

Friday, January 11. 2008

The PHPSpec Zend Framework App Testing Manifesto: Preamble

(...or some title designed to attract curious readers so I can captivate them into thinking this will not be a long torturous blog entry) Over the course of the next obsessional phase I go through I'll be attempting to pound the Zend Framework into submission so I can apply Behaviour-Driven Development (BDD) using PHPSpec when I write a Controller. Why? Because I feel like it, and it gives me an excuse to promote one possible incarnation of PHPMock and the PHPSpec Zend Framework extension. See? Perfectly reasonable selfishness!

The other reason is that whether you apply TDD, or the funky new presence of BDD in PHP, applying some form of predictive specification before you rush into PHPEclipse to start typing PHP is a good thing. The alternative is long nights debugging why Controller X and Model Y refuse to produce View Z. The immediate problem with this is that the Zend Framework currently does not offer some means of manipulating Framework operations with a simplified API, instead you have to muck about with `Zend_Controller_Front` and all sorts of other long-winded-named classes whose operation is often mysterious if you've managed to evade tinkering with them while building the Universe's next greatest social networking site.

Never fear, there are established solutions. One of the easiest to find is archived on the fw-general mailing list for the framework, a relatively straightforward post by the Master Of MVC Operations, Matthew Weier O'Phinney, using PHPUnit and an extended `PHPUnit_TestCase` class. Here's the URL in case you missed it:
How to use PHPUnit in a MVC project

For PHPSpec I hope to simplify even further. Moving from `Zend_Controller_*::someConfusingMethod()` x 5 to `PHPSpec_Context::doItAlreadyOrI'llKillSomething()`. The basis of this theoretical (unless you've found the top secret location of the subversion repo) discussion is that simple is better, and easier is way better still. To get there one needs to compact Zend Framework operations into a handful of easy to remember methods and practices. Unfortunately it does also require some Zend Framework fun by establishing a few ground rules on how you write Controller code. We'll get there much later, because it's one detail not necessary to cover in an opening sortie and it's not incredibly troublesome (I think).

To refresh your memory, PHPSpec as a BDD framework is designed to kick your ass into high-TDD gear without needing 2+ years of prior experience to get there. It's also designed to aid your thought approach by using a readable API. Besides being easy to learn, it also enforces several well-established best practices in TDD and xUnit Patterns. Here's a quickie shot:

```
class DescribeZendFramework extends PHPSpec_Context { public function itShouldMakeWebDevelopmentBetter() {
    $zend = new Zend_Framework;    $this->spec($zend->isBetter())->should->beTrue(); }}
Stored as ZendFrameworkSpec.php, you could run this spec by hitting it's location on the command line (there is a
HTML Runner option) and running:
phpspec ZendFrameworkSpecNot rocket science. Back to the topic of applying BDD... If one thinks a bit, any ZF request
is composed of only a few key components. There's the Request (POST/GET/COOKIE/URI Params), the Controller, the
Action on the Controller, and finally the Response. In between are bits and pieces which while important in a complex
application we needn't dwell on yet.
```

In a perfect world, we could wave a magic wand and drop big hints like:

```
class DescribeIndexController extends PHPSpec_Context_Zend{ public function
itShouldDisplayHelloWorldOnIndexAction() {    $this->get('index');    $this->response()->should->match("/Hello
World"); }}
The basic foundation above is simple enough. The Controller is derived from the class name, the Action is passed to the
relevant request method (get() for GET), there's a response upon which specifications can be defined.
```

In an imperfect world, it's not quite as simple as this. In applying TDD or BDD a big annoyance is ensuring we only test isolated specifics. In a typical Controller Action one can easily expect a few things:

1. Models
2. Misc Objects (e.g. `Zend_Mail`)

3. View Rendering

The biggest problem in there are Models. Models are incredibly irritating objects that insist on writing to databases. They are also as common as dirt in Controllers. Not exactly good company when applying TDD or BDD. The suggested best practice approach is to Mock or Stub these demons out of their evil ways, and turn them into predictable automatons (we can separately spec/test the actual Model classes independently). In other words, we need to Mock/Stub the dependencies of the class being tested/spec'd so it's isolated - should sound familiar as an intonation of Unit Testing practitioners worldwide.

That's the problem in a nutshell. Controllers tend to directly instantiate new objects without the benefit of good Dependency Injection. Makes them very difficult to test discretely without falling back on the double-edged sword of pure functional or acceptance testing. And that double-edged sword is quite popular in PHP right now.

Enter something like a Factory. Using a Factory, our specs/tests can potentially intercept objects used in a Controller and replace them with Mocked or Stubbed copies whose behaviour is controller by us. The simplest style would be to use a general object Factory to replace both `require_once` and `new` keywords (i.e. introduce a new Controller convention whereby we don't use the "new" keyword if avoidable in Action code). Something whose API is similar to:

```
$mailer = Zend_Factory::create('Zend_Mail');
```

A suitable `Zend_Factory` class would inherently know the `require` path for `Zend_Mail`, and would return a new `Zend_Mail` object (assume optional constructor params in the API are supported). Couple this with some backend Registry and one could allow `PHPSpec/PHPUnit` to play god with:

```
Zend_Factory::replaceClass('Zend_Mail', new Zend_Mail_Mocked);
```

Now any call to `Zend_Factory::create('Zend_Mail')` would actually return a registered Mock of `Zend_Mail`. Granted the API ignores multiple objects and non-`Zend` classes for now (an API potential discussion for another day if this goes beyond theory) but you get the point. Replacing "new Class" with "`Zend_Factory::create('Class')`" in a Controller Action would facilitate mocking by introducing a healthy dose of Dependency Injection. Let's kick another example using `PHPSpec` and `PHPUnit` (note: `PHPUnit` is undergoing development so the API is questionable as to its stability) where the `PHPSpec_Context_Zend` extension defines additional helper methods on top of the normal ones attached to `PHPSpec_Context`. This is supposed to be the revelatory part of the blog post so pay attention and critique it to death in the comments.

```
require_once 'Zend/Mail.php';class DescribeIndexController extends PHPSpec_Context_Zend{ public function before() { $this->mail = phpmock('Zend_Mail'); $this->replaceClass('Zend_Mail', $this->mail); } public function itShouldSendAnEmailUsingPostData() { // setup the Mocked Zend_Mail instance (no real email sent of course) $this->mail->shouldReceive('setBodyText')->with('This was a test email')->once(); $this->mail->shouldReceive('setFrom')->with('anon@example.net')->once(); $this->mail->shouldReceive('setTo')->with('padraic@example.com')->once(); $this->mail->shouldReceive('setSubject')->with('Testing...1...2...3')->once(); $this->mail->shouldReceive('send')->withNoArgs()->once(); // POST request to IndexController::mailAction() $this->post( 'mail', array( 'email'=>'padraic@example.com', 'subject'=>'Testing...1...2...3', 'body'=>'This was a test email' ) ); // and because no PHPUnit integration with PHPSpec yet...ugly append $this->spec($this->mail->verify())->should->beTrue(); // we're ignoring the Response for now..oh ok then $this->response()->should->beSuccess(); $this->response()->should->match("/successfully sent/"); }}
```

Screw "new `Zend_Mail`", let's mock the devil 'imself mo chara! What works for `Zend_Mail` would feasibly also work for Models. So in theory (at a personal perspective) I like my Manifesto Preamble. Having a simplified approach to applying BDD or TDD to `Zend Framework` controllers (someone else can take on `Symfony` if they wish) is something I'd very much like to have.

Now we have no implementation, just a BDD derived specification of what the future implementation should do once complete. To round off the example, here's a possible implementation:

```
class IndexController extends Zend_Controller_Action{ public function mailAction() { // ignore security for brevity; normally we'd be damned sure $post // was the result of filtered/validated data $post = $_POST; $mailer = Zend_Factory::create('Zend_Mail'); // presto; delivers PHPSpec defined Mock $mailer->setBodyText($post['body']); $mailer->setFrom('anon@example.net'); $mailer->setTo($post['email']); $mailer->setSubject($post['subject']); $mailer->send(); $this->getResponse()->setBody('Email successfully sent.');
```

Of course this is total drivel without a) a stable `PHPUnit`, and b) `PHPSpec_Context_Zend`. I'll take a quick

proof-of-concept stab later after I investigate the wonders of GIT to see if it kicks SVK up the rear... In the meantime is this drivel for real, or does it have potential? Commentors who use the phrase "Mad Irishman" will be booted...

More when the PHPSpec 0.3.0 branch is active.

Posted by Pádraic Brady in PHP General, PHP Security at 23:14

In terms of general acceptance, the biggest problem I see is the loss of autocomplete when you use a factory.

ie. once I do `$mailer = Zend_Factory::create('Zend_Mail');` I can no longer auto-complete on `$mailer` in the IDE. Of course, you "pros" can scoff, but every single dev who works for me relies on autocomplete to remember the whether its `setBodyText()` or `setTextBody()` (or whatever...)

Having said that, the most commonly used classes used in a controller are models that almost certainly either derive directly from `Zend_Db_Table` or aggregate to it. `Zend_Db_Table` has a useful static function called `setDefaultDbAdapter()` which if you could mock at that level, would be trivial to drop in.

I have a feeling that `Zend_Mail` works similar with its backend transports too. A quick look shows a `setDefaultTransport()` static function that almost certainly works the same way.

Regards,

Rob...

Anonymous on Jan 12 2008, 09:19

Ahhh, party pooper

I know code completion can be great, but I don't subscribe to it as a necessity all the time. Given a choice between autocompletion and better testing access, I'll side with better testing access and invest sometime downloading the Zend Framework manual.

Not sure what you IDE is, but for Eclipse PDT which I'm using it's quite a simple solution. First of all I need to make sure my PHP project's `include_path` has the path to the framework's location, say `"/path/to/zendframework/library"`. This ensures (as I'm sure you know already, but for the benefit of others) code completion for Zend Framework classes is available since PDT now knows what classes the code will likely include.

Next, there's the questionable choice of making autocomplete-needed variables class properties. The reason is fairly simple. Class properties can be annotated using phpdoc to provide a hint of their type. Using that hint, Eclipse PDT has no problem autocompleting...

The gotcha is the obvious ones. The variable needs to be a class property. If it looks untidy, the class connect can be removed by editing `$this->var` down to a local `$var`.

Anonymous on Jan 12 2008, 11:59

Rob, you always can write:

```
/** * @var Zend_Mail */$mailer = Zend_Factory::create('Zend_Mail');
```

... and get autocomplete in Zend Studio and PDT.

Anonymous on Jan 14 2008, 19:20

I know that I can

I was talking about wider acceptance by normal developers who don't write or comment on blogs

Regards,

Rob...

Anonymous on Jan 14 2008, 19:44

I have been using rspec for a while now and am really glad to see some real world usage examples of BDD with php in a good php framework. I look forward to reading more about your experiences with it.

Anonymous on Feb 22 2008, 14:57