next application requiring a minimal blog or integration with Paypal IPN and finding a third party module which does the trick so you can save some development time.

Achieving Reusability

When we discuss achieving reusability there are several factors and features covered when it comes to modules:

1. They are separated into their own parent directory.
2. They can apply specific configuration when accessed.
3. They require no special integration work.
4. Their classes are automatically available to the host application.
5. They are not required to contain controllers or views.

It's not an exhaustive list. Items 1, 4 and 5 are already a reality. Zend Framework modules do live in a module directory, using `Zend_Application` and some conventions their classes are autoloaded on demand and they are not required to contain controllers and views. A module may exist which merely offers models, helpers and some default forms.

So our path to reusable modules is hampered by items 2 and 3. Modules currently don't have on-access configuration unless we impose it through various means. This flows into integration work which is commonly needed to achieve this in the first place.

The Layout Example: Integration Through Front Controller Plugins

A simple example, taking our example administration backend (an "admin" module) is that of switching layouts. Suppose our main application uses a professional design but our administration backend uses a very simple minimal one. How do we switch layouts when the admin module is accessed so the correct layout template is applied?

An initial expectation might be to try this from our `application.ini` file (if using `Zend_Application`) using:

```
; Default Module
resources.layout.layout = "default"
resources.layout.layoutPath = APPLICATION_PATH "/views/layouts"
; Admin Module
admin.resources.layout.layout = "default"
admin.resources.layout.layoutPath = APPLICATION_PATH "/modules/admin/views/layouts"
```

Ah, module configuration! This is a very common first attempt since the expectation is that a module framed configuration will kick in only for that module.

Alas, this will not work even if it looks blatantly obvious that it should (expectations again). Module configuration here is used during the bootstrapping process which occurs before a request is routed, i.e. we can't know what module the request relates to yet because our routes are not yet applied. So any module configuration of this type is actually applied to the same resources as the previous set of settings, i.e. module configuration overwrites the main resource configuration. The example above, replaces the layout and layout path across the entire application with that of the admin module. Visiting any module, including the default module, will show the last configured layout being applied no matter what module prefixing you use.

What possible good is having this confusing module configuration then? Well, it's useful to pass custom options to your module's bootstrap class for something. Beyond that, I can't think of many other use cases. You could, for example, use it to register module-hosted plugins, classes, etc but that's just as easily done without the module name prefixed to the option. Note, this is my own ignorance speaking - I haven't seen any detailed examples using this.

In the meantime, how do we ensure the layout is only switched if the module is accessed? The above configuration won't work, obviously. Well, we first need to know what module is being dispatched to, so it must be done after routing has taken place. The most obvious location for our switching logic is therefore a front controller plugin which implements the `preDispatch()` method (i.e. it's executed just before any controller is called, giving us an opportunity to re-configure some resources like `Zend_Layout`).

Here's an example plugin for this. It's the simplest possible version - I've seen some examples which forget that `Zend_Layout` already offers a plugin we can subclass to keep things simple.

```
<?php
class ZFExt_Controller_Plugin_LayoutSwitcher
extends Zend_Layout_Controller_Plugin_Layout
{
    public function preDispatch(Zend_Controller_Request_Abstract $request)
    {
        $this->getLayout()->setLayoutPath(
            Zend_Controller_Front::getInstance()->getModuleDirectory(
                $request->getModuleName()
            ). '/views/layouts'
        );
        $this->getLayout()->setLayout('default');
    }
}
```

It's not a perfect class - every single module must follow the convention on using the same layout path and layout name. We could also add some logic to skip the default module since this would configure it twice for no reason. But it works! When we access the "admin" module, the layout path will be set to `/application/modules/admin/views/layouts` and the layout template used will be `default.phtml`. The default modules path will likewise reflect its original configuration.

To get this working, let's add a new layout resource option so our custom plugin replaces the default one from `Zend_Layout`:

```
; Default Module
resources.layout.layout = "default"
resources.layout.layoutPath = APPLICATION_PATH "/views/layouts"
resources.layout.pluginClass= "ZFExt_Controller_Plugin_LayoutSwitcher"
```

This does work by the way 

. Add the following test alongside a directory `_modules` containing a readable subdirectory `_modules/admin` and it will pass.


```
<?php
class ZFExt_Controller_Plugin_LayoutSwitcherTest extends PHPUnit_Framework_TestCase
```

```

{
    protected $plugin = null;
    protected $request = null;
    public function setup()
    {
        Zend_Controller_Front::getInstance()->addModuleDirectory(dirname(<u>_FILE_</u>) .
'_modules');
        $this->plugin = new ZFExt_Controller_Plugin_LayoutSwitcher(
            new Zend_Layout
        );
        $this->request = new Zend_Controller_Request_Http;
    }
    public function teardown()
    {
        Zend_Controller_Front::getInstance()->resetInstance();
    }
    public function testSwitchesLayoutNameIfAdminModuleDispatched()
    {
        $this->request->setModuleName('admin');
        $this->plugin->preDispatch($request);
        $this->assertEquals('default', $this->plugin->getLayout()->getLayout());
    }
    public function testSwitchesLayoutPathIfAdminModuleDispatched()
    {
        $this->request->setModuleName('admin');
        $this->plugin->preDispatch($request);
        $this->assertEquals(dirname(<u>_FILE_</u>) . '_modules/admin/views/layouts',
            $this->plugin->getLayout()->getLayoutPath());
    }
}
}

```

How about something different? What if our main application uses HTML 5 and our administration backend uses XHTML 1.0 Transitional. Damn, we need another plugin. Worse, this time we can't reduce it to a convention since a doctype can be anything and we have no way of predicting it. We could set it on the module layout, but layouts are rendered last - it would still not be applied to page level templates or partials. Forms would be messed up, for example. Same goes for the character encoding of our views (messed up escaping).

So slap in another plugin to handle doctype switching, and another to handle encoding changes. Why not add another just for fun so we can handle connecting to a module's shared database. Then there's the case where... Alright...enough of that 

. The point is a simple one. We are adding custom plugins all over the place to integrate modules into our application. These plugins will not be reusable, will require editing for different modules, and will need to be rewritten between applications. We need something more structured.

Integration: Modular Pre-Dispatch Configuration

As we can see, integration efforts are tricky. Relying on custom plugins and trying to wrestle the bootstrap

system into submission are a lot of trouble to go through. Zend_Application and bootstrapping may not offer us a good solution for integration, but they do give us the roadmap.

Zend_Application defines bootstrap classes which are used to initialise resources before routing takes place. Keeping it simple, we need to reconfigure resources after routing but before dispatching occurs. We may also need to initialise different resources if they are used by a module, but not the main application. What we need is something like bootstrapping that occurs after routing. After a bit of thought, we might come to the conclusion that the current Resource classes of Zend_Application could live parallel to counterparts who exist not to initialise a Resource, but to modify a pre-initialised Resource by resetting its configuration. These are what I term Configurators, maybe not the best name, which mirror Resources.

Take a Configurator class for Layouts as an example:

```
<?php
class ZFExt_Application_Module_Configurator_Layout
    extends Zend_Application_Resource_ResourceAbstract
{
    public function init()
    {
        $layout = $this->getBootstrap()->getResource('Layout');
        $layout->setOptions($this->getOptions());
    }
}
```

Our Configurator actually extends from Zend_Application_Resource_ResourceAbstract demonstrating its close relationship to a Resource. However, it does not create and initialise a Resource - it merely modifies the existing one by injecting a new configuration sourced from a module.

Where does this replacement configuration come from? I've decided to use a simple convention. If you want a module to impose its own configuration when, and only when, it is accessed then create that configuration in a file called module.ini located at /application/modules/admin/configs/module.ini. The configuration file could be any supported format, but I've used the INI format for simplicity. This would look like (using the typical environmental groups):

```
[production]
; Standard Resource Options
resources.layout.layout = "default"
resources.layout.layoutPath = APPLICATION_PATH "/modules/admin/views/layouts"
[staging : production]
[testing : production]
[development : production]
```

So, we have the Configurator class, and the configuration file it will use. Let's bind these together. We'll start by putting in place a class whose role is to use a collection of options loaded from module.ini to instantiate and run a set of Configurator classes.

```
<?php
class ZFExt_Application_Module_Configurator
{
    public function __construct(Zend_Application_Bootstrap_Bootstrapper $bootstrap,
        Zend_Config $config)
```

```

{
    $this->_bootstrap = $bootstrap;
    $this->_config = $config;
}
public function run()
{
    $resources = array_keys($this->_config->resources->toArray());
    foreach ($resources as $resourceName) {
        $options = $this->_config->resources->$resourceName;
        $configuratorClass = 'ZFExt_Application_Module_Configurator_' . ucfirst($resourceName);
        $configurator = new $configuratorClass($options);
        $configurator->setBootstrap($this->_bootstrap);
        $configurator->init();
    }
}
}
}
}

```

As you can see, it is very simple. It takes a configuration, detects what Resources it applies to, instantiates relevant Configurators and executes them. It could be improved a lot by allowing for custom Resources and other such customisations but for now the basics will do nicely.

Earlier, we mentioned that the application only becomes aware of the current module when the request is routed. Therefore, to get this working we need to trigger the Configurators after routing (or prior to request dispatching). We also need to check if the current module has a module.ini file and also ensure we skip over the default module (our main application space might be reusable so this is an arguable point and probably should be allowed for).

We'll accomplish this using a front controller plugin:

```


<?php
class ZFExt_Controller_Plugin_ModuleConfigurator
extends Zend_Controller_Plugin_Abstract
{
    public function preDispatch(Zend_Controller_Request_Abstract $request)
    {
        $front = Zend_Controller_Front::getInstance();
        $bootstrap = $front->getParam('bootstrap');
        $moduleName = $request->getModuleName();
        if ($moduleName == $front->getDefaultModule()) {
            return;
        }
        $moduleDirectory = Zend_Controller_Front::getInstance()
            ->getModuleDirectory($moduleName);
        $configPath = $moduleDirectory . '/configs/module.ini';
        if (file_exists($configPath)) {
            if (!is_readable($configPath)) {
                throw Exception('modules.ini not readable for module "' . $module . '"');
            }
        }
        $config = new Zend_Config_Ini($configPath, $bootstrap->getEnvironment());
        $configurator = new ZFExt_Application_Module_Configurator(
            $bootstrap, $config
        );
    }
}

```

```
$configurator->run();  
}  
}  
}
```


If you're still with me, and can piece this story together, you achieve a workflow as follows for the admin module when accessed from any URI like <http://example.com/admin>. I've skipped steps where not relevant.

1. Normal bootstrapping is completed with the layout being initially set using the application.ini options.
2. The request is routed. The module name "admin" is set internally.
4. `ZFExt_Controller_Plugin_ModuleConfigurator::preDispatch()` is called before dispatching commences (getting it done before other plugins can be addressed in the future).
5. The plugin detects `/application/modules/admin/configs/module.ini` and loads it as a `Zend_Config` instance.
6. The plugin instantiates `ZFExt_Application_Module_Configurator`, passes it the configuration and original bootstrap, and calls the new object's `run()` method.
7. The Module Configurator assesses the configuration for resource names. For each resource detected, it instantiates a Resource Configurator like `ZFExt_Application_Module_Configurator_Layout`.
8. The Resource Configurator is executed and applies the new configuration to the existing Layout Resource thus overwriting the original configuration.
9. Dispatching occurs - the admin module's requested action is rendered with the correct admin layout.

By itself, this seems like a lot of trouble to go through - except what if it becomes a Zend Framework feature? All of a sudden, countless custom plugins will meet their death and be replaced by a simple configuration file! 

Conclusion

The goal of this article was to highlight the problems of achieving reusable modules and implement, as a proof of concept, at least part of the solution with an eye to encouraging greater discussion of where to go from here. I, for one, would love to see this included in the Zend Framework so we can get over the trend of relying on custom plugins and evolve towards a more standardised means of configuring modules.

If you've enjoyed the article please do add a comment and make suggestions on what could be improved or added as a feature. If I get enough positive feedback I'll move this into a formal proposal (preferably with a partner or two 

). If you're interested in collaborating on a proposal addressing this let me know!

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 19:29

Pádraic excellent article. I can't wait to see the proposal on this. If i get some time i'd love to help on this. Just shoot me an email and let me know what's needed. Anonymous on Sep 13 2009, 19:45

Great idea.

Storing the configurator ini file within the module subdirectory is the wrong place though as these are likely to need configuring on a per-application basis.

I would have thought `{modulename}.ini` within `application/configs` would make sense.

Regards,

Rob.... Anonymous on Sep 13 2009, 20:22

That could work too - as I said, the more discussion on these the quicker an acceptable option will turn up



Feedback and debate is

the grease on this proposal's wheels. Anonymous on Sep 13 2009, 20:30

Padraic this is a really great start. Definitely a proposal is needed! +1 from me. Anonymous on Sep 13 2009, 21:35

What about triggering the router to find the module name before execute module bootstrap, so we dont have to execute every module bootstrap classes and we can solve your problem "2. They can apply specific configuration when accessed." Anonymous on Sep 13 2009, 22:10

The problem there is that that plugins depend on the order of execution. A plugin running just after the router completes may need something the bootstrap would have initialised but hasn't because we shifted routing to the fore. Also routing needs any custom routes, and these would be loaded during bootstrapping. Anonymous on Sep 13 2009, 22:25

Modules already have separate bootstraps, why can't those be used? Because they are executed before routing as well? Anonymous on Sep 14 2009, 03:00

Exactly - see Zend_Application_Resources_Module to see how module bootstraps are used. It's a simple loop where all of them are immediately executed for all requests. Anonymous on Sep 14 2009, 03:06

Awesome solution Padraic - would definitely love to see a proposal for this.

Just wondering about the possibility of "sub-modules" using this sort of implementation. Would they be possible? For example, if we wanted to create a "News" module as a sub-module of the Admin module, could we still use the same approach? Anonymous on Sep 14 2009, 07:31

Great idea Padraic, I've been working on this same problem and I find your solution very elegant.

I do see one problem though. What if a resource has not been initialized in application.ini but IS mentioned in module.ini? Say for instance your default module doesn't need a database and your admin module does (unlikely, I know). The configurator will try to configure a non-existing resource. Then you would have to place something like admin.resources.db.* in application.ini for your solution to work.

Having to do that defeats the idea of self-contained reusable modules. You should be able to just drop them in and everything should work without editing application.ini. To achieve this, I have created a Zend_Application resource called Modulesetup, that checks all modules for a module.ini and 'adds' their config to the main config during bootstrapping. This way, all resources mentioned in a module.ini are bootstrapped as if they were found in application.ini.

I think this could work together with your code very well. Check it out here:

<http://blog.vandenbos.org/2009/07/07/zend-framework-module-config/> Anonymous on Sep 14 2009, 10:22

I know about that problem but skipped over it. However, since Configurators live beside Resources they could also bootstrap uninitialised resources using Zend_Application. So it's not that difficult. This also avoids needing config merging, for now perhaps (something else I'll be looking at as the proposal moves forward).

I have your blog bookmarked from last week, by the way



. There are only a handful of us blogging about all this. Anonymous on Sep

14 2009, 12:28

I have a similar setup for my modules. But in my mind, the goal for a REAL modular app is a all-embracing modular setup. I mean default, blog and so on must be modular as possible. The main difference in default modules I used is the layout, the functions are often similar (navigation, main content, login), also in blog, guestbook or shop modules.

I will decouple and split the ini files and the needed bootstrap classes for each module, without repeating functions or entries twice in it.

My 2 cents... Anonymous on Sep 14 2009, 12:56

Good point.

Also, I am interested in helping you out with making a proposal out of this. Just let me know what I can do to help. Anonymous on Sep 14 2009, 13:22

Good points, but to define module based configuration files you have (at least to choices) put them into the application.ini in [modulename] sections or as Rob mentioned put them into separate directory where they can be automatically or manually picked up when needed (like additional ini files in php or apache2)

r. Sandor Anonymous on Sep 14 2009, 13:36

True - I used the module.ini example though because it's the most self-contained option, especially if a module comes with provided default settings. Something alongside application.ini could be optionally used to provide override or other custom options also. Anonymous on Sep 14 2009, 13:47

of course 'to choices' should have been written as 'two choices' (chouckle).

Other thing that should be taken into consideration is resource handling, as of php is more related to filehandling routines (rather than java and ruby which relates more on memory usage) separating settings into multiple ini's is waste of resources, every time a application or module settings needs to be defined all the ini files will be read by php. I would rather say to read them once and use them during the application from a more resource friendly place like Sessions (strictly through own Session wrapper classes or objects).

This is my 2 cent...

r. Sandor Anonymous on Sep 14 2009, 15:18

I couldn't agree more. I think self contained modules are the next big leap for the Zend Framework, just look at how successful distributable independent modules are in other (admittedly more "opinionated") frameworks. Drupal and Joomla jump to mind in the PHP world.

As you mentioned, much of the infrastructure is now in place and there seems to be people working on the problem. I just hope we can come to a usable standard in the near future. Anonymous on Sep 14 2009, 15:20

Zend framework needs self contained modules to stay relevant to today's technologies. Joomla is leaving Zend in their dust in recent years, Drupal too. Anonymous on Sep 14 2009, 22:09

How would you propose handling inter-module dependencies?

Would we need to provide some sort of system where you can check if a modules exists etc?

Also would we have problems if a module contained models, with no standard model defined in ZF you could have a mix of different model strategies, though I suppose this would be up to the developer to choose the right module for their application...

Anyway just some initial thoughts hope they help, nice post... Anonymous on Sep 15 2009, 14:41

The simplicity of this standardization could be no more better.

This seems like a very good proposal. When I was reading about modules in the ZF's Reference Guide, I was thinking how to make it easier.

The admin backend example module don't help much. When I think about modules, the first thing I remeber are the Drupal modules, however, the mechanics are very different as Drupal is a CMS with a framework very coupled to the database (not entirely, sure).

What I wanted to know more is how do you see modules that ships with Models that needs to be persisted? The use case is very different (again, Drupal as comparison) as the user here is strictly a ZF developer.


If a module with a Model needs to be persisted, something in Zend_Tool could help manage with the integration tasks.

Just a little bit more of ZFExt_Model_OpenSource::addCents(0.02); Anonymous on Sep 15 2009, 14:43

Hard to say at the moment. It would be a bit like how Resources currently work in bootstrap classes I suspect. If you need a specific module to be configured before the current one, then it would be defined possibly as a configuration setting. The Configurator class is very simple so it would easy to implement such an approach. Anonymous on Sep 15 2009, 16:14

I think it do make sense to put a config file in the module itself, with the default settings.

When you want to override a setting of the module (e.g. another layout), you should be able to put an extra config file (with only that particular setting) in the applications config directory, which overrides just that default setting.

In that case you can make completely stand-alone modules, which are highly configurable as needed, without braking something because you forget a setting 

Anonymous on Sep 16 2009, 08:16

Nice.

To the discussion: I've also seen other approach to the problem (more here <http://code.google.com/p/mz-project/>).

It is basicaly about registering one plugin (passing the application instance as param in constructor) which at each hook method checks the current moduleName (so it can get to right module resource) and then if the module Bootstrap does not contain "current hook" method...

The convenience here is, that you write you modules bootstraps as usual (with the _init* methods which get always executed), plus regular plugin-like methods (preDispatch, postDispatch, ...) which get only executed at the particular event.

Oh my english... Anonymous on Sep 16 2009, 11:08

Great synopsis, everything would be much simpler if the framework didn't rely so much on plugins to conquer certain elements of programming. Anonymous on Sep 22 2009, 21:16

Hey, good points, I have similar setup for my modules. But to define module based configuration files it is better to store put them into the application.ini. Anonymous on Sep 29 2009, 14:43

I know this is an old thread but I think this is an important topic for ZF to continue to develop so here goes...

I've been expanding on this idea in a new project and I'm encountering the need for the modules to register their own front controller plugins to provide authentication/authorization type stuff. In the past I've just done this in the main application bootstrap but then each time I added a module I would have to modify the logic of that plugin to "know" about the new module and apply it's specific needs (e.g. what page should it forward to for access denied or login, etc.) which seems like a whole lot of logic to be running on **every**

request, not to mention it violates your points about keeping the modules independent.

Also I didn't want to do this in the module specific bootstraps since, again, you'd end up with plugins running on **every** request even if the request had nothing to do with that module. So ...

Long story short I've modified your ModuleConfigurator plugin class to override Zend_Controller_Plugin_Abstract::routeShutdown() instead of preDispatch() so that the specific module resource configurator classes can register more plugins and have them run prior to dispatch. One drawback of this, however, is that such plugins would not be able to override routeStartup() or routeShutdown(), since those would have run before such plugins were even registered.

Just putting that out there so I can get feedback on this and also to hear how others are tackling this type of issue.

Great article btw. Anonymous on Feb 23 2010, 17:46