

Sunday, September 30, 2007

PEAR OpenID support packages released

After proposing these back in June/July (and getting held up by August's vacation!) I have gotten around to releasing three packages on PEAR which are required for an OpenID package later on.

[Services_Yadis](#)

[Crypt_DiffieHellman](#)

[Crypt_HMAC2](#)

All are released as beta. Next step is getting the OpenID Consumer proposed...

Update: I forgot to thank my PEAR peers whose feedback and assistance on the proposal road was exemplary.

Posted by Pádraic Brady in Openid and Yadis, PHP General, PHP Security at 01:57

How a proposal process could work - if retaining contributors is a goal

Over the last year in open source PHP I've spent time wheeling through various projects making proposals and it's apparent that some I work really well with (some people are a genuine joy even), and others just end up getting me worked up to the point of ranting. I rarely rant - it's unconstructive, often makes you look like an ass, but given enough motivation I'll do it anyway to vent a little.

The problem with ranting from a Project POV, is that it's coming from the one person a proposal process is supposed to motivate: the proposer. When a proposer turns up, you are lucky to have found someone willing to give freely of their time. To write code and documentation. To shepherd or advocate some idea which (hopefully) would be useful to a project's users. So you really don't want to give these folk the wrong message that they are wasting their time, or contributing to so much empty air, or will be beaten into submission by everyone else with an IDE and a PHP Manual.

But I've ranted twice this Summer already. So let's go simple.

What features would Paddy's preferred style of proposal process have?

1. A proposal is not judged on how much code it contains unless code is the basis of its acceptance

Because it's a proposal - for code that WILL be written. Judge on its merits - it's goals, objectives and ultimate utility. Don't put code on a pedestal. Put goals on a pedestal. If code IS required - then for God's sake say so up front instead of wasting my time. Otherwise the code is likely some tracer implementation for illustrative purposes.


Here is the last comment I made about such code in one proposal:

"Please be aware the svn code is only intended as an experimental example and it has known issues which may require hand editing. It has served its purpose to demonstrate what actual production code could be implemented."

2. A proposal is not an invitation for others to implement its content


My number one frustration is having others dissect, implement, re-implement, and spend unbelievable quantities of email/IM space debating meaningless, implementation points. I do NOT care whether the "example" (i.e. cobbled together in 1 hour) code used `array()` or `ArrayAccess`. I propose, I advocate and if lucky I will later implement - with whatever development methodology I prefer - probably while thankfully deleting the cobbled up tracer code.

To an agile developer this sort of stuff is about as bad as it gets. NO TDD, no BDD, no set defined goals that have been

discussed, no timeline other than a version number - and everyone with a pastebin full of code I'll be judged against. This while at the same time witnessing serious talented people take that example code and using it in production because it's so incredibly useful. And it works - mostly 

3. A proposal sets goals - so take a hint and debate the goals.

Implementation is something a monkey can do, if you set the right goals. I may propose we do A, and you want B. So fine, explain B and see if I'll compromise and adopt it. When the goals are finalised (and minimised to their core) I can assemble a picture of how the eventual code will behave.

4. A proposal does not lead to the belief nobody trusts you (well, unless it's deserved 
)

It's sad, but after everyone has implemented their pet vision of your proposal, and compared it to your cobbled together code which has scarce test coverage, and then detailed why yours sucks - well, you start to wonder why. Do they think you're an idiot? Do they not trust you to go away and implement it yourself?

It's not even funny when the proposal is accepted after everyone else has implemented it.

5. A proposal should leverage off the proposer's strengths

Some people know a topic, and know it insanely well. So when they propose a new library or class which draws on that experience a project (if open to that idea) should be pretty happy. Now re-read points 1-4.

That's not an environment I ever want to spend time in.

6. Stealing a proposer's thunder, alienating them, and ignoring their work without real cause is just plain wrong

Irishman makes proposal X. Adds Y some time later. Announces Y to a public mailing list for comment. Project informs Irishman on the same public list, to the same subject, that they are about to propose something like Y. They are going to announce that same week. Irishman predicatable goes a bit crazy - being Irish. Irishman figures out in time this new Y that nobody else knows about took 3 days to cobble together. Irishman gets slightly more crazy. Irishman goes to PEAR and has actual fun. Project figures out that they are missing X (secret component to make Y useful). Irishman eventually re-proposes X and Z but not his Y.

Irishman gets invited to Y's Foundation in Europe as a Member-Subscriber advocating OpenID in Europe. Irishman finally sees the funny side. It gets funnier still later on.

7. A proposal should be granted a timeline for review and acceptance.

This way the Proposer isn't on vacation when it's supposed to be reviewed.

8. A proposal should have an informed motivated proposer

Not one so out of the loop they decide they're clueless about how the proposal process actually works outside it's documented workflow (or if the workflow just plain makes no sense).

9. A proposal process should support (or allow for) best practice in development

Not quite literally work in such a way that the only option is to take anything written by Martin Fowler, Kent Beck or Ward Cunningham and toss it out the nearest window. That not only sucks, it basically tells anyone who actually likes those

guys that you don't want them.

I really like those guys. Why don't you?

10. Acceptance notices will not hamstring the proposer's core methodology until they have sufficient time to review the notice and respond to it. I don't care how long the vacation was.

This ends the mighty list of proposal process wishes. Way better than my earlier rant!



Posted by Pádraic Brady in PHP General, PHP Security at 00:57

Sunday, September 23, 2007

Behaviour-Driven Development Explored

Since I last posted I've talked/exchanged emails to a few people interested in the concept of [Behavior-Driven Development](#), so this entry is a slightly less rambling introduction to BDD with some PHP oriented examples later. Any questions can be directed to the comments, or the usual email address.

Behaviour-Driven Development (BDD) is an evolution of Test-Driven Development (TDD). Now, as Noel Darlow pointed out in the earlier comments, any Unit Testing framework can do BDD. This is the biggest barrier to understanding BDD - since it's not completely incorrect (BDD is an evolution, not something sparkly new). Dave Astels (author on two "Practical Guide" books for TDD and XP), boils it down to a simple phrase:

"Behavior-driven development is what you are doing already, if you are doing Test-driven development well."

The phrase is an oversimplification but let's work from there.

The problem? Most people who practice TDD are not doing it well (this is a generally well known problem). Whoa, most people don't do it anyway full stop. A challenge to BDD anywhere is asserting the opposite is true - though honestly I can't see to many people arguing the point.

A core mistake that practitioners, and newcomers, make is entertaining the belief that TDD is all about testing. This is untrue! TDD is not, will not, and never has been, a testing methodology. Tests are a nice, happy side effect - but not the goal of TDD. Test-Driven Development is actually a design methodology - we perform it to improve quality of design (among other benefits).

Yet the main complaints about TDD, are related to...testing. Tests are boring; tests take too much time; I can't test complex code; PHP5 won't let me test private members, yada yada yada. So not only are most people not doing TDD well, far more never adopt TDD because they simply do not understand it from the current TDD literature - which keeps yammering on about "test-first" and other misleading catchphrases. And so the penetration of TDD has remained relatively limited.

Unfortunately, thinking in terms of testing is difficult to avoid. TDD is bombarded with the word. Every book, every article, every forum post, its very name - and every Unit Testing library. Test, test, test, test...oh, and test.

The first rule of BDD? Stop using the word "test" - it's screwing with people's heads!

We do not want to test! What we actually want to be doing, is specifying!

But behold, "test" has allies in this nefarious plot to confuse the mind - like "unit". What is a Unit? Most folk would define a unit as an independent class, method or property isolated from the rest of the system. In TDD, which utilises Unit Testing, we usually end up testing on a per unit basis. Now in a testing approach this is perfect. In a design approach it misses the point.

The second rule of BDD? Stop using the word "unit" - it's the minion of "test"!

Instead, let's pick a word with all sorts of immediate impacts: Behaviour. We do not design code by reference to units alone, we design (and refactor) by reference to the behaviour we wish our code to exhibit. Small, tiny, focused pieces of behaviour. Unit != Behaviour (you see they're spelled different). This is a foundation stone in eXtreme Programming already where prior to commencing coding, we may assemble a collection of User

Stories collected from the client and other stakeholders (like intended users) which our code must implement in order to be useful. Each story is not about units! It's about detailing required behaviour in terms a client (without a programming background) can easily understand and critique.

Next up is "assertion". It too belies a core devoted to oxymoronic forces (to design by testing doesn't make sense to many people). An assertion, in terms of testing, is a tool to verify. Verify what? Code that exists. So another insinuating hint to test - not design.

Third rule? Exile "assertion" to the wilderness. We might as well consign any thoughts of "testing state" (all those who consider testing private properties are wrong anyway) to the same place. In place of assertions, we shall have expectations - expectations of how our selected piece of behaviour will exhibit itself when we get around to writing code.

Now for many people, this will get the immediate response "BDD is just a language change". Yes, it is - but not in isolation because a change in language often precipitates a change in thinking, and the scope of such thought. So let's not get too hung up on the language differences alone. It's how BDD gets you thinking differently that's interesting. Consider the following quick descriptive exercise with the updated terminology:

"In BDD, we describe the behaviour of a system by writing executable specifications."

This, self-summarised, comment captures a goal of BDD in language a lot of experienced TDD users would feel comfortable with (because it's obviously a core facet of TDD completely buried under the test-centric nomenclature). Consider the impact on someone new to TDD - they are now informed about writing concrete examples of desired behaviour. That's a hell of a lot more useful than talking in terms of tests.

But let's not stop here! We have to address one possibly show ending issue - if TDD as described and evolved through BDD makes sense, then how do we escape the fallacy of testing, and move towards the bright light of specifying behaviour? The BDD community of users and advocates (which is growing from it's humble beginnings) has answered this with perhaps the most inflammatory of responses:

Stop using Unit Testing frameworks. One prays Marcus and Sebastian let me survive that 

Language is the big barrier. Though one can perform BDD in a testing-centric UT library, it requires an amount of skill to translate between the test oriented language and organisation of Unit Testing to the purely behaviour driven language required by BDD. It's like viewing something in Dutch, and needing to constantly translate it into English (unless you are a native speaker already). It's difficult. So difficult, that TDD is itself by relation difficult to fully grasp and perform well. TDD has had long term exposure - perhaps it's time to do the inevitable: Evolve. Now what?

Use Behaviour Specification frameworks.

Don't be afraid. Remember we're evolving, not expanding the root taxonomy of eXtreme Programming.

A BDD Framework is generally designed from the ground up to orient itself completely to specifying behaviour. As an evolution, it shares undoubtable ancestry with Unit Testing but also borrows from Acceptance Testing. The BDD Framework I'm most familiar with is called [RSpec](#), provided for the Ruby community (variants for Smalltalk, .NET and Java exist). I've been discussing a viable PHP API since my call for PHP BDD tools with others, and we've cooked up a few possibles. Here's one example inspired by RSpec:

```
require_once 'Bowling.php';
```

```

class describeBowlingResultsAfterGutterGame extends Behaviour {
    public function before() {
        $this->_bowling = $this->getSpec('Bowling');
    }
    public function itShouldHaveAScoreOfZero() {
        $this->_bowling->hitGutter();
        $this->_bowling->score->should()->equal(); // by the heavens, it's plain English!
    }
    public function after() {
        unset($this->_bowling);
    }
}

```

Now before we get into a "you can do this in PHPUnit/SimpleTest" declaration of defiance, let's repeat an obvious statement from before. BDD is an evolution of TDD. Okay? So obviously, a BDD framework will appear similar to a TDD framework. But note the differences - remember the language barrier TDD users are forced to deal with?

Now what's going on here? To specify behaviour, that behaviour must be given a context. In Bowling, one could fling that ball into the gutter (because it's a dodgy alley, and we suspect the lane is none too level) or hit a strike. Both are contexts, each giving rise to different behaviour (i.e. the expected score) within the system.

Wow, our normal English explanation looks...familiar. Hey, it's the exact same format as the BDD class! Bingo. The class encourages several guiding conventions, quite literally robbed from RSpec, including class prefixing with "describe" and method prefixing with "itShould". These get your thoughts on track and focused on what matters - describing a behaviour by specifying what it should do. In fact it can be represented in plain English by stringing all the class/method names in what is often called a "spec dox" output:

```

Bowling results after gutter game
- should have a score of zero

```

Here, I trail off and end the entry. With my own thoughts a little more focused, and some more PHP developers hopefully inspired to look deeper at BDD.

I have omitted a few things though - BDD is often looked as an evolution of TDD borrowing additional qualities from Domain-Driven Design and Acceptance Testing Driven Development (I wrote an article on [Acceptance Testing for Zend Devzone](#) during the Summer if looking for a PHP oriented flavour). Not enough room in a blog post to explore everything! One useful aspect, is something DDD calls the "ubiquitous language". The language of behaviours and specs, being an accessible readable format, is not only a programming language - it's equally accessible to everyone from your DBA to your client. Now the utility of this varies - a client may not be so intested in how Bowling works in isolation. But we're seeing something more accessible emerge. Maybe another time I'll discuss another BDD Ruby tool called rbehave (Java has JBehave) which pushes further horizons in the direction of ATDD.

Posted by Pádraic Brady in PHP General, PHP Security at 10:12

Any Behaviour-Driven Development Tools for PHP?


Behavior-Driven Development (BDD) is an evolution of Test-Driven Development (TDD) which pushes the focus away from the concept of testing to one of specifying behaviours in a form more easily understood. It's actually pretty hard to explain BDD in a few short lines but I'll throw in a quick intro after posing a question:

Are there any usable PHP Behavior-Driven Development libraries? Anywhere? I get the feeling PHP has once more fallen behind Ruby and Java which have four such libraries between them. Anyway - if not I'll have to cheat with SimpleTest for now.

Anyway, in the past I undertook to learn Ruby and got started with the RSpec BDD library after a few weeks - thanks to Brian D. for the introduction to RSpec. If you prefer a Java outlook, there's JBehave or JDave. The reason for having a replacement for the traditional Unit Testing tools is obvious after you've practiced BDD for a while. Unit Testing tools don't support the BDD terminology and ways of thinking - it's possible to cheat with JUnit (and PHPUnit/SimpleTest in my case) using test messages and careful method naming but the result isn't very pretty even if its functional, and you lose a lot of the benefits by having to put up with the "test" oriented nature of an xUnit lib and its lack of a fluid language base.

Here's a typical RSpec behaviour spec of a non-existent Fleet class for a game. Just as with TDD, we write the specification (not "test!") first and then write enough code to meet it.

```
describe Fleet, "when first instantiated" do
  before(:each) do
    # just like using setUp() in PHP UT
    @fleet = Fleet.new
  end
  it "should contain no Ships after formation" do
    @fleet.population.should eql(0)
  end
  after(:each) do
    @fleet = nil
  end
end
```

BDD avoids any mention of "test"; since it revolves around specifications. I know that sounds a bit vague - just trust me that BDD does not involve Unit Tests in the classical sense. RSpec notes itself as providing a "Domain Specific Language with which you can express executable examples of the expected behaviour of a system". See, not tests and no mention of units either 


This tiny example can be summed up by followed the keywords:

Describe a Fleet when first instantiated:
- it should contain 0 Ships after formation
- ...

This small text snippet "describes" a Behaviour by setting out what "it should" do. In fact a good BDD library will generate such text summaries for you. Honestly, after a few months of occasional RSpec usage this has become a pretty intuitive way to think of things - more so than TDD which requires you hit "behaviour"

keyword on your own initiative.


It get's cooler when you create a Squadron which needs to behave like a Fleet. RSpec gets around this by introducing the concept of "shared behaviors" which I will not describe here except to say it's a really neat approach involving Refactoring.

Any other BDD fans in the audience? Any last ditch memory of a PHP BDD library? 

Posted by Pádraic Brady in PHP General, PHP Security at 21:28


Tuesday, September 18, 2007

Eclipse PDT (PHP Development Tools) 1.0 Released Today

Actually that's a "will be", as in a matter of hours 

. The download will be available later today. Unlike those of us in GMT+, Zend Technologies is 8+ hours behind us on the US Pacific.

Edit: 1.0 is now available for download from [HERE](#) .

I'm looking forward to getting rid of the nightly build I have from God knows when, and grabbing the Eclipse PDT 1.0 release. I'm a long time Eclipse user, so I really want to get that download (when Zend get around to rolling out of bed today 

) and start using it.

The nightly I have is already cool enough. It has all the usual features one expects a good IDE to show: code completion, syntax highlighting, phpDoc features, code folding, and debugging. It has a few noteworthy extras like inspection (file and project outline, the PHP Explorer view) and integration with WTP (Web Tools Platform). It's a nice collection, and the 1.0 release should resolve a couple of issues the nightly gave me.

I think one of the issues to watch, is that PDT doesn't offer a debugger off the bat. You need to get hold of Zend Debugger, or my personal favourite XDebug. XDebug incidentally had 2.0.0 released by Derrick Rethans during July so go look it up. An all-in-one download will include the Zend Debugger if you're installing from scratch .

You can find out about Eclipse PDT over at:

<http://www.eclipse.org/pdt/>.

You should see the new 1.0 download appear in a couple of hours. If you are new to Eclipse, there will be an Eclipse PDT All-In-One style download to take the usual pain out of installing/updating Eclipse/PDT/WTP/Zend-Debugger individually by installing the lot at once. From there you can expand your platform - installing Subclipse (Subversion support!) for example.

Posted by Pádraic Brady in PHP General, PHP Security at 16:57

Friday, September 7, 2007

Prodigal PHP Developer Returns

First blog entry for a long time, and predictably it's the infamous "I'm back!" topic. So I'll be quick. I'm back. The Maug lives - though the four weeks I spent offline (mostly; I couldn't evade certain persons who pay certain salaries forever though...) has left me bumming across my previous haunts scratching my head like a bumpkin tourist.

I'm sure I'll settle back in to a few interesting posts here, and elsewhere, and in the arcane planes occupied by mailing lists. Just need to get my bearings, put up with "red-man" jokes (I can bum around Spain and get sunburnt, but mother nature, in her sorry excuse for Wisdom, decided I didn't need to tan afterwards!), and figure out how many emails I missed during August. I've replied to a couple of the more interesting ones - the rest of ye land lubbers can wait 'till tomorrow! If you're desperately needing to contact me throw an email to the usual place - I am staying current right now so few delays in responding.

Those who have been showing interest in the OpenID library I kickstarted during July will be happy to see it progressing once again shortly. Draft 12 of the OpenID 2.0 Specification has been recently published so I'll be reviewing it for any necessary changes in the libraries operation. I suspect there will be few, if any, fundamental updates. A large part of the remaining work on the library is creating an integration testing platform to recreate the various conditions and options OpenID 2.0 allows for. Unit tests (as opposed to integration tests) were always sparse since the library is specification derived (i.e. you need around 95% of completed code before OpenID Authentication even works in a basic form).

The library is, of course, being persued for distribution via PEAR, and PEAR2 in the future. It's been a month since I had a chance to discuss things with Dmitry, so a Zend Framework outlet may still be possible. I'm reasonably sure the Zend_Yadis proposal (a separate specification OpenID 2.0 requires) will be available with the Zend Framework eventually since it's 99.9% complete barring unit test coverage. You can knock around [OpenID For PHP](#) for the current versions (from July - alpha status) used to obtain acceptance to PEAR.

I suppose my other main interest is checking in on where the whole Zend_View Enhanced / Zend_Layout debate has moved over the last month, and whether there has been any adoption of solutions in my absence.

So, for those who sent emails, IMs or other stuff to me over August - working through them now. Until next time!

Posted by Pádraic Brady in Irishisms, Openid and Yadis, PHP General, PHP Security, Zend Framework at 20:40