


Friday, April 27, 2007


Complex Views with the Zend Framework - Part 3: Composite View Pattern

In the previous two parts of this series of blog posts, I've been looking at the task of implementing complex views with the Zend Framework. Part 1 [looked at what complex views are, what support for complex views the Zend Framework offers out of the box, and a reference to two design patterns](#) useful in adding further support: View Helper and Composite View. In Part 2, [I tackled the View Helper design pattern](#).

As a brief recap, the View Helper pattern promotes the idea of creating helper classes which allow the View layer of a Model-View-Controller application to directly query the Model layer in a read only fashion - effectively bypassing the Controller layer altogether. In such a way, the pattern offers the option to programmers using the Zend Framework to avoid using the current method of nesting Controller Actions with the `Zend_Controller_Action::_forward()` method which can be an overly complex approach for parts of a View we know are common to many pages.

In this post, I offer a brief explanation of the Composite View pattern. It's beyond its scope to show an implementation using the Zend Framework though that's what I'm building up to accomplish in a later blog entry 

If you haven't already guessed, the Composite View pattern is related to the Composite Pattern as defined by the Group of Four (GoF). If you're not familiar with the Composite Pattern, an [excellent PHP article for it was published on Zend Devzone](#) and is worth reading.

Here's a UML diagram of the class relationship although we do things a little different below 



In brief the related Composite View pattern organises View objects into a nested tree structure, in which both parents and children implement some common interface (let's say `render()` for now). You can think of each View representing a single element of the overall page, with parents representing sections of the overall page which may contain other component elements. The essential characteristic, inherited from the Composite Pattern, is that the common `render()` method can be called from the root View, and this method call propagates across all nested objects in the tree until the output from all child Views is finally aggregated into the overall page we intend sending to the browser.

With this description we can make a few assumptions regarding the Zend Framework. The main one being we will need multiple instances of `Zend_View` to pull it off (as distinct from multiple Controllers). Also all this talk of objects hasn't explained how the nesting is controlled. We'll handle the second now, and come back to the `Zend_View` multiplicity after.

There are two methods of handling nesting of View objects. The first follows a purely object based scheme, where objects are nested and combined at runtime. The second passes control over nesting to our friends - the templates. The template method basically involves adding a function (think of the Composite Pattern's interface for "composites") to include another template. Here we'll call it "attach", though it could be anything you prefer. We'll look at this approach below since it's the simplest one to follow if you're familiar with `Zend_View` templates. In addition, the template approach is probably the easiest for programmers and

designers to handle since it allows the View layout to be put in place around all these composite View includes.

Now, we may appear to have come full circle here - this sounds suspiciously like a glorified "include" statement but there are differences. The main one being each composite "attach" creates a new View object which can be sourced from other Modules, and with Module specific helpers and filters (and of course Model accessing View Helpers!). If we assumed the composite method was "attach" (common Composite Pattern method), a composite template could look like:


articlelist.phtml (part of the imaginary Articles application Module):

```
<?php $this->attach('stdheader.phtml', 'default'); ?>
<div class="body">
  <?php echo $this->attach('overbody.phtml') ?>
  <div class="content">
    <?php foreach($this->articles as $article): ?>
      <div class="article-abstract">
        <span class="title"><?php echo $article->title ?></span><br />
        <?php echo $article->summary ?>
      </div>
    <?php endforeach; ?>
  </div>
<?php echo $this->attach('underbody.phtml') ?>
</div>
<?php echo $this->attach('stdfooter.phtml', 'default') ?>
```

overbody.phtml:

```
<div class="overbody">
  <div class="headlines">
    <?php echo $this->attach('last5headlines.phtml', 'Blog', array('feedUri'=>
'http://www.planet-php.net/rss/')) ?>
  </div>
</div>
```

Here we have two templates, both utilise an imaginary attach() method from the View object to attach new sub Views to the template at specific locations within the layout (you're right in thinking attach() will handle rendering of the child View). It also allows for a template to attach Views from other Modules of the application if required (and would transparently manage the different View settings for that Module). I've assumed the lack of a "module" parameter means the attached View should come from the current Module only. The last overbody.phtml template allows for more dynamic parameters targeting the RSS View Helper utilised by the Blog Module's last5headlines.phtml template where we're using the View Helper pattern to grab headlines from the given RSS feed. We visited very similar functionality in my last post.

Of course in all this we haven't ruled out Controller nesting completely. That too is still possible, even alongside View nesting and in a very similar fashion. Since each element attached is an independent entity you can mix and match what you want to use, even from other Modules. Of course with all this floating around we still haven't seen usable code yet! Well, that one's for another blog entry 

Back to the multiplicity issue for a moment, i.e. the presumed use of multiple Zend_View instances. You can guess that Zend_View will need to be subclassed and/or a standard helper class containing the attach()

implementation added. There are other issues also. For example, each View object needs to be independently configured before use - i.e. for paths, encoding, etc. In Part 4 of this series I'll take a stab at adding finer control over creating View objects in a simple reusable fashion.

I'm sure many of you will already suspect the design pattern that will involve 


Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 15:58

Complex Views with the Zend Framework - Part 2: View Helper Pattern

Part two of my ongoing look at the View layer of the Zend Framework turns its attention to the topic of View Helpers. The Zend Framework manual provides a fairly narrow definition of its helpers which indicates they enable complex tasks, like generating form elements, to be extracted out of views into dedicated helpers. Here I'll try to explain in greater depth the View Helper pattern which is another of those patterns in the J2EE catalog, and which adds to the range of tasks View Helpers are capable of performing.

The simplest explanation is in the separation of layers enforced by the Model-View-Controller pattern. In a typical web application using the Zend Framework, a URI is used by the framework's Controller architecture to select an action on a `Zend_Controller_Action` class to execute. This action method will then normally access the application's Model to fetch any data it requires, operate on that data, and pass a subset of it to the View (for use in rendering templates). It's clear to see so far, that the View by default relies on the Controller to pass it the data model it needs when rendering templates.

As in my previous post however, the View may also include any number of elements (header, footer, widgets, etc.) common to ALL pages. Many of these may require data of their own.

So what is the problem? Anytime a partial View needs extra data (the View's Model) it needs to push calls to extra Controllers (following the current practice for the framework) in order to get that Model. This involves yet another complete dispatch cycle, with any number of classes, plugins, and operations involved. Yet in most cases this is completely unnecessary - why not just let the View request data from the Model directly? 

In simple terms, the View Helper acts as a middle-man sitting between the View and the Model. It's job in our scenario is to replace the need to nest Controllers, and give that nesting/layout control back to where it belongs - the View layer. When a View (or partial View) requires Model data not supplied by the current controller, it calls upon a View Helper to go fetch that data independently of any Controller.

Let's take a simple example. While building a blog, we've decided to add the last 5 entry titles from Planet PHP to the bottom of the page - every page in fact. In keeping with promoting reusability we create a generic `Zps_View_Helper_Rss` class which can consume RSS, and to which we can add extra methods in the future should they be needed for other Views. This is slightly different to the current view helper description in the framework manual - here we can offer a full selection of public methods forming a fuller (read-only) interface to the underlying Model - the RSS XML.

```
require_once 'Zend/Uri.php';
require_once 'Zend/Feed/Rss.php';
class Zps_View_Helper_Rss
{
    protected $_channel = null;
    public function __construct($url)
    {
        if (!Zend_Uri::check($url)) {
            throw new Exception('RSS URL is invalid!');
        }
        $this->_channel = new Zend_Feed_Rss($url);
    }
    public function getTitles($number)
```

```

{
    $titles = array();
    $count = ;
    $number = intval($number);
    // Zend_Feed_Abstract implements Iterator!
    while ($count <= $number && $this->_channel->valid()) {
        $titles[] = $this->_channel->current()->title();
        $count++;
        $this->_channel->next();
    }
    return $titles;
}
}
}

```

Now comes our sample template - one of those recurring Views:

```


<?php $feed = new Zps_View_Helper_Rss('http://www.planet-php.net/rss/'); ?>
<div class="feedtitles">
<ul>
<?php foreach($feed->getTitles(5) as $title): ?>
<li><?php echo $title ?></li>
<?php endforeach; ?>
</ul>
</div>

```

And not a controller in sight... 

Of course the Model can equally be running from any datasource including the database, so there's a lot of flexibility for these View Helper classes. I haven't done so here, but you can probably still fit them into the current view helper convention with the single public method (in our case rss()) just returning instances of the object.

Of course, there really are times when the recommended Controller _forward() (or a viable alternative) can still be useful. Reliance on plugins was suggested on the mailing list, for ACL perhaps. But the point here is that controller nesting is not required except for those exceptional cases where a View Helper just won't cut it.

Edit: I figured it be save confusion by noting this helper could equally run from a cache of the selected RSS source which is updated separately at regular intervals (hail to cron!). Save the server making the trip on every request 

Posted by PÃ;draic Brady in PHP General, PHP Security, Zend Framework at 06:34

Thursday, April 19, 2007

Complex Web Pages with the Zend Framework?

27 March 2008:

With the inclusion of Zend_View Enhanced as first documented, discussed and publicised in this blog series, in the Zend Framework as of 1.5.0 I'd like to thank everyone involved in the process. A few of you pioneered it's implementation and provided immense feedback which tailored the proposal to some specific needs, others kept bringing it up on the mailing lists and to the attention of the Zend reviewers, and others still spent a lot of time converging ideas towards a unified proposal. It just goes to show that the community really does have the power to influence the big picture and get stuff done!

The rest of this series can be accessed as follows:

<http://blog.astrumfutura.com/archives/282-Complex-Views-with-the-Zend-Framework-Part-2-View-Helper-Pattern.html>

<http://blog.astrumfutura.com/archives/283-Complex-Views-with-the-Zend-Framework-Part-3-Composite-View-Pattern.html>

<http://blog.astrumfutura.com/archives/285-Complex-Views-with-the-Zend-Framework-Part-4-The-View-Factory.html>

<http://blog.astrumfutura.com/archives/288-Complex-Views-with-the-Zend-Framework-Part-5-The-Two-Step-View-Pattern.html>

<http://blog.astrumfutura.com/archives/291-Complex-Views-with-the-Zend-Framework-Part-6-Setting-The-Terminology>

Recently I've been involved in a long discussion about the Zend Framework on the [PHP Developers' Network forum](#). Our approach was to pick a simple application (we decided to borrow the Java BluePrints Pet Shop for J2EE) and starting from a basic "Hello World!" example for the Zend Framework work towards a fully functional example. Of course, one of our goals wasn't just to "do it", we wanted to explore the framework in greater detail, and identify how best to use, misuse, subclass, and where it was logical to even replace components should they prove deficient for our needs.

After wrangling about configuration, the advantage/disadvantage of build tools (I love Phing and cannot survive without it!), the location of the bootstrap file, and a few other odds and ends we finally put up the "Hello World!" example in subversion. Many thanks to [Chris Corbyn](#) of [Swiftmailer](#) fame for contributing the repository!

<http://w3style.co.uk/devnet-projects/pet-store/trunk/>

We then decided to look at how the Zend Framework implements Views. In essence, the framework isn't as developed in that respect as its peers. A simple page is easily built using a Zend_View instance, a PHP/HTML template for a list of entries, and a few view helpers and filters. After that however, a complex page becomes progressively more difficult. This is complicated in part by the growing practice of instantiating Views using a helper function on the Controller - unfortunately this is unusable since it introduces coupling making re-use more difficult in other applications where the View has been subclassed.

Back on track, the main problem of a complex View, is that the current Controller is only aware of a subset of its own required Model (data) and the current View. So how do you get the View to include extra sections - for example, details from Technorati for your blog - which are common to ALL pages?

The Zend Framework currently suggests using a Controller's `_forward()` method - basically if your current View needs data from Technorati, then forward from the current Controller to a "TechnoratiController" to fetch it and assign it to the View, and then `_forward()` back. This works for simple things - but what if there are three, or even more sections? What if some sections, have additional embedded sections? How in heaven's name are you supposed to track all those `_forward()`'s?

While pondering the problem, we all agreed using the Controller `_forward()` was not a good idea - it's prohibitively complex, forces Controllers to become aware of other (possibly unrelated) controllers, and in general doesn't promote reuse. I haven't measured the performance impact of such a tactic but it can't be good. We also determined the Zend Framework's view helpers were of limited use being simple classes designed to offer a single public method to templates.

Finally we decided to visit two design patterns to help name the problem and offer a possible solution: Composite View and View Helper. So far we've just implemented a very basic Composite View (KISS) to

further the brainstorming. But it shows a lot of promise in offering a simple, effective solution to building complex web pages with the Zend Framework - granted it breaks the normal practice, and takes subclassing to introduce, but it's a reusable solution.

I'll visit Composite View in more depth soon - for the moment here's the J2EE BluePrints page for the design pattern: [Composite View](#). If anyone knows of a really good example in PHP or Ruby, it would be great to hear about it!

On the View Helper bit - the Zend Framework seems to persist a belief that only Controllers should interface with the Model layer (i.e. Database and associated Model classes). This Model-Push strategy can be complemented with an equally valid Model-Pull strategy - where View Helper classes have the ability to directly access and read data from the Model to pass to the requesting View. This completely avoids the fickle tactic of Controller forwarding and the complexity it tends to introduce into an application.

Stay tuned for a deeper look at both these patterns - specifically how they can be brought to bear on the Zend Framework with a little subclassing of `Zend_View`.


Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 20:36


Friday, April 6, 2007

S.T.A.L.K.E.R.: Shadow of Chernobyl

I finally got my copy of the game at the weekend, played a little, got the patch, found my save games were incompatible, restarted, and replayed (sigh) a little more. It's an interesting game, full of interesting ideas and sometimes exasperating combat. Surprisingly, given the level of buginess so evident (and definitely not addressed by that patch) it's a good game overall. So it gets a thumbs up from me, with a sidenote that you should have a pretty decent rig to really get the most out of the game, and should have a certain tolerance for the inevitable bug or two.

Even so, there are some spectacular misses. Gameplay stands by itself very well, so these aren't as important as they would be to a lesser First-Person Shooter, but still... The most obvious is that the settings menu for Video Options is a mess - Antialiasing and Anisotropy are non-existent. One is simply not working, the other is useless because STALKER's game engine uses a weird rendering system that doesn't support it. I've been reading the TweakGuide.com examination released earlier and all is not lost - there are tweaks you can make since the game is capable of being modded by extracting the content archives (much like Oblivion and Medieval 2). TweakGuide also comments on the effectiveness of some low-level options (which don't seem to work better than the "medium" ones). Also HDR, important for GeForce 6 series GPU's, is not a separate setting to be disabled.

Back to the game play - STALKER excels at a few graphical additions (even with the above). The most obvious is the weather. When it rains, it pours. When it let's loose lightening, your video card takes notice. The second is the day/night cycle - and night is something to really take care with. Unlike Oblivion, or other half-assed attempts at day/night cycles, STALKER just takes the realistic of approach of making the sun set and rise at genuine times. Simple, effective. Half Life 2: Episode One (the level with the underground garage populated by Zombies) is a close match for how you'll feel when night arrives. Even bumping up the gamma level doesn't help with visibility. Even the flash light isn't much help beyond a certain range. This really packs a cool punch - especially when you have to spend the next 7 hours of gametime (a lot less in real time of course 

) stumbling around in the dark hearing the howls of a pack of mutant dogs and barely being able to see them when get close. Synchronising attacks with lightening strikes becomes a quickly learned skill 

The other cool thing is the "ALife", watching mother nature at war with packs of Blind Dog mutants run into confrontations with those mutant boars around this corner, or something even weirder around another - all at random in a natural fashion. It's cool to watch - especially once you realise those creatures have defining habits. At the start, I remember finding the body of a fellow Stalker surrounded by mutant dogs. After shooting up two, and scattering the rest I returned some hours later to find the escapees of my first attack had returned to the body and were apparently ravaging it - and their dead compatriots. I almost felt sorry for the poor sod whose pixelated body was being gnawed on, I was even tempted to use the "drag body" action to move him to a place safe from the local man-eating wildlife.


A few places where it goes wrong includes the mission system. Missions are obviously not finished in full - most are given by a handful of Traders, or Stalker leaders. Most come with no other in-game scenery, events, or NPC interaction other than the text descriptions. Oblivion was far better - of course Oblivion has voice overs for ALL the text which made it more immersive to start with. This lack of gloss in some areas, and the opposite in others is a sad thing - finished this game would have hit a few 90%+ marks in reviews, rather than the telling 80%+ it seems to be getting.

Overall, it's flawed and buggy, but that's easily forgiven once you jump into unique action. My score? I'd go with an 85% and hope it prods future FPS's to new levels of originality.


Posted by Pádraic Brady in PC Gaming at 00:33

Thursday, April 5, 2007

Alien Assault Traders 0.30 Released!

Congratulations to Mark Dickenson (aka Panama Jack) and Rick Thomson (aka Tarnus) on their recent release of [Alien Assault Traders 0.30](#). It's been in development for a while now and it's cool to see the end product finally let loose 

Alien Assault Traders is an online browser-based space strategy trading game written in PHP. It's a fork of Blacknova Traders. If you're looking for a free open source web game then here's one to take a peek at. PJ is well known for focusing more than the usual effort on optimisation (he wrote the ADOdb Lite and Template Lite libraries) so it's a bit easier on your hardware than many alternatives.


I haven't been tracking development as much as I used to (busy with Astrum Futura, QGL and my Zend Framework proposals) but the new version has a significant overhaul of the previous 0.2x code. Have fun! 

Posted by Pádraic Brady in PHP Game Development at 00:33

One insecure PHP app too many?

Over the years I've been developing in PHP I've seen the number and impact of security exploits and issues on the web increase and even flourish. It's everyone's favourite game (I'm hardly innocent) to assign blame and PHP has proven an attractive target due to its high profile presence on the web, its large body of inexperienced or just ill-educated programmers, and of course its infamous ease of misuse.

It's always been interesting seeing how PHP and the broader community responds to these challenges (or not as the case may sometimes be) to quell the criticism and offer solutions. Some have backfired in spectacular style (magic quotes!) while others have introduced genuine improvement (no newlines in header!). Unfortunately the nature of PHP as a programming language is that it's easy to foul up. And this has inevitably left the responsibility of security completely up to the individual programmer. The results have been less than comforting, leaving an internet populated by God know how many insecure PHP scripts and applications written by well meaning but woefully under educated programmers and casual users.

At the end of the day, programmers require education, experience and guidance before they finally hit the jackpot and learn about revolutionary concepts like input filtering, output escaping, `mysql_real_escape_string()`, Chris Shiflett's website 

. But the level of education doesn't end with the revolution - beyond the horizon there awaits an entire world of odd animals like Code Injection, Path Disclosure, and my favourite - Unicode. Who'd have thought Unicode could be so cute, and such a problem for those reared on ASCII?

In any case, blaming stuff that goes wrong on PHP is the easy way out - PHP has historically offered honest and complete access to raw user input and doesn't accept responsibility for reporting when you practice poor security. If you want to be a professional programmer, those problems are yours to solve, young Padawan. PHP has done its bit over time giving us the marvel of the filter extension, and adding all sorts of interesting INI settings to play with should a programmer bother to read the INI once in a while and contemplate unholy actions like changing anything after the "=" character. If your shared host's name occupies a sentence with certain **f**** or **s**** words (the **a***** word is generally rare though popular with "Father Ted" fans) you should of course do your homework and find a more flexible host where the INI file is editable (by some method) and file permissions are a bit less stupid.

This is old news of course. There are a few PHP Security books published you can easily order from Amazon, or if you still live in the stone age, the local bookstore. If adventurous, some of them can be purchased on the cheap as PDF files to feed to your printer or read from your PC. They all note the highlights, and go into detail where it matters and page count allows. Of course, we all know few PHP programmers actually read those books - many sit unread on the shelves of bookstores and sit huddled together in small groups staring at PHP programmers who never glance at them twice. Such is the problem of giving programmers sole authority over security - too many don't feel any urgency to educate themselves. The result is well publicised - PHP applications littering the web with enough insecure code to depress many a security professional.

It's perhaps symptomatic then to look at how the community reacts to the growing pressure to prioritise security and make it easier for programmers to implement measures against the common security exploit vectors. One example to take is the Zend Framework. Fast approaching the dubious "1.0" version which means the API will finally remain stable and actually does not stand for any particular "completion" status - as mysterious as that might be (it's kind of weird comparing Symfony's 1.0 state to the ZF's). In development for over a year, the Zend Framework has a mixed security bag on the basic task of accessing user input. On the one hand, there's a set of neat classes called `Zend_Validate` which replace the now defunct and static heavy `Zend_Filter_Input` - on the other there's no overall system for pulling these into a neat validator chain and forcing access through it to the relevant superglobals. In other words, it doesn't add much beyond what direct procedural use of `intval()`, `ctype`, or those nifty PCRE functions offer. They might even be

faster...and require less typing. Now the argument has inevitably erupted before on the mailing list, and a nebulous something will happen after the 1.0 release - proposals, reviews, all sorts of inevitably good advances - but the fact it's still not present after over a year? Where was the urgency to add such a basic component to the framework? Already, I've seen three examples of programmers retrofitting their home brewed solutions into the framework as a consequence - programmers who care have no intention of using the current system when they're already spoiled by frameworks like Symfony and it's nifty YAML-based solution.

A second development worth a little verbose exploration is PHP's core filter extension. Unfortunately I'm hitting the 1000 word count limit I need to set to reduce the risk of writing too much (okay, so we passed that limit a while back - shoot me!). In summary, the filter's reception was less than lukewarm. It's odd to see something so apparently useful given so little attention. It's so bad in fact, that there's a rollercoaster ride happening to invent solutions to allow applications pretend it doesn't even exist. Honestly - the antics of PHP programmers get a little weirder with each passing year... I'll be writing something on the filter extension at some point - it's an under explored potential solution.

In conclusion (having broken that damn 1000 word limit), I'll burn the edifice above and simply beg programmers who know they are behind the curve on security education in PHP to catch up. Don't put it off and consider it a PITA since if you intend to carve a profitable slice of work from the PHP market you better know your stuff - and not just bits and pieces. On your next project stick a big label at the top of the milestone list which says "Implement basic security" - it's often surprising at how focusing on security up front and as part of the overall application design can encourage simpler, and very importantly, consistent solutions. Leaving it to the end, or as a low priority will predictably encourage inconsistency which makes mistakes easier to make.

1,130 words... I think I have a charity box I owe €1.30 to for that 

Posted by Pádraic Brady in PHP Security at 06:03