


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

Have you tried XSLT for templates?

Propel is also a great ORM tool that helps in complex Views with paging/filtering/sorting etc.

Here's how we do complex views:

<http://www.xml.it/Blog/2007/06/05/DIY+tips%3A+Propel+does+a+great+job+for+Views> Anonymous on Jun 8 2007, 08:59

I'm still confused, but on a higher level. 

1. Layouts seem like MVC exceptions to me. I don't mean the programmer's kind of exceptions, I mean the literal sense. I understand that they can save us from redundant includes. But why should we include header and footer when we can have a generic top view that includes a nested main view? The controller could tell the top view which one it should include. Then we wouldn't need those "exceptions" anymore.

2. Those placeholders look appealing. But shouldn't it be possible to append (instead of overwrite) data to them? Maybe line-wise? I'm thinking about JS includes, where every view component needs its own script. Therefore every component should be able to add another include line to the placeholder value.

3. I've been asking this to myself for a while now: You're proposing nested views, which is a good idea. And you're saying, that views should be able to read from the model directly or over a view helper. That sounds great, but I'm thinking in components. So what if a nested component (i.e. a nested view) needs to change something to the model? It doesn't have controller anymore and no state changing code should be put in that component's view.

Here's an example: I've got a login component (box) somewhere on most of my pages, but not all of them. According to your composite view pattern, it's being displayed by a nested view and pulls it's data (if and who's logged in at the moment) directly from the model, somehow. But what if a user logs in or out? That's a change to the model and should be handled by the controller. But I can't put that login checking code in every page's controller. That would be unDRY and, after all, not every page has a login box.. maybe it's displayed dynamically. So where to put that code? Can i create a new controller and attach it to a nested view somehow? This applies to other components as well (comment component, rating component, etc.).

My guts say I missed something simple here..



Anonymous on Jun 8 2007, 17:15

Confusion is cool



1. I've taken a stab at defining the process, but the implementation can vary. Many folk might prefer letting the Response object operate as your "top view". Others might prefer a Two-Step View style implementation using dispatch plugins. Here I've tried to keep Zend_View isolated. As far as a user is concerned these implementations work without Controllers even present (apart from Dispatches obviously). By removing the Controller as a dependency I'm forced to keep everything completely decoupled.

2. Placeholders, like some other of these, can look incredibly obvious



. It's actually little more than a proxy to a specific named Registry. I'd like to add it as a View Helper but not sure if the resulting interface is better for it (everything needs the \$this->placeholder()-> prefix with ZF View Helpers...). Yep, data should be appendable - main methods could be has(), get(), set() or add(). set() overwrites always, but add() simply appends the value either to a string, or a stack. Maybe append() is more obvious as a method name?

3. From what I understand you mean checking per page (dynamically) whether or not to display the login form or a default greeting/status type message?

The way I do it is simple enough. The login form is in it's own template since it's a variable element across pages (and not always displayed). When it is displayed it calls to a small View Helper. The Helper then perform a Model read to check whether the user is logged in or not. Model here takes the broad definition - in my ZF apps I use a simple call to get a boolean result using:


```
return Zend_Auth::getInstance()-&gt;hasIdentity();
```

A simple one liner. Not sure if this is what you meant though? Either way, ALL Model changes may only take place from a Controller. A login/logout is never handled by the View - all it can do is read that status and determine whether or not to display elements with a dependency on authentication. If the entire page is authentication sensitive then the Controller (higher up) needs to do something (like redirecting).


Make any sense here? Anonymous on Jun 8 2007, 21:36

Very identical to symfonys way of views. Anonymous on Jun 8 2007, 23:28

That doesn't really surprise me.

I'm sure frameworks have different names for these concepts, but the concepts themselves have been in use since the dawn of time 

In case there's any confusion I'm not inventing this from scratch - I've used a lot of frameworks over the years across PHP, Python (Django) and even Java. The concepts are obvious to anyone with similar experience.

I'm also sure I tried using Symfony as a club when discussing Zend_View on the ZF mailing lists a few times 

. Anonymous on Jun 8


2007, 23:59

I believe much of the terms you mentioned above could also be done in 'template level (tpl engine)' and not in 'the framework level (views)'. Whether this is the right way I don't know but this works for me.

I am using smarty to achieve more or less the things you mentioned above.

1) includes: just using the smarty equivalent `{include file='filename.tpl'}`

2) partials: using `{include file='filename.tpl' data=$data}` in foreach statements

3) dispatches: I have no idea on how to implement this, because there is some discussion on how to do it with as less overhead and loss in performance (for example the widget function that was posted on the wiki). I strongly believe (unless you can convince me otherwise 

) that this is an absolute must to use. I believe that retrieving data should be centralized. For example if you want to retrieve a list of books you have a bookcontroller with an action called list. When you have a widget on your homepage that displays the books you should be able to use the data provided by that action, rather than writing a new one that actually does the same. The output (view) of both can be different but the data is the same. I find it an interesting discussion and would like to know more on how you guys think about this.

4) layouts: I am using the smarty capture blocks to achieve this.

Example:

page.tpl file:

```
{capture name="content" assign="content"}
this is my content
{/capture}
```

```
{include file="base.tpl" content=$content}
```

base.tpl file:

```
{include file="header.tpl"}
```

```
{include file="sometemplate.tpl"}
```

```
{ $content }
```

```
{include file="footer.tpl"}
```

5) placeholders: I am using the same approach as mentioned above. you can define more capture blocks that insert things in a specific place (placeholder).

Example:

page.tpl file:

```
{capture name='content' assign='content'}
this is my content
{/capture}
```

```
{capture name='head' assign='head'}
```

```
{/capture}
```

```
{include file='base.tpl' content=$content
head=$head}
```

base.tpl file:

```
{$head}
```


```
{include file='header.tpl'}
```

```
{include file='sometemplate.tpl'}
```


```
{$content}
```

```
{include file='footer.tpl'}
```


I don't know if this is the best approach when looking at performance but this works for me. I would like to have some comments on my way of working and compare it to the article above. Anonymous on Jun 9 2007, 16:38


You have to remember that Zend_View is by itself a template engine. While Smarty has tons of functionality to leverage off, Zend_View doesn't and only has a small collection of View Helpers. Without something mature like Smarty, Zend_View lacks those features a lot of us take for granted in Smarty. 

Which is one of the reasons why I think Zend_View needs it's own set of implementations which work with the stock class, and which don't interfere with backwards compatibility (and certainly not with any Smarty derived subclasses!).

Even something like Partials can be found everywhere these days: Smarty, Solar Framework, Symfony, etc. - but not Zend_View 

Anonymous on Jun 10 2007, 01:03

I am in a hurry to read the 7th part 

. Thank you very much for your article (very good job), even if I am not very at ease with english 

Anonymous on Jun 20 2007, 23:24

Great article again. looking forward to the next one. Anonymous on Jun 28 2007, 23:29

```
class Zend_View_Helper_Partial
{
/**
```

zend view instance

@var Zend_View

```
/
public $view;

public function setView(Zend_View_Interface $view){
    $this->view = $view;
}

public function Partial($sTemplate, $variables = array())
{
    // copy old view
    $view = clone $this->view;
    // unset old variables to remove name conflicts
    $view->clearVars();
    foreach ($variables as $id => $val){
        $view->$id = $val;
    }
    return $view->render($sTemplate);
}
}
```

partial helper Anonymous on Sep 7 2007, 00:40

Response Segments

It is quite easy to make a view helper that allows to you declare certain parts of the output in a named segment. We can then forward to another controller that would populate other named segments like the common 'header', 'menu', 'footer'

And at the end, as you said construct a page from the named segments combined with a common template.

For a simple application, we can complete avoid the more complex proposals of Zend_Layout. This simple approach looks very friendly with Template_Lite too.

This is what i'm doing:

```
class View_Help_Blocks {
    $_currentSegment

    function begin($name) {
        $_currentSegment = $name;
        ob_start();
    }

    function end() {
        getResponse->appendBody($_currentSegment, ob_get_clean());
    }
}
```

Perhaps you can something like this in your Zend_View_Enhanced proposal

Regards, Anonymous on Oct 18 2007, 17:30