

Complex Views with the Zend Framework - Part 4: The View Factory

In parts [1](#), [2](#) and [3](#) I've been taking a look at the [Zend Framework](#) and putting together a broad picture of a potential implementation to add support for complex multi-part web pages. This refers to the practice of building a web page in an application from a number of common reusable elements. An example of such elements include header sections, footers, menu bars, widgets, etc, which surround the main content returned by any client request. In Parts [2](#) and [3](#), I introduced two useful design patterns for this purpose: Composite View and View Helper.

Here in Part 4, I revisit a characteristic of the Composite View pattern - the idea of using multiple Zend_View objects (or subclasses thereof) to encapsulate each element of the web page. Zend_View is rather a complex class to instantiate. Not only does it have optional settings, it also requires path information in order to locate it's templates, helpers and filters. To make matters worse, we would like for each concrete instance to have access to it's ultimate parent's Model as given to it by the Controllers which handle requests.

You can guess that there's a long block of code involved in creating a fully functional View object in the Zend Framework (check out Zend_Controller_Action::initView() for an idea). Most likely those using the Zend Framework have it all located in either a subclass of Zend_Controller_Action, or within the bootstrap index.php file, or rely on the default initView() method. This is fine for one single instance of Zend_View - but once we get to three, four or more View objects, as needed by the Composite View pattern, this won't work.

As you can guess, we would have to duplicate the same block of setup code wherever a new View object is created. Now since duplication is an obvious code smell, and since the logic concerns object instantiation and configuration, we may identify the problem and its solution - the Factory Pattern.

In this blog entry I'll present a class (the Factory) dedicated to churning out any number of Zps_View (subclass of Zend_View) objects using a class called Zps_View_Factory. To make things more interesting we'll also centralise all those default settings a View might require into a configuration file which our Factory will be able to use. But first let's look at the UML class diagram of a possible Factory setup.



As you can see the Zps_View_Factory encapsulates all the steps needed to create View objects. All someone on the outside need do is instantiate a Factory object, call it's createInstance() method with some optional parameters, and receive a configured View object ready for rendering templates. Before we jump into the Factory code though, let's put all our various preferred View settings into a config file for Zend_Config to chew on.

```
[general]
```

```
; Extracted from config.ini
```

```
; Standard View settings
view.encoding = "UTF-8"
view.escape = htmlentities
view.strictvars = 1
```

With our configuration file we improve the centralisation by putting all our settings information in one editable file. I've neglected to add path information - instead we'll rely on a convention that the View's base directory is of the form: applicationPath/moduleName/views.

Now for the Factory class. Before we go there, I've first added a Zps_View class. This extends Zend_View to add a pair of methods for accessing/setting the "main" View. In essence, the View our Controller layer creates. We use this as a common parameter for all Views so that composite Views can access the Controller generated Model (as granted to the "Main View") for any request-specific data a View may require.

As a hint to the future, Zps_View has a protected _factory() method which attempts to locate a valid instance of the View Factory from the local Zend_Registry instance.

```
require_once 'Zend/View.php';
require_once 'Zps/View/Interface.php';
class Zps_View extends Zend_View implements Zps_View_Interface
{
    // Set a main parent View sourced from each Request's controller.
    public function setMainView(Zend_View_Interface $view)
    {
        $this->_mainView = $view;
        return $this;
    }
    // Get the main parent View.
    public function getMainView()
    {
        if (!isset($this->_mainView)) {
            return $this;
        }
        return $this->_mainView;
    }
    // Method to clone this View assuming the sub-View (the clone) is from
    // the same application Module as the original, and therefore is likely
    // a simply duplicate of its parent.
    // Here we are simply getting rid of the inherited public variables which
    // represent the ancestor View's model (isolates Views to prevent coupling).
    public function __clone()
    {
        foreach(get_object_vars($this) as $key=>$value) {
            $this->__unset($key);
        }
    }
    // Fetch an instance of Zps_View_Factory which should be present in
    // in the default instance of Zend_Registry.
    protected function _factory()
    {
        if (!Zend_Registry::isRegistered('ViewFactory')) {
            throw new Zps_View_Exception('Registry does not contain an entry for the ViewFactory');
        } else {
            $viewFactory = Zend_Registry::get('ViewFactory');
        }
        return $viewFactory;
    }
}
```

The Zps_View_Interface interface merely enforces the revised interface on the Zend_View subclass - in our case the new setMainView/getMainView mutator and accessor methods.

Our Factory class isn't all that complex in the end. The configuration file, and some bootstrap sourced Registry values are the main inputs. The rest is simple enough except for the automated handling of Main View references.

```
require_once 'Zps/View/Factory/Interface.php';
require_once 'Zps/View.php';
class Zps_View_Factory implements Zps_View_Factory_Interface
{
    protected $_config = null;
    public function __construct(Zend_Config $config)
    {
        $this->_config = $config;
    }
    public function createInstance($module = null, Zps_View $view = null, array $params = null)
    {
        if (isset($module) && !ctype_alnum($module)) {
            require_once 'Zps/View/Exception.php';
            throw new Zps_View_Exception('Invalid module name; must only contain alphanumeric
characters');
        }
        if (isset($view) && is_null($module)) {
            $subView = clone $view;
        } else {
            if (is_null($module)) {
                $module = 'default'; // assume the default module
            }
            $subView = new Zps_View();
            // This is the conventional ZF directory layout. Some configuration
            // improvements could allow more flexibility here.
            $subView->setBasePath(
                Zend_Registry::get('BasePath') . DIRECTORY_SEPARATOR . $module .
                DIRECTORY_SEPARATOR . 'views'
            );
            if(isset($view)) {
                $subView->setMainView($view->getMainView());
            }
            $subView->setEncoding($this->_config->encoding);
            $subView->setEscape($this->_config->escape);
        }
        // Place holder for parameters set by a parent view on a sub-view which
        // can be used as inputs to certain View Helpers in the sub view. I'll
        // explain later .
        if (isset($params)) {
            $subView->params = $params;
        }
        return $subView;
    }
}
```

Here's the Factory being instantiated from an extract of the bootstrap file (bootstrap.php or index.php depending on setup) and being added to the Registry. You could instantiate it from anywhere really but this is the lowest central point independent of any Controllers which has direct access to the configuration file.

```
$ApplicationDir = dirname(<u>__FILE__</u>);
// Include path setup (assuming no facility to edit php.ini)
set_include_path(
    $ApplicationDir . PATH_SEPARATOR
    . $ApplicationDir . '/extends' . PATH_SEPARATOR // holds ZF subclasses
    . get_include_path()
);
// Load configuration
require_once 'Zend/Config/Ini.php';
$Config = new Zend_Config_Ini($ApplicationDir . '/config/config.ini', 'general');
$ConfigView = $Config->view;
// Create the View Factory
require_once 'Zps/View/Factory.php';
$ViewFactory = new Zps_View_Factory($ConfigView);
// Create the Registry
require_once 'Zend/Registry.php';
$Registry = Zend_Registry::getInstance();
$Registry->set('Configuration', $Config);
$Registry->set('BasePath', $ApplicationDir);
$Registry->set('ViewFactory', $ViewFactory);
```

Last of all, here's an extract from a possible subclass of Zend_Controller_Action. Since we have subclassed Zend_View the default initView() method can't be used because it is tightly coupled to the class name "Zend_View" (so it won't work with a subclass called "Zps_View"). With our Factory in place however, it's a few short lines.

```
require_once 'Zend/Controller/Action.php';
class Zps_Controller_Action extends Zend_Controller_Action
{
    // Instance of Zend_Registry passed by Zend_Controller_Front as an
    // invocation argument.
    protected $registry = null;
    // init() method called by Zend_Controller_Action constructor to setup
    // this Action (or its subclasses). Used here to assign FC params as
    // class properties and perform general preparation.
    public function init()
    {
        $this->_initRegistry();
        $this->initView();
        $this->_initResponseHeaders();
    }
    protected function _initRegistry()
    {
        if (!$this->getInvokeArg('Registry') instanceof Zend_Registry) {
            require_once 'Zps/Controller/Exception.php';
            throw new Zps_Controller_Exception('Registry invocation argument is not an instance of type Zend_Registry');
        }
    }
}
```


```

    $this->registry = $this->getInvokeArg('Registry');
}
// Over-rides Zend_Controller_Action::initView()
public function initView()
{
    $viewFactory = $this->registry->get('ViewFactory');
    $this->view = $viewFactory->createInstance(
        $this->getRequest()->getModuleName()
    );
    return $this->view;
}
// Configure the default Response header "Content-Type" value.
protected function _initResponseHeaders()
{
    $characterSet = $this->registry->get('Configuration')->view->encoding;
    $this->getResponse()->setHeader('Content-Type', 'text/html; charset=' . $characterSet);
}
}

```

Last words? Using the Composite View pattern we need a way of managing the creation of a multitude of View objects. Rather than relying on the typical coupling `Zend_Controller_Action::initView()` offers, it's better to centralise the code which manages object instantiation in a dedicated Factory class. This will centralise the code, more easily allow for subclassing/editing, and prevent code duplication.

Creating the Factory within the bootstrap just maintains a single instance of the Factory in the Registry for reuse. Although it could be a static method or Singleton within `Zps_View` itself, both of these options might add some inconvenient coupling to a specific class name (much the same problem that the default `Zend_Controller_Action::initView()` method has since it explicitly refers to "Zend_View" and is therefore in need of being overridden in any `Zend_View` subclass if you intend relying on it).

Let's try for an overall Composite View system in Part 5! 

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 15:08

I have a (perhaps naive) question:

With seemingly complex multi-element web pages being SO common, these days, why (at this late date) are the "best practices" for implementing such things not 'cast in concrete', by now? i.e. why is the implementation architecture still open to such debate?

Perhaps my main reason for investing the time to explore the Zend Framework was an expectation that the 'best & brightest' at Zend would likely implement things, according to 'best practices' (an assumption that may or may not be justified, regarding other framework authors). The substantial MVC approach changes, I've observed, through the various ZF beta versions have cast that supposition into considerable doubt.

At this point, I'm left wondering if building multi-element pages has really matured to the point of credible, defensible 'best practices' ? ... in any language/framework?


Nevertheless, I applaud and stand in awe of those experimenting to arrive at defensible MVC 'best practices' ... and hope that the results are consistent with original ZF **simplicity** design goals. Thanks - to you ... Anonymous on May 3 2007, 19:56

The root of the issue is the 80/20 requirement the framework applies. It's intention as of v1.0 is to meet a specific set of requirements as laid out in the roadmap. You'll notice past v1.0 that further layers of functionality will eventually get added to push the envelope

further.

This isn't a particular flaw as such, getting past a certain level of functionality requires additional time and resources - and the ZF doesn't have so much of either that it can get everything done on time to the satisfaction of everyone (esp. when you start adding requirements so late).

As with any open source project, those who come late to the party can't expect to get everything they want done immediately. They missed the boat on making comments, adding proposals early, and arguing their positions. I'm confident the ZF will get nested views integrated in time - we'll just have to wait until the v1.0 is made and the Zender's have the time to review proposals for this.

I suppose the last point is valid another way - you really only see so much attention and experimentation when there a relatively stable core to experiment with 

. Anonymous on May 3 2007, 22:29


No doubt nested views will pass to the 80s part from the 20 ones in a near future. As Mike says, it's being pretty common these days. Just a question : I guess you already have tested the overall Composite View system, I'm wondering what is the impact on the execution speed of this system and how deep have you been in the views nesting ?

However, I can hear two people at least aplauding by now 

Anonymous on May 4 2007, 13:01

I haven't tested performance. While the design is reasonable, it likely does take a small nibble at the performance apple. In technical terms, using many View objects vs one View object should be minimal. Once the series has outlasted its welcome I'll run a round of performance benchmarks (which will be labelled useless as soon as posted no doubt) and publish them.

I'm sure it can be done faster - but such an optimisation always has some negative impact on the design values. Anonymous on May 4 2007, 14:43

Interesting article. Looking forward to the next part, in which this all will be made clear to me 


I'm really curious to see how this will end up from the perspective of the view layer. From that point of view, putting together a page with different parts shouldn't be more difficult then including a few other modules? Anonymous on May 6 2007, 08:55

Good work, thank you for a good article!

When can we read Part 5 of the series? =)

Only one thing is not clear for me yet - how nested view objects can get data from models? When we use attach() method, newly created view have no assigned vars.

Can you give me a quick hint, how to get it all to work?

Thank you once again! 

Anonymous on May 6 2007, 09:23

Hi Sergey, check out Part 2 which introduced a more liberal explanation of the View Helper pattern. In summary, when a View requires Model data it has not been given by a) the "Main" View, or b) the Controller, it may read that data directly from the Model itself using a read only API bundled in a View Helper. Anonymous on May 6 2007, 12:42

Thank you for the answer. Anonymous on May 6 2007, 13:27

Hi PÃ¡draic,

More I'm playing with your concept and more I'm getting confused by the data retrieving.

Here is what I understood :

We have a controller which instantiates a view (the main one),

The main view, through the sub views must get the rights templates and the data needed to render them.

Each sub view can access view helpers supplied by the main view in order to access the model.

Now, let's take your example of the part 3 :

In the articlelist.phtml template, you "attach" overbody.phtml and underbody.phtml.

Let's imagine now, these two subviews need partially the same data supplied the myObject->load() method for example. The subview "overbody" calls a view helper which access the model calling the myObject->load()method.

My questions are :

Where do you push the result so that The subview "underbody" would not need to call twice the same view helper ? In th main view ?

Is the main view acts like a data tank where subviews would check for the data model supplied previously both by the controller or other subviews ?

If I'm in a saveAction for example and after having saved data successfully, shall I push the data in the main view so that they would be available for the subviews instead of using a view helper ?

Can you summarize me in two lines, the irish history from the begining until this day which is, I guess, a special one ?



best regards,

fred Anonymous on May 8 2007, 15:38

Hum...forget these questions, sometimes just putting the problem on the paper, reveals the solution.

The problem was I didn't digg deeper enough in the view implementation in the zf as I used it just to render smarty templates. It doesn't mean I'm not looking forward to reading the part 5 of this exciting serie -).

Fred Anonymous on May 9 2007, 11:35

It's not a bad questions. Some of it is solved easily using the Main View as a data collector. The other part - having two Views access the same View Helper is more difficult. Each will try to instantiate a new object, so one method is to implement them as Singletons (i.e. no matter how many Views are in use they always access the exact same instance of a Helper).

Not sure how to confine this behaviour within Zend_View's approach to helper methods. It might just be easier to ignore the Zend_View's integration in helpers and use the classes directly. On a plus side - if the helper class is very small then having multiple instances might not cost much (if anything) in performance. Anonymous on May 9 2007, 14:05