


## Complex Views with the Zend Framework - Part 5: The Two-Step View Pattern

It's been a while since I continued this series. Unfortunately real life workloads are unforgiving of the best of intentions 

Part 5 of our series takes a small time-out from approaching a Composite View solution to reusable Views to take a peek at a simpler approach useful for simpler types of web applications. As we've discussed previously Composite Views allow the nesting of reusable View elements, effectively building a View based on a hierarchy of Views. But often there are simpler solutions to simpler problems. One such solution is the Two-Step View pattern, sometimes called Layouts if implemented in a specific way (as we do below!).

Imagine a simple website. You have hundreds of pages of unique content, but the header and footer for each page is identical. Applying the Composite View approach has the same problem as applying a standard PHP include or `Zend_View::render()` call to including these common elements - the calls themselves are scattered across each and every template. This is the hallmark of a Layout - duplicated includes/renders across multiple views:

```
<?php echo $this->render('standard_header.phtml'); ?>
<p>Here is our unique content! But look, all unique templates
now have the same "standard" headers and footers defined by a
render() call. How do we remove these completely and apply them
automatically instead?</p>
<?php echo $this->render('standard_footer.phtml'); ?>
```

The key is to take the duplicated calls and other duplicated markup and stick them in a Layout template which will encapsulate all Views automatically. Then the only thing our templates contain is unique content!

```
<p>Here is our unique content template! But where have the
standard header and footer includes vanished to?</p>
```

All the Layout file needs to do is provide a "hook", a method placement which signifies where the main View output (which is generated by the current dispatch cycle) should be placed. Since we define two parts the View here we'll refer to them as the "Layout" and the "Main". A Layout might look like:

```
<?php echo $this->render('standard_header.phtml'); ?>
<?php echo $this->main(); ?>
<?php echo $this->render('standard_footer.phtml'); ?>
```

The new `Zps_View::main()` method (`Zps_View` is a subclass of `Zend_View` to which we can add customised behaviour) simply tells the View to render its output at this location in the template. This assumes the default `Zend_View::render()` method now takes a two step approach to rendering (this is where the Two-Step View Pattern comes into play).

1. Render a Layout if one is defined
2. Render the Main template into the Layout

The only funky logic is that the presence of a Layout forces our View object's render method to take a detour so that Layouts are rendered first. This is a pretty simple change to `Zend_View`. Here's our `Zps_View` class with the revised logic.

Please forgive the lack of proper phpDoc comments - Serendipity won't play nice with them.

```

// Zend_View <strong>/
require_once 'Zend/View.php';
// Zps_View_Interface </strong>/
require_once 'Zps/View/Interface.php';
class Zps_View extends Zend_View implements Zps_View_Interface
{
    //
    // The Main Template (i.e. the template file a Controller wishes
    // to render).
    // _mainFile cannot be set by a public setter so it doubles as
    // as a safety valve to prevent unwarranted use of main().
    //
    // @var string
    //
    protected $_mainFile = null;
    //
    // The Layout Template
    //
    // @var string
    //
    protected $_layoutFile = null;
    //
    // Overrides Zend_View::render() to introduce a two step view approach when
    // a Layout template has been defined. The two steps are handled using
    // separate calls to parent::render() which calls the Zend_View render()
    // method without overriding.
    //
    // @param string $name The script script name to process.
    // @return string The script output.
    //
    public function render($name)
    {
        if ($this->hasLayout() && !isset($this->_mainFile)) {
            $this->_mainFile = $name;
            return parent::render( $this->getLayout() );
        }
        return parent::render($name);
    }
    //
    // Set the filename of a Layout template to be used. The existence of a
    // Layout filename will force the over-ridden render() method to detour
    // and render the Layout, only rendering the Main template when a main()
    // call is issued in the Layout template.
    //
    // @param $file string
    // @return void
    //
    public function setLayout($file = 'layout.phtml')
    {
        $this->_layoutFile = $file;
    }
    //
    // Return the filename of the Layout. Layouts are like any

```

```

// other template script and are located in the same place in
// application filesystem.
//
// @return string
//
public function getLayout()
{
    return $this->_layoutFile;
}
//
// Returns true if a Layout has been set for this View.
//
// @return bool
///
public function hasLayout()
{
    return isset($this->_layoutFile);
}
//
// Inform the View object that it should render the Main View, i.e.
// render the template handed to the render() method by a Controller.
// This method is only useful if a Layout is being used, otherwise
// expect an Exception.
//
// @return string
// @throws Zps_View_Exception
//
public function main()
{
    if (isset($this->_mainFile)) {
        return parent::render($this->_mainFile);
    }
    require_once 'Zps/View/Exception.php';
    throw new Zps_View_Exception('Invalid call: There is no primary View template to render');
}
//
// Method to clone this View assuming the sub-View (the clone) is from
// the same application Module as the original.
// Here we are simply getting rid of the inherited public variables which
// represent the ancestor View's model.
// We also disable any Layouts (the inheritance would lead to infinite
// looping otherwise - Apache would bark and die on the spot!)
//
// @return null
//
public function __clone()
{
    foreach(get_object_vars($this) as $key=>$value) {
        $this->__unset($key);
    }
    $this->setLayout(null);
}
}
}

```


So there we go, functional code for allowing Layouts in a Two-Step View approach. Notice how the render() and main() methods interact. Because it's both have very specific uses, main() is only useable within templates when injecting the main template into a Layout.

Sample usage is pretty simple - I won't delve into any details since you just need two additional pieces of work when instantiating a View object:

1. Create a Layout template (you'll notice a setLayout() default is "layout.phtml" but you're not bound to that convention by any means.
2. Set the Layout on a View object, e.g.

```
$view = new Zps_View;  
$view->setBasePath('/path/to/default/view/directory');  
$view->setLayout('layout.phtml'); // this will force render() to perform a Two-Step  
$view->render('sometemplate.phtml');
```


If you followed my previous posts you'll be able to integrate the Two-Step View/Layout approach pretty easily into your Views. Of course, as usual, a key observation is that in using a Zend\_View subclass be sure not to rely on Zend\_Controller\_Action::initView(). You'll need to override that method in your own application specific Zend\_Controller\_Action subclass.

If the lack of a Zps\_View\_Interface worries you it's just a declaration of all the public methods above. I won't post it here, this entry is long enough 

Any final words? A similar system is also possible using an alternate implementation. Matthew Weier O'Phinney, when I originally started asking about complex views in the Zend Framework, posted a Two-Step View implementation using a dispatchLoopShutdown() plugin. You can read more about this over at the mailing list archives - here's the exact link to Matthew's email.

<http://framework.zend.com/wiki/display/ZFMLGEN/mail/27145>

Finally, this is completely compatible with using a Composite View system. You can imagine creating a component full of widgets or plugin output. Each of these would be aggregated into a Composite View. But that doesn't mean they don't all share one thing - a common layout. So Layouts (Two-Step View) and Composite Views play quite nicely together.

Have fun! 


If anyone comes up more bright ideas throw them into a comment for the hordes of blog readers to consume!


Posted by Pádraic Brady in PHP Game Development, PHP General, PHP Security, Zend Framework at 02:35

Really useful entry as always!! I've been following your work on the Dev Net Store, as well as reading here. It's been really helpful looking through sites like this while i'm learning my way through the Zend framework.


Anyway keep up the good work!

Mike Anonymous on May 24 2007, 00:43

@Mike - Thanks 


@Everyone - Added a small bug fix to eliminate Layout nesting. I have made a mental note to Unit Test snippets I extract from my existing code before posting them online 

. Works fine now.

P.S. Those who are following DevNetStore like Mike can grab a fuller copy from our in-progress XP User Story Manager at <http://svn.astrumfutura.org/userstory> where this is actually seeing real world use - albeit with a sucky CSS them that needs an artistic touch 

. Anonymous on May 24 2007, 05:22

Hi

Your series of articles is great. Much appreciated. 

I have used the following for these "generic" type pages, and would be interested in your view of such a (simple) approach.

```
[geshi lang=PHP]
// no sub-class
$view = new Zend_View;
$view->setBasePath('/path/to/default/view/directory');

// layout name (or something from a controller / action)
$view->layout = "sometemplate";

$view->render("generic.phtml");
[/lang]
```

in the "generic.phtml" file

```
[geshi lang=PHP]
$this->render("header.phtml");
$this->render($this->layout);
$this->render("footer.phtml");
[/lang]
```

It seems that Named Segments (zf-ref 7.9.3) is another method. Any thoughts?

Keep up the good work.

Cheers

Nick Anonymous on May 25 2007, 08:14

i'm having problems trying to make the default script extension be "tpl.php". before, i used "viewSuffix" to do that, but it isn't working anymore. where should i change this default extension??

thanks!!

Helder Anonymous on May 25 2007, 21:03

One assumes you just call:

```
render('name.tpl.php');
```

Zend\_View doesn't care about the filename ending. Anonymous on May 25 2007, 22:08

nice article!, I've made the small improvement ;), maybe it could be useful to set many templates and render them in layout, consider `$_mainFile` property as a array. `render()` can set index 'default' in `$_mainFile` and also exists methods like `setMain($file, $name)`, and finally method `main($name = 'default')` would render template with index `$name` in `$_mainFile` Anonymous on May 31 2007, 19:48

Great work, thanks a lot!

But I think than layout must be rendered **later** than main template. Why? For example, if we want to use **placeholders** (like in 6th chapter). For example, if we define placeholder in layout, and fill them in main template.

So I made some changes to change the order:

```
...
private $_content;
...
public function render($name)
{
if ($this->hasLayout() && !isset($this->_content)) {
$this->_content = parent::render($name);
return parent::render($this->getLayout());
}
return parent::render($name);
}
...
public function main()
{
return $this->_content;
} Anonymous on Jul 12 2007, 00:06
```

Yay, i read this article several times, and i cant figure it how i can implement this on my ZF site. I am having problem to display header and footer files, so google it to this article. Well, i dont know where this class should be. Where to save this class? in what directory? Anonymous on Feb 22 2008, 21:53