

Tuesday, June 5, 2007

Having a bad ViewRenderer day in your ZF app?

Over the last week a lot of the activity on the Zend Framework mailing lists has revolved around the introduction in 1.0.0 RC1 of the ViewRenderer action helper. As of RC1 this helper has been enabled by default. Lot's of queries have been raised about how to disable, modify it, and generally how current applications can be made to work with the ViewRenderer.

The ViewRenderer "action helper" is the class `Zend_Controller_Action_Helper_ViewRenderer`. It's primary purpose is to facilitate the automated rendering of View scripts (templates) based on the generally accepted Zend Framework conventions. It's these conventions which will cause a lot of people grief, since the previous reliance on programmers defining the template to render has likely led to inconsistent template names. The helper itself replaces much of the View integration methods contained in `Zend_Controller_Action` like `initView()`, `render()`. This is a good thing, since it decouples the automated View code from the Controller class. Decoupling is hugely important in allowing classes to be modified or replaced easily and efficiently.

The problems most people are having boil down to a few simple ones:

1. ViewRenderer is enabled by default
2. Inconsistent naming of template files
3. Inconsistent location of template files
4. Disagreements with the ZF conventions
5. Reusing template files across Controller actions

The first can be remedied by disabling the ViewRenderer completely. This is easily done by passing the following parameter to the `Zend_Controller_Front` class:

```
Zend_Controller_Front::getInstance()->setParam('noViewRenderer', true);
```

This should be sufficient in many cases, allowing you to pretend the ViewRenderer does not exist. The problem here is that ViewRenderer is not something evil - it's something good. Using it reduces the code you need to type for all actions. So a more attractive step is adding support for the ViewRenderer to your code. This is where problems 2-4 arise.

The first problem is inconsistent names for templates. I can't help you much here except advise you use a convention to name templates. Using a convention would allow ViewRenderer to find them easily. Using ad-hoc names without structure makes any level of automation immensely difficult.

The Zend Framework has gradually been documenting a set of conventions for how you should organise your application. The most famous is likely the [Conventional Modular Directory Structure](#) which addresses the filesystem. The ViewRenderer action helper introduces another for template naming. It assumes that templates are related on a 1:1 basis with Controller actions and uses a naming convention of the form:

```
{controllerName}/{actionName}.phtml
```

For example, the template for `IndexController::indexAction` would be located at:

```
index/index.phtml
```

The full path would therefore be: /src/default/views/scripts/index/index.phtml

From the booing in the audience I gather some of you may disagree 

. This is easily remedied by setting your own custom format for template names. In fact you can change a few things in the ViewRenderer by customising your own instance of it! Imagine you did not like splitting templates into separate directories, and kept everything in the form:

```
index_index.tpl.php
```

under /src/views/scripts (no "default" Module subdirectory). Then you can do some customisation using the following snippet:

```
require_once 'Zend/Controller/Action/Helper/ViewRenderer.php';
$viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer;
$viewRenderer->setViewSuffix('tpl.php');
$viewRenderer->setViewScriptPathSpec(':controller_:action_:suffix');
$viewRenderer->setViewBasePathSpec(APPLICATION_PATH . '/views');
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
```

The above code explicitly creates the ViewRenderer instance, and applies some new settings. You could put it in your bootstrap file. If you want to put it after the Front Controller has been instantiated you'll need to replace creating a new instance with:

```
$viewRenderer = Zend_Controller_Action_HelperBroker::getExistingHelper('viewRenderer');
```

This demonstrates how a decoupled helper makes customised View conventions much easier to implement than you might suspect. Sure it looks more complex, but it's simpler than replacing the whole class or performing heavy subclassing.

There is one small issue here. You can influence the BasePath set on a View, but not the individual sub-directories of the BasePath like /scripts, /filters, /helpers. Luckily, ViewRenderer does not override existing paths set on a View, it just adds to the list - so yes, you can customise to this level also by setting a custom View object for the ViewRenderer to chew on using your preferred path settings:

Let's say we didn't use a /scripts subdirectory for Views. And we wanted to use a custom Smarty driven View object also.

```
require_once 'Zend/Controller/Action/Helper/ViewRenderer.php';
require_once 'Zps/View/Smarty.php';
$smartyView = new Zps_View_Smarty;
$smartyView->setScriptPath(APPLICATION_PATH . '/views');
$viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer;
$viewRenderer->setView($smartyView);
$viewRenderer->setViewSuffix('.tpl');
$viewRenderer->setViewScriptPathSpec(':controller_:action_:suffix');
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
```


The above code now allows the ViewRenderer to search two locations for templates. The first is the custom

/views. The second is the ViewRenderer set default of /views/scripts. Setting a custom View object prevents the default instantiation of a stock Zend_View instance.

The final problem folk have met is how to selectively reuse templates across actions. In writing a LoginController class recently, I needed to use the same View (a login form template with optional error message) across two actions: indexAction and processAction. The indexAction method would map 1:1 to login_index.phtml. How would I get the processAction method to use the same template, and not the ViewRenderer automated selection of login_process.phtml?

There are actually two methods. Firstly you can render into the Response object manually using:

```
$this->_helper->viewRenderer->setNoRender();
$this->view->loginError = true;
$this->getResponse()->setBody(
    $this->view->render('login_index.phtml')
);
return;
```

Above I generally get the Response object from Zend_Controller_Front in my bootstrap and manually echo it. There's no coupling if there's no automation 

. So before 1.0.0 RC1 I avoided all the integration methods until finalised (lucky for me!).

Secondly, you can rely on the earlier View integration method Zend_Controller_Action::render() using:

```
$this->view->loginError = true;
$this->render('index'); // tell ViewRenderer (if enabled!) to render same as indexAction()
return;
```

Viva la ViewRenderer!

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 19:24

Nice article, I find it very useful. Zend_View and stuff is very discussed subject nowadays, some people may find the ViewRenderer complicated. I must confess that I haven't completely understood all the system yet.


One more thing, just a small mistake:

`$viewRenderer->setViewSuffix('.tpl');`

should be in my opinion

`$viewRenderer->setViewSuffix('tpl');`

That dot is extra, I believe. Anonymous on Jun 6 2007, 00:15

Quite right! That dot is not needed since it's assumed already by the ViewRenderer code - will correct shortly if I find the time 

Anonymous on Jun 6 2007, 00:52

Personally, ViewRenderer has left a very sour taste in my mouth. Up until now I was enjoying the Framework with it's high degree of loose coupling, at least compared to its competition.

But switching to version 1.0 RC1 I found a small step backwards with ViewRenderer.


I mean, it's nice to have a class that can automate output, I'm all for that. Even before version 1.0, I took all of the suggestions of the

Framework manual to heart, and built my directory structure and template naming theme accordingly. I extended the Action Controller somewhat to allow more advanced rendering and auto initialization settings, but the one thing I made sure is that actual echoing **and** rendering was left to the action itself.

You see, the problem is that I'm building an AJAX web-application. Most of the Actions perform behind-the-scenes manueveres, sometimes output is delivered in JSON, sometimes not, sometimes NOT at all. For some reason, I could not turn off ViewRenderer auto rendering. The methods that are supposed to handle it (setNoRender() and setNeverRender()) are not working. Or rather they are working, it just not having any effect - I tried manually changing the default "false" to "true" values for the _neverRender and _noRender ViewRenderer class memebers, and the ViewRenderer was still doing its thing, rendering everything in sight, making my web-application a complete mess. I had to shoot it and disable it globally for all eternity, May it rest in peace.

At the end of the day, all I can say is nice try, but no cigar on the ViewRenderer.

By the way, the other important change for the version 1.0 that might be somewhat overlooked is the return of Zend_Filter_Input in a new and improved design. Useful. Anonymous on Jun 6 2007, 09:06

I'd be wary of Zend_Filter_Input. Bryce Lohr has another proposal for post 1.0.0 which is quite an improvement once the interface is narrowed down 

. Anonymous on Jun 6 2007, 17:22

hi Padraic

One question, previously i was subclassing the Zend_View class so that i could easily render a standard header and footer around the main content (the view script) without having to include that in every view script.

With the ViewRenderer, how would i go about this? Is it possible to pass the view object to the ViewRenderer, and override the render() method, to also render my other two templates? Or do you know of a better way?


thanks

dave Anonymous on Jun 7 2007, 07:20

oops, i meant to add, subclassing the Zend_Controller_Action class, and overriding that render() method. Anonymous on Jun 7 2007, 07:37

I would stick with your subclassed Zend_View. It's very clean to keep things in a View class instead of depending on the Controller. You would just need to manually set it on the ViewRenderer (as I did above for the Zps_View_Smarty object). Otherwise it should function same as before.

Admittedly I do have a strong opinion about avoiding too much Zend_Controller reliance for rendering. ViewRenderer violates that, but at least it violates it more honestly and offers methods for disabling/modifying it. I'd still be inclined to leave it as is as much as possible.

If you follow the mailing list for ZF you'll see a tortuously long email from myself in a day or two. I'm hoping to try and push a more complex View system into a proposal. Not sure if it will stick but maybe it will get more people thinking 

. Anonymous on Jun 7 2007,

15:51

cool, thx

good luck with your proposal, although it'll be pretty tough to get something in for 1.0 Anonymous on Jun 8 2007, 18:32

Definitely wouldn't get added until well after 1.0.0 

Anonymous on Jun 8 2007, 20:55

Eran is almost right. It is a LARGE step backward. Loose coupling is essentially dead and for no reason. This new paradigm doesn't solve any particular problem and, as he found out, it narrows the scope of the framework. Then main problem is the View has broken out of its encapsulation and View related issues have spread to the dispatch shutdown loop, the new viewrenderer and who knows where else. This is EXTREMELY disappointing. It was also done with a level of hubris and arrogance that would shame Microsoft and worse The Bush Administration. All this could be avoided by simply extending the View. Anonymous on Jun 22 2007, 09:12