

Wednesday, June 27, 2007

HTML Purifier 2.0.0 - new version of the PHP HTML filter library

There are a lot of cool people on the PHP Developer Network forums. One of them is Edward Z. Yang.

On June 20th, [Edward released HTML Purifier 2.0.0](#). [HTML Purifier](#) is a standards compliant HTML filter written in PHP. It uses a whitelisting approach and outputs standards compliant code, even if originally scrambled into an unintelligible mess. It uses functionality in the background based on Tidy's behaviour (so your preferred DTD is adhered to while filtering).


It's purpose, in case you're not familiar with HTML filter libraries, is to filter HTML user input to ensure it only includes whitelisted elements and attributes, and absolutely no XSS. The site contains a page dedicated to tests against the infamous <http://hackers.org/xss.html> exploits. There's even a demo page for testing it against your own possible exploits.

For my own part, HTML Purifier is probably the finest HTML filtering library in PHP at the moment. Its design is top notch, it's a doddle to extend, and the API is intuitive for whitelisting (see the [Advanced API](#)). The library's website has a stock of documentation for users and developers - including some useful tips for improving performance. Go get a copy and give it a whirl.

Posted by Pádraic Brady in PHP General, PHP Security at 00:28

Monday, June 25, 2007

Refactoring an OpenID Library

Since I have a three track mind (but only one dedicated to PHP), I'm stuck with another OpenID post. Over the weekend, I managed to grab a few hours to dig around my OpenID library with the ultimate development tools: patience and experimentation. There are also a few integration tests as a safety net though 

. So I will be sending a copy around to a few people including Dmitry who has posted a Zend Framework proposal for OpenID on the wiki (which is currently down, as usual, so I haven't managed to add a comment on it yet - should be more stable in the near future I hear).

The refactored library changed a few aspects of the original code I wrote last Autumn. The original placed a lot of responsibilities in the Consumer class which meant it was very difficult to actually change things without knock on effects elsewhere. What I've refactored towards is a splitting of the OpenID process based on three categories: Request, Redirect, Response. The OpenID Specification is all about communication, and this type of splitting seems to have worked out very well. The other change was to centralise all values which are effectively constants to the top-level OpenID class. This class also manages the current version via a static public method - `OpenID::setVersion()`.

So what does the library not include (always a good question!).

1. XRI Support is not complete. I need to add an interface to the Yadis object to extract an XRI's Canonical ID which is the actual value used as an OpenID claimed identifier.
2. Stateless Mode is not supported. Also called Dumb Mode, it's necessary when the server cannot store state between requests. Since PHP is perfectly stateful I omitted it for now.
3. Good session integration for frameworks. The library currently directly accesses `$_SESSION` under an "OPENID_SESSION" namespace. For the Zend Framework and others which have a Session class (e.g. `Zend_Session_Namespace`) this may not work well if sessions are being written to the database, or session expiry dates being set.

What does it support?


It supports almost the entire OpenID 2.0 Authentication Specification (draft 11 since it's not finalised). It also transparently covers a few common edge cases such as OP Servers which do not always return an "ns" namespace value in responses. It also downgrades and tracks the association type when an OP cannot use SHA256 (e.g. a number of PHP4 based providers). It also supports OpenID Extensions - you can add extension types quickly to the current list (i.e. the Simple Registration Extension (sreg) for now).

Improvements over original library?

It now has wider support for Diffie-Hellman and HMAC using one of mhash or hash extensions (both cryptographic algorithms are segregated to their own class also). Classes are now lowly coupled, so maintenance and individual class testing is simpler. Logic is more dispersed across smaller methods - again making it easier to maintain, test, and modify discrete chunks of logic. I've also centralised the constant values, and amended the API to be similar to that used by JanRain's PHP4 library. There are of course differences from JanRain's API but the flow is very similar.

Other stuff?

Except for some API improvements I should make, it includes a functional Yadis implementation. A task for later is allowing the OpenID library check whether an OP supports extensions like Sreg by checking the list of available Service types discovered by the Yadis protocol. This only needs a minor change - I first need to check whether a missing sreg service in an XRD document is a definitive sign it is not supported, or whether it's entirely optional to show it as a


service. (More spec reading this evening 
).

Aside from an example to be passed to the Zend Framework proposal, the library in it's refactored form will be proposed to PEAR. I think I'll propose Services_Yadis first since it's a component of the larger OpenID scheme.

Posted by Pádraic Brady in Openid and Yadis, PHP General, PHP Security, Zend Framework at 19:03


Friday, June 22, 2007

OpenID 2.0 Library - to PEAR, Zend or both?

As a follow on from my previous entry about OpenID in the Zend Framework, I've been in brief contact with Dmitry Stogov across a scattering of emails. Dmitry posted his OpenID proposal for the framework over at the Proposals Wiki earlier in the week. I'll be passing comments on this whenever possible (i.e. whenever the phone stops ringing this weekend 

).

The main differences between Dmitry's code and my own are hard to really explain. Dmitry's sample code is very brief, to the point, and gets OpenID 1.1 done. There is no OpenID 2.0 compatible consumer since the code omits Yadis, but it should catch most HTML based discovery elements from an OpenID alias. It doesn't capture everything perfectly however, and there's still a ton of coding and test work required to get it working robustly. The difference to my library is pretty big as a result. I do a lot of abstraction in my coding so I don't have a single Consumer class - I have about 5 classes working in unison handling various authentication stages. I also have the cryptographic and math logic split into free standing components (see the PEAR proposals below for example). It's a heavier set of code which evolved over a much longer time and so naturally covers a lot more of the specification and it's edge cases.

It's actually very hard to comment constructively rather than simply handing over my code which probably says a lot more all by itself. I really really need to get my code refactored fully at the weekend so I can slap a New BSD License on it and hand it over - I can't keep referring to empty air 

Now what's this about PEAR?

After my weekend blog entry I took onboard Greg Beaver's prompting in the comments. Greg noted that PEAR was looking for an OpenID implementation. One of the reasons I had for not going to PEAR before was that JanRain (the maintainers of PHP OpenID - a PHP4 library) had a proposal over there. It looks like that proposal has now stalled and JanRain are no longer interested in pursuing a PEAR route. Among the listed reasons are the PEAR Coding Standards and the difficulty of migrating the library to PHP5. I think their reasons are a bit odd, but their choice. The PEAR conditions make a lot of sense if you intend setting a high quality standard for code.

Anyway, I've agreed to port my OpenID library to PEAR as a PHP5 package. I checked with the mailing list, and the approach I've taken in splitting the library across a number of freestanding components hasn't seen any objections. On the flipside, it does help by providing upgrades to existing PEAR Encryption packages which are not yet migrated to PHP5 versions.

To get the process rolling I have proposed two packages so far: [Crypt_HMAC2](#) and [Crypt_DiffieHellman](#). These are the two cryptographic areas an OpenID Consumer must implement. The code is fairly intact from what would have been the Zend Framework Zend_Crypt_* proposals.

The next PEAR proposal will likely be the heavyweight Services_Yadis, a reflection of a similar proposal for the Zend Framework. Yadis is an XML based service discovery protocol - required for OpenID 2.0 Consumers. In my OpenID implementation it's also the component solely responsible for managing and validating XRI identities and finding their OpenID required CanonicalID values.

In using PEAR for years, I've never proposed a package before. It was quite a pleasant surprise to find out how easy it was. Following the PEAR Coding Standard is already a habit and building example installable packages for a proposal just requires minimal reading of the [PEAR_PackageFileManager](#) documentation,

which is useful in generating those package.xml files which tells PEAR about the package and install locations of files. I also seem to have joined up to PEAR just in time for a future announcement regarding PEAR2. [Travis Swicegood already blogged about this earlier.](#)


Posted by Pádraic Brady in Openid and Yadis, PHP General, PHP Security, Zend Framework at 22:38



Monday, June 18. 2007


Zend_Service_Openid Is dead; But It's Reincarnation Is Coming


Back in February of this year I started the process of proposing a PHP5 OpenID and Yadis library for the Zend Framework. This was to be based on a proprietary implementation (IP clean, I hold the copyright) I had started working on. After posting some questions to the mailing list before adding any additional formal proposals, I have learned another OpenID library for the Zend Framework is also in progress within Zend.

The upshot of this, given it's a Zend sponsored library, is that I feel like there's little choice but to abandon my own version of a proposal for the framework. So it's status is "withdrawn" at this moment as is Zend_Service_Yadis, being a linked proposal. The library itself is not being abandoned, far from it as it has always been my intention to release it under a New BSD License and that plan is not wavering.

Now it's been a few days since I learned this. So I've calmed down a little ;), and look forward to a review of the Zend code which Andi kindly asked Dmitri Stogov to supply. I can't help but pass some comments however (not as "rant and rave" as I might have posted on Saturday though 

). The first is that it is frustrating to have a second Zend Framework OpenID library start and for the team behind it to miss the existing effort totally. I had posted to the mailing list back in February, put a formal Zend_Service_Yadis Proposal on the Wiki in plain sight (a significant piece of supporting OpenID 2.0), and blogged about it a few times. The simplest of research tactics (Google for "zend openid" and guess who owns the first page of results 

) could have avoided this. Secondly, this second project has never been notified to the Zend Framework mailing lists, something I had thought was part of the Proposals Process before writing a formal proposal. This entire scenario only ever cropped up because a decision was made away from the public eye. If it had been announced, I could have joined the team and contributed my ideas and code at that time.

It's very disappointing this has occurred and I urge Zend to publicise their future Zend Framework proposals rather than developing them in the dark and springing a last second surprise. It could save some other poor soul from a shock in the future 

However what remains to be seen is how the Zend proposal plays out. My own proposal took a particular approach to OpenID. OpenID is a complex set of specifications, something the JanRain library makes abundantly clear. But in isolation, its parts are relatively simple. My proposal would have advocated the separation of concerns to a large extent - something I feel has been amiss with the framework in many places. The core of this proposal has been to pick apart OpenID 2.0 and split it into many components which are completely decoupled from an OpenID specific class, and therefore are completely reusable Framework-wide and also add far more value than highly coupled methods.

If the Zend OpenID library does not take this approach, then it still leaves the field open for a fair proportion of my library to end up being proposed for the Framework (e.g. Zend_Crypt, Zend_Math_BigInteger) so that components have a central reserve of reusable code. For example, Simon Mundy seems very supportive of a standalone HMAC implementation for Zend_Mail (something a Zend_Crypt proposal would include). How many others would like to see RSA, XTEA, Diffie-Hellman (with Wez Furlongs awesome incoming openssl support included!), and company standing alone and begging for reuse?

Going by very recent information from Dmitry Stogov at Zend, it's even possible the original Zend_Service_Yadis proposal I initiated can still make it through the proposal process. So I may yet publish my non-OpenID proposals to the framework wiki.

In the meantime however, I'm left with an OpenID library in PHP5 that's nearing completion (OpenID 1.1 is just about there, and 2.0 a short distance beyond that) and appears bereft of a framework route. This, fortunately, was not the only distribution path I had considered so there's still a few possibilities to explore.

If you're one of those following this saga since February, stay tuned



. A PHP5 OpenID library (sorry, two




) are incoming.

Yep, silver lining and all that...

Posted by Pádraic Brady in Openid and Yadis, PHP General, PHP Security, Zend Framework at 19:35

Complex Views with the Zend Framework - Part 6: Setting The Terminology

Regular readers have had a gentle (I hope!) introduction into the world of Complex Views over the course of the last 5 parts of this ongoing series. We've started with an overview of Zend_View and its shortcomings in assembling web pages composed of many reusable, and even nested, elements. Over the previous 4 parts we delved into some theory, described a few solutions, and learned that I need to unit test my examples more exhaustively 

. Everyone's a winner

here at Astrum Futura, even me!

Part 6 now takes our previous fragments of theory and attempts to stitch them together into a cohesive whole - a description of a possible end solution. Yes, I know code talks loudly, and we'll get there with an almighty bang in Part 7 (which I promise is no more than a week away). Until then it's more verbosity.

So let's get stuck in!

Half the trouble in conceiving of a extended View system is agreeing on terminology. Everyone has their own idea of the basic concepts, but without names we're left with vague descriptions. Here I'll throw out some terms, some borrowed, others mangled slightly, the rest fairly obvious. These terms all describe specific rendering processes. Methods of capturing presentation logic in neat parcels which carry specific consequences, follow object oriented practices, and provide (we dare hope) commonly sought functionality.

The full list (and please feel free to add/suggest changes to these in the comments or by email) is:

1. Includes
2. Partials
3. Dispatches
4. Layouts
5. Placeholders
6. Response Segments

Includes

The mighty Include is the mainstay of rendering in PHP. Smarty has the {include} tag, Zend_View allows for template inclusion using secondary Zend_View::render() calls, even stark naked PHP is a master of the Include using the aptly named include() function.

Includes are simple to understand - they drag additional templates into the same scope as the calling template at some specific location. In essence, all variables the calling code has access to, are instantly available to the included file. This does however add a little smell - all includes are almost by definition tightly coupled to the calling code. It's stitched into the fabric of any class that uses include(), such as Zend_View::run(), with nary an interface in sight.

The old reliable...

```
<?php echo $this->render('article/intro-blurb.phtml') ?>
```

Partials

Partials are not dissimilar from Includes, except that they have one very specific characteristic. They are decoupled from the calling class. This can be managed by assigning each Partial to it's own respective

independent View object. The relationship is a typical Parent->Child, where the Parent View creates a new Child View (one template per View mind) wherever it uses a Partial.


Because both are independent entities, the Parent View must provide all necessary data to the Child to enable rendering, usually when the Partial is rendered. If you can follow this ingeniously vague description, it's the basis of the Composite View Pattern I discussed previously.

Partials also have a second characteristic. You can dynamically change their context when rendering. Imagine creating a list of articles on an HTML page. Your primary View renders everything except the article list. At that point, it iterates across a list of article data (the Model) and calls the same Partial each time, passing it each article Model in turn. The Partial will simply render the same template over and over, but applies the new data each turn.

Example Partial called in a foreach loop with a new context each time:

```
<?php foreach($this->articles as $article): ?>
    <?php echo $this->partial('_article.phtml', array('article' => $article)) ?>
<?php endforeach; ?>
```

As I've noted before you can call these "Glorified Includes" since they are so similar. The main difference is that handling Modules is managed externally either through convention or configuration. Unlike `Zend_View::render()` there's no path definition required. Also the many benefits of an independent View object (nested Layouts, Placeholders, etc.) are all intact.

If you're watching the code, these samples are suggestions of an interface. In Part 7, we'll define something concretely with "real" code 

. Above we're employing a convention the templates names starting with an underscore are special in that they are not intended for rendering as a primary View.

Dispatches

The Dispatch rendering process is a heavy weight. It's purpose is to generate a full bore Controller dispatch using a customised request. It can be used similar to a Partial or Include, but requires a full dispatch cycle to a Controller and Action. As a result it has several possible flaws - it depends on the presence of a suitable Controller component (i.e. `Zend_Controller`) which means it's not likely portable, it will suffer performance wise from all the extra work required to dispatch a request, and it has very few advantages over a Partial.

One possible reason for using it is if a View has a dependency on some authorisation/authentication system like `Zend_Auth` and `Zend_Acl`. Even here though I would still recommend using a View Helper if feasible to query both. Another potential use if the View is being rendered from an external source (e.g. integrated third party application). So while it's imperfect, it's still quite useful to have available.

Dispatch and echo the rendered view from `ArticleController::adminlinksAction()`:

```
<?php echo $this->dispatch('adminlinks', 'article') ?>
```

Layouts

The subject of Part 5 of this series. Layouts define common markup which will encapsulate any number of Views. While it's easy to use Includes to add Layouts, you would have to copy the Include code to every single template (the traditional include header/footer code). This is hardly the most maintainable method of doing things. What if some Views suddenly need a different header? Will you run off to edit 100+ templates? For this description, Layouts are "secondary" templates, into which the main template/View being rendered is

implanted before being echoed to a client.

The funny thing about Layouts is how obvious and useful they are once you figure out what they really do. Not only can they be used as an overall template for presentation of a full HTML page, they can also be used to capture the layout of subsections or other nested elements. Because Layouts are a top-level application setting, they are simple to manage and assign to different groups of Views with the right setup.

For this discussion we need to make one assumption for later. Layouts are always the last rendering step when processing any View. This allows nested template the opportunity to make subtle changes to the Layout before it's rendered. We'll find a use for this render-order with Placeholders.

In Zend_Controller_Action app specific subclass, or another appropriate place:

```
$view = new Zps_View;
$view->addBasePath(APPLICATION_ROOT . DIRECTORY_SEPARATOR . 'default/views');
$view->addScriptPath(APPLICATION_ROOT . DIRECTORY_SEPARATOR . 'default/views/layouts');
$view->setLayout('application.phtml'); // ./default/views/layouts/application.phtml
```

An example layout with a hook method call where the View rendered output is inserted:

```
<html>
<head><title>My Page</title></head>
<body>
  <div id="page">
    <?php echo $this->content(); ?>
  </div>
  <div id="footer">
    Copyright &copy; 2007 PB
  </div>
</body>
</html>
```

Placeholders

A common problem with simple Layout systems is that they cannot capture the full context of a page. How can a Layout know whether a specific View demand additional layout properties like extra CSS files, that once off AJAX script, or a few extra meta elements?

Placeholders define a position where templates can append extra markup if they wish. Since Layouts are rendered last, they can allow for other templates of the primary view to make such additions. Placeholders can also be used to define default markup for rendering, unless overridden by a template, or unless some condition is met.

A really simple example is a page composed of a Layout, and several nested Views. If one View presents a blog page, the designer may want to append a element in pointing to the local RSS/Atom feeds. It's only shown for that single page. Using a Placeholder in the application wide Layout would enable this very easily. The blog template could set the Placeholder's value, and the Layout when rendered would insert that into the header dynamically.

Example Layout with a placeholder where templates can append child elements:

```
<html>
<head>
<title>My Page</title>
<?php if($this->placeholder()->has('HEAD')) echo $this->placeholder()->get('HEAD') ?>
```

```
</head>
<body>
  <div id="page">
    <?php echo $this->content(); ?>
  </div>
</body>
</html>
```


A template for displaying a blog which needs to add an RSS link to the section of the Layout:

```
<?php $this->placeholder()->add('HEAD', '<link rel="alternate" type="application/rss+xml" title="RSS"
href="http://www.example.com/rss.xml" />') ?>
<div id="content">
  <!-- Some content to be inserted in Layout -->
</div>
```

Response Segments

Response Segmentation was introduced into the Zend Framework some time ago. It's a Controller driven process which assigns the rendered output of a View to a named segment in the Response object, which is then organised into the full page before it's echoed to the client. In a sense it's not dissimilar from Placeholders, except that the segmentation is not influenced from within the View object tree itself, being delegated to the Response object in Controller instead.

You can [read all about it in more detail](#) on the Zend Framework manual.

So there you go. 6 terms to keep in mind (pending feedback from you, the Reader) for Part 7. Now that we have these terms, several of which you are acquainted with if following this blog series, some of which are likely new, and some sample interfaces we have opened the gates to defining some starting Unit Tests. I'll spare you the TDD process 

. Part 7 will elaborate on most of these with cold hard (and colourfully commented) source code.

As always, I value your comments and emails. It's always great to hear of others' experiences in the field.

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 05:06

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