

Writing A Simple Twitter Client Using the PHP Zend Framework's OAuth Library (Zend_Oauth)

During yesterday, I finally got around to patching and finishing Zend_Oauth's Consumer implementation for the OAuth Core 1.0 Revision A specification. Once I had it finished, I used it to write a quick and simple interface to post some Tweets on [Twitter](#) while I was testing it out. With some documentation and a few extra unit tests, the Consumer implementation should find its way into Zend Framework 1.10...along with the Server implementation I think. In this article I'll explore how to write a quick Twitter client so you can post tweets (those short messages of less than 140 characters) once authorised across the OAuth protocol.

You can download all the necessary files (just be sure to edit them as described) or pull them from git from: <http://github.com/padraic/Tweet-Lite/tree/master>

What is OAuth?

If you're not aware of what [OAuth](#) is, the OAuth specification puts it this way:

The OAuth protocol enables websites or applications (Consumers) to access Protected Resources from a web service (Service Provider) via an API, without requiring Users to disclose their Service Provider credentials to the Consumers. More generally, OAuth creates a freely-implementable and generic methodology for API authentication.

In other words, it's a means of allowing websites to access your data on other services via a service API, like Twitter's API or Google Gdata, without actually providing those websites with your username and password. Instead, OAuth allows you to authorise such websites to access your data so that they don't need your username or password - they just use an Access Token supplied by your service provider - and you can easily deauthorise them if desired. The benefit is immediately obvious - your username and password are not shared or handed out to potentially untrustworthy sites. The glut of services using Twitter are a prime example - until recently they all needed your Twitter username and password and honestly, how would you know they wouldn't misuse that? Because they said so? OAuth eliminates this problem.

The protocol works like this. The website (consumer) that wants to access your data from a service provider, contacts the provider using HTTP to retrieve a Unauthorised Request Token. The consumer will then redirect you, the user, back to your service provider so you can authorise the consumer's access. The redirect URL will contain the Unauthorised Request Token as a parameter. If you approve the access, you are redirected back to the original website with a verification code attached to the URL. The website now knows you approved its access, so it contacts the service provider, including both the newly approved Request Token (once again) and the verification code in the URL. The response to this should be a fully authorised Access Token (associated with the User) which the consumer can use in all future requests when accessing your data (until either it times out or you deauthorise the access). The Request Access token can be discarded now - in OAuth parlance you exchanged an unauthorised Request Token for an authorised Access Token.

Preparations Are Always Inevitable!

With this understanding in hand, let's get to writing this small Twitter client as an example! You will need to download the Zend Framework (whether the latest release or via subversion). You will also need to download/checkout the Zend Framework Incubator since Zend_Oauth is not yet part of the main trunk. Once you have stored both somewhere, make a note of the paths to their "library" directories so you can add them to the PHP include_path later.

On the Twitter side there are three steps.

First, get a Twitter account! Hopefully you already have one and are following @padraicb (i.e. me!).

Second, you need to configure your operating system for a new local domain. Under Linux, this is done by editing /etc/hosts. You'll need root privileges here so use "sudo myeditor /etc/hosts". Under Windows, the same file is located at C:\Windows\System32\drivers\etc\hosts and will require Administrator privileges to edit. Adding a local domain is dead simple, and it's needed to use Twitter's API on a local machine - Twitter refuse all requests from localhost or 127.0.0.1 but they won't filter out a local domain name since it's unfeasible and would make a lot of application developers extremely crazy. Edit the file to include a new entry like:

```
127.0.0.1 mytwitterclient.tld
```

Once saved, your browser should immediately respond to any address in the form of http://mytwitterclient.tld by attempting to call localhost/127.0.0.1 on your system. If you have a default web document root configured with a web server running, this is where the request will be mapped to and where you should store any files. If you are really troublesome and can't take how easy that was, you can run off to play with Virtual Host configurations to use a different path.

Third! OAuth providers like Twitter don't hand out access tokens to every Tom, Dick and Harry. Well, they do - but they like to know whether your name is Tom, Dick or Harry! You need to register all applications with them for this purpose and to get hold of a key to access their OAuth service. This is pretty much standard for all similar providers. Visiting http://twitter.com/oauth_clients and logging in using your Twitter account opens up a menu where you can register new applications/clients or delete them. Create a new client record here. Naming is not important - just make absolutely sure that a) you select "Browser" as the Application Type, b) set a callback URL of http://mytwitterclient.tld/callback.php (edit domain and path for your preferred location), and c) select "Read & Write" as the Default Access since we will be sending new Tweets and need that write access. We will not be using Twitter for logins. Once registered - it's immediate - you'll be faced with a page of details including URLs to various OAuth endpoints, and most importantly, the Consumer Key and Consumer Secret you should use for your application. Don't worry! You can revisit this page at any time.

A Glimmer Of Actual Source Code

Since we're sticking with the concept of "simple" here, this will be a scripted effort not an exercise in writing the next Zend Framework powered super app. Any permanent (these can be refreshed indefinitely so don't worry about losing them) Access Token will be stored to the current session for reuse so we can skip a database.

Our mini application is comprised of six files (none of them big): config.php, common.php, index.php, tweet.php, callback.php and clear.php. Splitting this out across a few files makes it easier to follow, understand and edit. Let's start with config.php which contains our OAuth configuration which you will need to edit for your own details.

```

<?php
define('URL_ROOT', 'http://mytwitterclient.tld');
$configuration = array(
    'callbackUrl' => 'http://mytwitterclient.tld/callback.php',
    'siteUrl' => 'http://twitter.com/oauth',
    'consumerKey' => 'xxxxxxxxxxxxxxxxxxxxxxxx',
    'consumerSecret' => 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
);

```

In a perfect world, this should be everything you need to add to your configuration. You should always be aware that a Service Provider is capable of setting their own requirements, service endpoints, and even preferred request method. A fuller configuration might look like this (still valid for Twitter since it mostly demonstrates internal defaults used by Zend_Oauth):

```

<?php
define('URL_ROOT', 'http://mytwitterclient.tld');
$configuration = array(
    'version' => '1.0', // there is no other version...
    'requestScheme' => Zend_Oauth::REQUEST_SCHEME_HEADER,
    'signatureMethod' => 'HMAC-SHA1',
    'callbackUrl' => 'http://mytwitterclient.tld/callback.php',
    'requestTokenUrl' => 'http://twitter.com/oauth/request_token',
    'authorizeUrl' => 'http://twitter.com/oauth/authorize',
    'accessTokenUrl' => 'http://twitter.com/oauth/access_token',
    'consumerKey' => 'xxxxxxxxxxxxxxxxxxxxxxxx',
    'consumerSecret' => 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
);

```

We won't worry about the details for now. The only two to bear in mind are "requestScheme" and "signatureMethod". The request scheme is a means of sending the OAuth Protocol Parameters to a service provider. By default (and it's the preferred option per the spec) we send these parameters by way of an Authorization header. There is a fallback to two other methods - as raw data in a POST body, or as part of the query string (GET or POST). The default method should be supported as recommended by the OAuth Specification. The signature method has four supported options in Zend_Oauth: HMAC-SHA1, HMAC-SHA256, RSA-SHA1 and PLAINTEXT. HMAC-SHA1 is a reasonable default, but the others may be needed on some service providers. Finally, you will note there are three URL options and no "siteUrl" as in the short version. Setting a siteUrl is perfectly fine if the actual endpoints use the convention of /request_token, /authorize and /access_token as the paths in their endpoints.

Not let's look at common.php (predictably the file all other files except config.php will be including):

```

<?php
// If you haven't edited php.ini to add the Zend Framework and the
// Zend Framework Incubator to the PHP include_path, then do so here.
// Don't use mine!
set_include_path(
    '/home/padraic/projects/zf/trunk/library'
    . PATH_SEPARATOR . '/home/padraic/projects/zf/incubator/library'
    . PATH_SEPARATOR . get_include_path()
);
// Make sure Zend_Oauth's Consumer is loaded

```

```

require_once 'Zend/Oauth/Consumer.php';
// Start up the ol' session engine
session_start();
// Include the configuration data for our OAuth Client (array $configuration)
include_once './config.php';
// Instantiate an instance of the Consumer for use
$consumer = new Zend_Oauth_Consumer($configuration);

```

The common file is pretty simple stuff. You'll note we set the include path here (you can do it in php.ini either), start our session, and instantiate a new OAuth Consumer. Zend_Oauth currently offers a full Consumer, the Server/Provider implementation will follow in the very near future.

Time to look at the starting point to our little app, index.php:

```

<?php
// include some common code
include_once './common.php';
// Do we already have a valid Access Token or need to go get one?
if (!isset($_SESSION['TWITTER_ACCESS_TOKEN'])) {
    // Guess we need to go get one!
    $token = $consumer->getRequestToken();
    $_SESSION['TWITTER_REQUEST_TOKEN'] = serialize($token);
    // Now redirect user to Twitter site so they can log in and
    // approve our access
    $consumer->redirect();
}
// Got past that if block! Must have an Access Token. Let's kick out a simple
// form for the user to submit their tweet with.
// echo the xml declaration in case shorty tags enabled - grrr. They're
// a bloody nuisance at times.
echo '<?xml version="1.0" encoding="UTF-8"?>';
?><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" >
<head>
<title>Tweet Lite Script</title>
<script language="javascript" type="text/javascript">
<!--
function imposemax(Object)
{
    return (Object.value.length <= 140);
}
-->
</script>
</head>
<body>
<?php if (isset($_GET['result']) && $_GET['result'] == 'true'): ?>
    <p style="background-color: lightgreen;">You successfully sent your tweet!</p>
<?php elseif (isset($_GET['result']) && !empty($_GET['result'])): ?>
    <p style="background-color: red;">Oops! Tweet wasn't accepted by Twitter. Probable
failure:</p>
    <div style="background-color: red;"><?php echo $_GET['result']; ?></div>

```

```

<?php endif; ?>
<p>All that work on Zend_Oauth, and all you do is send Tweets with it? <br/><br/></p>
<form action="tweet.php" method="post" id="statusform">
  <p>What do you want to say to Twitterland using 140 characters?</p>
  <!-- Bit of a JS hack without dumping in more code to do it right,
  to impose 140 char limit. It'll prevent deleting when limit reached -->
  <textarea name="status" id="status" rows="2" cols="70%" onkeypress="return
imposemax(this);"></textarea>
  <br/><input type="submit" id="submit" value="Tweet!"/>
</form>
<p><br/><br/>Click below to delete the Access Token and force start another authorisation
leg:<br/>
<a href="clear.php">Clear Access Token</a></p>
</body>
</html>

```

In the top half of this script, we're checking to see if we previously stored an Access Token (needed to use the Twitter API on behalf of the current user) as a serialized session variable. If it's there, we'll print the form, accept a status textfield, and send it to tweet.php. If we don't have an Access Token, we need to get one. Calling "\$consumer->getRequestToken();", asks the Zend_Oauth_Consumer to fire a request to Twitter asking for a new Unauthorised Request Token, and we should get one back. With this in hand, we may then redirect the user to Twitter (the redirect URI will include the token) so they can approve our access. Before we redirect the user, we'll serialise the token and store it as a session variable. Note that all tokens in Zend_Oauth are objects of type Zend_Oauth_Token.

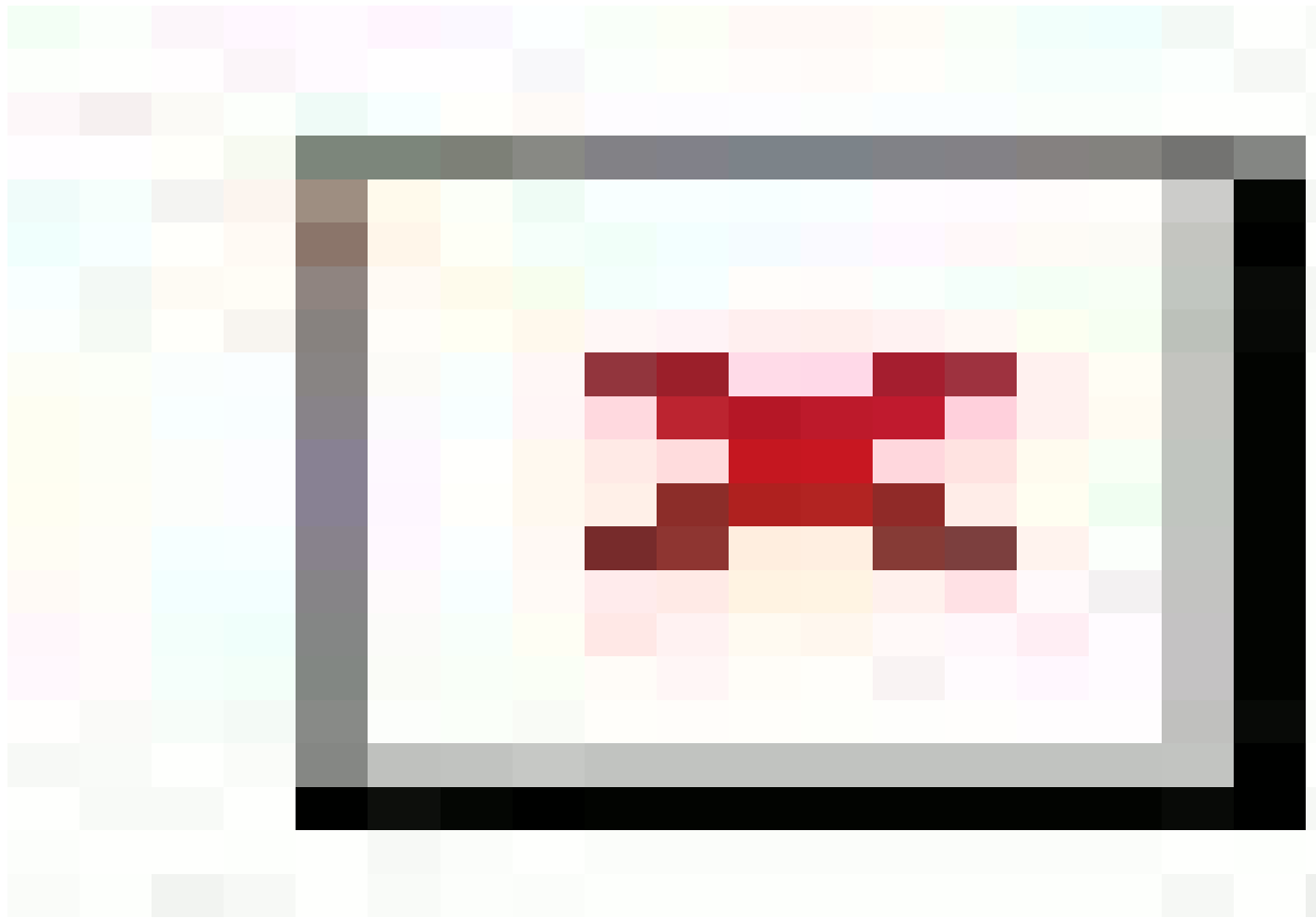
The next step happens outside the application since we redirected the user back to the provider.

Here, a user is given the option to approve our access or deny it. As Twitter notes, users may also retract their authorisation at any time. Once the user clicks the big "Allow" button, Twitter will redirect the user to us. Within the redirect URI should be a verification code (think of it like a PIN number) with which we can now request an authorised Access Token. Now, as we configured, and as it was added during registration, a user should be redirected to callback.php:

```
<?php
```

```
// include some common code
include_once './common.php';
// Someone's knocking at the door using the Callback URL - if they have
// some GET data, it might mean that someone's just approved OAuth access
// to their account, so we better exchange our current Request Token
// for a newly authorised Access Token. There is an outstanding Request Token
// to exchange, right?
if (!empty($_GET) && isset($_SESSION['TWITTER_REQUEST_TOKEN'])) {
    $token = $consumer->getAccessToken($_GET, unserialize($_SESSION[
'TWITTER_REQUEST_TOKEN']));
    $_SESSION['TWITTER_ACCESS_TOKEN'] = serialize($token);
    // Now that we have an Access Token, we can discard the Request Token
    // Keep on eye on gathering RTs in real life which are never used.
    $_SESSION['TWITTER_REQUEST_TOKEN'] = null;
    // With Access Token in hand, let's try accessing the client again
    header('Location: ' . URL_ROOT . '/index.php');
} else {
    // Mistaken request? Some malfeasant trying something?
    exit('Invalid callback request. Oops. Sorry.');
```

In the callback file, we check the incoming request for a query string and if we also have an outstanding Request Token, we pass the \$_GET array and the Request Token (unserialised from the session) to the Zend_Oauth_Consumer::getAccessToken() method. This sets up another request from the application to Twitter which includes both the Request Token and the verification (PIN) code from the user redirect. Twitter should now respond with an authorised Access Token, which we'll serialise to a session variable for future use (in a serious app, this would be associated with the user account more permanently), and with an Access Token in hand we can discard the now useless Request Token. With this done, we can redirect the user back to index.php where they will be greeted by our application in all it's glory (or not).



Now that we are here, our OAuth authorisation has been completed. But wait, there's more!

Users may now input a status message of no more than 140 characters and submit it to Twitter. The form on the page is sent to `tweet.php`, so let's see what it is doing:

```
<?php
// include some common code
include_once './common.php';
// Check for a POSTed status message to send to Twitter
if (!empty($_POST) && isset($_POST['status'])
&& isset($_SESSION['TWITTER_ACCESS_TOKEN'])) {
    // Easiest way to use OAuth now that we have an Access Token is to use
    // a preconfigured instance of Zend_Http_Client which automatically
    // signs and encodes all our requests without additional work
    $token = unserialize($_SESSION['TWITTER_ACCESS_TOKEN']);
    $client = $token->getHttpClient($configuration);
    $client->setUri('http://twitter.com/statuses/update.json');
    $client->setMethod(Zend_Http_Client::POST);
    $client->setParameterPost('status', $_POST['status']);
    $response = $client->request();
    // Check if the json response refers to our tweet details (assume it
    // means it was successfully posted). API gurus can correct me after.
    $data = json_decode($response->getBody());
    $result = $response->getBody(); // report in error body (if any)
    if (isset($data->text)) {
```

```

    $result = 'true'; // response includes the status text if a success
}
// Tweet sent (hopefully), redirect back home...
header('Location: ' . URL_ROOT . '?result=' . $result);
} else {
// Mistaken request? Some malfeasant trying something?
exit('Invalid tweet request. Oops. Sorry.');
```

When I said there's more, I meant it. From now on, every single API request concerning our user must be authorised using our OAuth Access Token. This is done, handily enough, by signing all requests using HMAC-SHA1. Now, doing this manually can be a pain, so using the Access Token object you can simply retrieve an instance of Zend_Http_Client (subclasses by Zend_Oauth_Client) which will handle all the OAuth protocol parameters and request signing transparently. Just use as normal as you would Zend_Http_Client.

So using this new client, we merely implement part of the Twitter API. Sending status messages is done by sending a POST request to `http://twitter.com/statuses/update.format` (where "format" must be changed to one of "json", "xml", "atom" or "rss") which includes a "status" parameter (in the POST body) containing our tweet. Once the response is received, we do a little checking to ascertain whether it was a success or failure, and redirect the user back to `index.php` once more.

The final file we haven't mentioned is linked to from the bottom of `index.php`'s output. `clear.php` is a simple script to delete the current Access Token and trigger a new OAuth authorisation process back through Twitter (just in case you missed it the first time around):

```


<?php
// include some common code
include_once './common.php';
// Clear the Access Token to force the OAuth protocol to rerun
$_SESSION['TWITTER_ACCESS_TOKEN'] = null;
// Redirect back to index and the protocol legs should run once again
header('Location: ' . URL_ROOT . '/index.php');
```

Now go and do some Twittering! In any other Twitter client, your new tweets should also display nearby the name of the application you registered with Twitter.

Other Considerations on OAuth?

There are a few things I'd like to mention before closing. The first is that Zend_Oauth implements Revision A of the OAuth specification (1.0a) which accounts for a session fixation vulnerability in the original specification. There is no need to worry about that here, and 1.0a is being rolled out to other service providers as we speak, if not already. Twitter implemented the improvements back in June. The component is still compatible with the original specification for service providers who haven't yet updated their implementation - it's completely transparent. Zend_Oauth also makes use of Zend_Crypt (in the Incubator also) so we offer full support for HMAC and RSA hashing via its subcomponents.

Conclusion

Well, there you have it. A simple little app for making tweets using the Twitter API over OAuth. Obviously this is a very simple example to show off OAuth but I think I kept it neat enough to explore its operation without confusing everyone thoroughly 


. In reality, tokens will need to be managed with much more care since they are valid for extended periods (in fact many times the provider won't expire them for the foreseeable future). You also need to ensure they are associated with the correct user.

Have fun with OAuth!

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 20:39

Monday, July 27. 2009


Unapologetic To The End? I think Zend_OpenId hates me... Hates it back!

Disclaimer: The following is obviously a spur of the moment tirade of sarcastic commentary and lots of (probably bad) humour. A lot of it refers to events from two years ago (that's around ZF 0.8!). Ancient history 

. Mostly. In any case, I have to rant about something once a year related to the Zend Framework. It's traditional. Park any attempt to take this too seriously at the door. I get a bit dramatic at times...

In certain places, I do sometimes get a reputation of being a crazy person. Usually it happens shortly after I find myself going down a road that leads somewhere truly undesirable (or when I throw in a weird blog post like this one). I had one of those moments today that served as a flashback to such a point in time from 2007. It's not the first time, but it will be the last since I'll ignore it completely from now on. Believe it or not, I do get a bit upset at myself when I let these times get to me. They are completely unimportant and silly - but when I have to relive them I will react in a pretty irritated fashion (ed: that should be kneejerk incendiary fashion).


This one started in 2007, right about the time I was first starting to write proposals for the Zend Framework. I felt pretty great at the time. I had lots of ideas, plenty of motivation, and then it all seemed to go downhill from there. The turning point was in June 2007, when after three months of work I felt I had enough done to propose OpenID support for the framework. Obviously a worthy cause, and also one which shouldn't cause much grief since the ZF keeps advertising its wicked web service support. Before long, that boat was scuttled with news that Zend itself had been working on just such a proposal. Okay, that was disappointing. Zend had somehow managed to completely miss a three month effort on OpenID. Just another day on the farm. Next step? Collaborate? So I waited a few days and Zend's code for what would become Zend_OpenId appeared in my inbox. Let the review commence! Okay, that ended a bit abruptly...it's a bit short. Almost like it's missing stuff and was written in a handful of days as a prototype. Talking went on and off for a while, and eventually my interest in collaborating petered out. Like any open source contributor, I participate to scratch an itch. My OpenID itch was well and truly scratched by now with my own source code, and continuing to throw time at another OpenID solution based on a different paradigm with a different ideology, little of which I agreed with, when I already had one functional was pretty pointless compared to something that still needed scratching like the Zend_View Enhanced proposal (and I really needed that shepherded through). So Zend_OpenId continued without my assistance, with something for Yadis persisting in the Incubator as an orphaned proposal. It still persists waiting that one final patch and call to the trunk... One day...

This all sounds reasonable and a bit dry, but in reality I found it all intolerable. A self-professed OpenID expert who is a member of an OpenID Foundation decides to add OpenID support to the Zend Framework after working on it for three months and who has actual code to prove it, is then told about the other proposal which just happens to be coming at the same time, and has code...of course. Well, a bit of code. If I were a real nut...I'd have a field day. Collaborate? I'd already finished this twice. I'm supposed to go for round three? Yeah, I admit, I was getting pissed off very quickly back then. I was close to constructing an elaborate conspiracy between Zend, George W. Bush and the Illuminati (how could you NOT have the Illuminati 

). What was nagging me from the start was the obvious - I should have told someone to stuff that proposal somewhere and forged ahead with my own. Of course I didn't. I was new, and had green stuff growing out of my ears, and it was Zend. The Zend. Silly me...

The summer of 2007 left me wishing the Zend Framework Proposal Process would shrivel, die and get gobbled down by Orcses (I was hoping for Gollum but he's a hotshot movie star now). It remains the single worst open source experience I have had since I joined the open source gig in 1995. Not only was it discouraging, it was demoralising. There were points when I was driven to contemplate whether my input was even wanted. Which is probably why I remain so overly sensitive about it - I can understand the new guy is immediately suspect. But not like this. In September 2007 I finally

wrote the blog post "[How a proposal process could work - if retaining contributors is a goal](#)" out of frustration. As the title suggested, and others guessed, I was borderline on taking a hike from contributing. Some months later I gave some ground, and [deleted Zend_Yaml as a proposal](#). Waiting for a review, or challenging non-existent reviews, or noticing the distinct lack of communication between people gets tiresome eventually. I have a spectacular tendency to vanish for months on projects, but not a whole year. Also, using it as a condition when reviewing another proposal was just fucking stupid. After a few bouts of initial enthusiasm, Zend_Service_Yadis joined it for any worthwhile purpose. That is my fault of course - my motivation to finish a library already released on PEAR had simply evaporated. That scratch your itch thing again - it was scratched on PEAR already. By now Yadis had been in use for over a year in production in other forms, and Zend_OpenId wasn't going anywhere.

Thankfully, since mid-2008, that horrible system seems to have mostly vanished for good. The change in topics may have helped too. Not many people have considered reimplementing Zend_Feed, and for a change, nobody had been busy on any secret OAuth components. It's taken roughly a year, but these are finding themselves in the trunk and Incubator respectively and should see ZF 1.9 and ZF 1.10 in turn. Hurrah, I committed actual finished code for once... 

. I've also dropped any pretense at applying Agile methods to proposals. When I propose something, it will have backing code, class skeletons and use cases (basically the entire integration test suite). It won't be complete, but it will be sufficiently functional. Overall, I haven't been irritated in months (well, before today).

But why, why does the whole Zend_OpenId episode still carry weight? It hit me today, when I was scanning my email and I spotted a note about some Zend Framework libraries showing their age and lack of maintenance. Perfectly normal for something the size of Zend Framework - code rot is persistent anywhere. It can be difficult to diagnose though unless you are a regular user or the maintainer, so I shot off a reply adding that the same applies to Zend_OpenId. It was only intended as a supportive example, something I'm personally familiar with since I'm the guy with "Zend Framework" and "OpenID Europe Foundation Representative" written all over their email signatures, so apparently this makes me an expert on the topic. I field a lot of questions about Zend_OpenId at times. As anyone using it knows, Zend_OpenId just doesn't work at the moment depending on the Provider. It's woefully outdated.

However, it appears that...I am the Zend_OpenId maintainer! Really? Huh? Wait a minute... WTF?

Que 2007 flashback... Que obligatory irritated kneejerk response disavowing all alleged responsibility for afore mentioned proposal I have never contributed any code to...

I know where this started. At some point in 2008 I did agree to look into OpenID 2.0 support. It was one of those goodwill moments, where you know you'll get around to it...eventually...after A, B, C and probably D. And you really mean it too, until you're dragged into someone's office and told about E. Then the mysterious Z pops by for a few months with a couple of mates. Anyway...it never went further. No proposals, no code, no discussions, no nothing. I only vaguely remember the email even. I imagine I forgot about it within a week until I was reminded about it today, and am apparently being held to it by superglue and ISP records of the email (just joking about the ISP records). God help me. Why are you laughing? This is fucking serious!

Now, being the apparent Zend_OpenId maintainer and therefore responsible for all issues filed since Spring/Summer 2008...I resign, effective immediately. It's bad for my heart, and my irritation levels. More so now that I know about it. Now that I have resigned, I can see something of the funny side. If I complain about Zend_OpenId - it really is all my fault. Even if I was unaware of it. Allegedly.

Seeing a funny side, and having the benefit of hindsight, I can recognise where it all went off the rails back in 2007. Back in 2007 I should have been more of an ass. Yessir, I should done a Hugh Laurie on someone! Instead I retreated into politeness, followed by disinterest. I should have done exactly what I did with

Zend_View Enhanced - write the bloody thing to death in a multi-part monologue that would have directed my legion of rag wearing subjects (er...readers) to assault the edifice of Zend Technologies Inc. until they caved and decided proper OpenID support was popular enough to warrant some real attention, i.e. raise bloody murder. If nobody listens in private, then shout it from the rooftops. But I didn't. And Zend_OpenId is broken. And the OpenID Community is all the poorer for it. So it really truly is all my fault. I wasn't a big enough ass. I will try to endeavour to be a bigger asshole in the future. May god have mercy on all your souls.

Now that we've established blame, and didn't even need svn to do it (since I'm not listed there...aha), can we update a few records? In the maintainer field, right on the table called Zend_OpenId, can we change that to NULL or "Zend" since they actually wrote it so presumably are responsible also for its maintenance? While we're at it, can we make it a new rule that the ex-proposer of features later proposed by a completely different party, not be held responsible for the completely different party's screwups? Thanks.

This years "Paddy Rants About Something In The Zend Framework You Couldn't Care Less About" was brought to you by...well...me.

Posted by Pádraic Brady in Irishisms, PHP General, PHP Security, Zend Framework at 01:50

Sunday, July 19. 2009

Zend_Feed_Writer and Zend_PubSubHubbub In Proposal Queue

I have a few proposals with Zend Framework. I also have an established record of not finishing them very reliably, yay! Ok, so that's not a good thing. I seem to have established a weird tradition of finding myself in just the right scenario at the completely wrong time to hold me up. Luckily (it's a bit like thinking THIS year will see a real Summer in Ireland), I do have oddles of time to burn right now. The ZF book is finally climbing Reboot Hill (the translated versions are progressing, and I will get to the next chapter soon - lots happened in the ZF since the drafts were written). Zend_Feed_Reader is in the ZF trunk, and definitely will be in 1.9. Zend_Oauth has been reviewed by me and patches are incoming in the next day to finish it (that should make it's way into 1.10 along with Zend_Crypt).

Before this all goes sour and I a) wind up hospitalised after a freak accident involving Stephen Fry running me down while sending a Tweet on an iphone, b) see Eircom attacked by mysterious hackers intent on the downfall of Brian Cowen because he shows his appreciation for free speech by introducing legislation with the new criminal act of "blasphemy" defined or c) I'm abducted by Kerry men (just because...well...they're all mad down there), I've started pounding frantically on my keyboard for a new proposal called Zend_Feed_Writer.

ZFW is the counterpart to Zend_Feed_Reader (as if that wasn't obvious). It's purpose is to, once again, offer an alternative to the current Zend_Feed component using similar principles to those applied in Zend_Feed_Reader:

1. Use a simple, intuitive and limited API to eliminate guesswork and uncertainty.
2. Utilise PHP's DOM to handle the complex internal construction of feeds.
3. Adhere to standards: RSS 2.0 (based on the RSS Advisory Board 2.0.11 spec) and Atom 1.0 (RFC 4287)
4. Behind the scenes, implement support for commonly used RSS/Atom modules like Dublin Core/Slash/Atom Threads
5. Allow users to implement/register Extensions (i.e. plugins) to add support for other modules
6. Liberally throw tests at every conceivable (including the possibly insane) scenario for use.

Just like Zend_Feed_Reader, there is obviously a question mark. Do we even need an alternative to Zend_Feed? ZFR had the major advantage that it was conceived of not only as a major simplification making developer's lives significantly easier, but also as something that understood feeds - it was able to sift through numerous alternatives for commonly queried data points until it found a match, removing the need for developers to take on that role with custom abstraction layers and interpretive work. Zend_Feed_Writer does something similar, only in reverse. It creates feeds based on the most commonly inputted data points which contain the most logical or specification defined elements. It removes the guesswork, the need to cram up on the RSS/Atom standards, the specifications and the specs for all the different modules used. It eliminates work - and that's always been the main goal. If you want to learn a bit more - look at the Zend Framework 1.9 Preview, and compare the documentation for Zend_Feed with that for Zend_Feed_Reader. It should highlight where both diverge in a meaningful way.

I'm currently drafting the Zend_Feed_Writer proposal over at [Zend_Feed_Writer - Padraic Brady](#).

Which brings us to upcoming proposal number two, the colourfully named Zend_PubSubHubbub! Or Zend_Feed_PubSubHubbub depending on which name is most appropriate.

To explain, PubSubHubbub defines a protocol where any subscriber (that's you) can subscribe to a hub server which notifies you when any feed you want to follow has changed. Wait for it... Publishers, the origin of the RSS/Atom feeds you want to follow, can then notify their hub server which in turn notifies you. It's a push system. Publisher adds new content, and notifies Hub immediately. The Hub can then push the update to you right away. There is no polling every 30 mins for new content - it's delivered to you, or you are told where to fetch it from, right away. It eliminates the delay between source and polling, the lack of which has given services like Twitter a major advantage in getting news/articles/videos out to hundreds of people almost

instantaneously and seen traditional feeds lose some of their attraction. As the PubSubHubbub team put it:

A simple, open, server-to-server web-hook-based pubsub (publish/subscribe) protocol as an extension to Atom.

Parties (servers) speaking the PubSubHubbub protocol can get near-instant notifications (via webhook callbacks) when a topic (Atom URL) they're interested in is updated.

The protocol in a nutshell is as follows:

An Atom URL (a "topic") declares its Hub server(s) in its Atom XML file, via . The hub(s) can be run by the publisher of the Atom, or can be a community hub that anybody can use. (RssFeeds are also supported!)

A subscriber (a server that's interested in a topic), initially fetches the Atom URL as normal. If the Atom file declares its hubs, the subscriber can then avoid lame, repeated polling of the URL and can instead register with the feed's hub(s) and subscribe to updates.

The subscriber subscribes to the Topic URL from the Topic URL's declared Hub(s).

When the Publisher next updates the Topic URL, the publisher software pings the Hub(s) saying that there's an update.

* The hub efficiently fetches the published feed and multicasts the new/changed content out to all registered subscribers.

The protocol is decentralized and free. No company is at the center of this controlling it. Anybody can run a hub, or anybody can ping (publish) or subscribe using open hubs.

Note, while Atom is prominently mentioned - the protocol supports RSS also (be kind of stupid if it didn't!). Atom however is a basic unit in its operation, just like it's an excellent basic unit to utilise in any web service dealing with collections of items defined in an XML syntax.

The reason a PubSubHubbub proposal is interesting to me (besides always being game for a challenge) is that like OpenID and OAuth, it's another decentralised open protocol that operates over HTTP. Also, the basic units are already or will soon be implemented/released! Zend_Feed_Reader (ZF 1.9), Zend_Oauth (ZF 1.10), Zend_Crypt (ZF 1.10) and Zend_Feed_Writer (ZF 1.10 with a little luck). Putting the protocol on top of those ready to go components will save a lot of time and effort at the end of the day.

To be honest though, I have a few doubts on this one because PubSubHubbub is so new that it is only starting to seep into implementations in the wild. So getting it into the Zend Framework right now might not happen - as an early first spec its implementation will be continually evolving/growing over many months). I'll what a review brings from the community once the proposal is written up this week.

That said, a week ago the spanking brand new [PubSubHubbub Core 0.1 Specification \(July 8, 2009\)](#) was implemented in at least one meaningful way - [initial support has been implemented in FeedBurner](#). Then we have a Wordpress plugin in progress, and several reference implementations including the for Google App engine. Still early days though. Of course, PubSubHubbub was created by Google engineers (Google run FeedBurner too) but it's really a brilliant protocol, in my opinion, compared to something using Jabber/XMPP or worse which is overly complex (with a few exceptions) for this use case (HTTP+REST FTW!). I can see this easily taking off in a big way in the future once a number of full stream uses exist - maybe Google Reader will come next and that's hugely popular.

Zend_Feed_Reader promoted to Zend Framework trunk (watch out for ZF 1.9!)

I'm happy to say that Zend_Feed_Reader has been made it through its two week cleanup effort and emerged from Matthew's review to join the Zend Framework trunk. Once the Zend Framework 1.9 release process spins up I look forward to more feedback how this component has turned out. Thanks to Jurri Stutterheim (my co-conspirator), Matthew Weier O'Phinney for shepherding this through, and a special mention goes to Kawsar Saiyeed whose feedback while using Zend_Feed_Reader to build a feed aggregator over the past week was invaluable.

Zend_Feed_Reader grew out of my need to have something that is not just capable of reading feeds, but was capable of understanding and interpreting them. If you've used Zend_Feed, you know that getting something simple like content, or a creation date, is a task that requires a bit of work. Feeds come in three distinctly different forms: RSS, RDF/RSS and Atom, all with multiple versions. Each has it's own way of presenting information. Each can also utilise extensions like RSS's popular Dublin Core 1.1 module or Atom's Threaded Extensions RFC. Getting a simple point of data can mean sifting through feeds to see what type and version they are, what elements to look for, what alternatives exist, and what alternatives should be prioritised over others. It's work that has led developers to write long classes designed to handle the task. Zend_Feed is also not without its flaws. Its API is inconsistent, its namespace handling questionable, and extending it is not as easy as it looks.

So, is Zend_Feed_Reader any different? Well, I hope so



. But you'll have to dig a little deeper to see it. Here's a quick

example from the documentation showing off the API briefly.

```
$feed = Zend_Feed_Reader::import('http://www.planet-php.net/rdf/');
$data = array(
    'title' => $feed->getTitle(),
    'link' => $feed->getLink(),
    'dateModified' => $feed->getDateModified(),
    'description' => $feed->getDescription(),
    'language' => $feed->getLanguage(),
    'entries' => array(),
);
foreach ($feed as $entry) {
    $edata = array(
        'title' => $entry->getTitle(),
        'description' => $entry->getDescription(),
        'dateModified' => $entry->getDateModified(),
        'author' => $entry->getAuthor(),
        'link' => $entry->getLink(),
        'content' => $entry->getContent()
    );
    $data['entries'][] = $edata;
}
```


Here's a similar example using Zend_Feed (based on its manual Introduction).

```
Zend_Feed::import('http://rss.slashdot.org/Slashdot/slashdot');
$channel = array(
    'title' => $slashdotRss->title(),
```

```

'link'      => $slashdotRss->link(),
'description' => $slashdotRss->description(),
'items'     => array()
);
foreach ($slashdotRss as $item) {
    $channel['items'][] = array(
        'title'      => $item->title(),
        'link'       => $item->link(),
        'description' => $item->description()
    );
}

```

Wow, I hear you say, that is so different! Bloody marvellous! Stow the sarcasm as I explain a bit further though 

Zend_Feed_Reader and Zend_Feed are siblings - there is little doubt there. But what the example doesn't show is what happens behind the scenes. Zend_Feed's API allows you to run literal queries against the underlying XML. Essentially, the API is tied to the structure of the feed's XML document making Zend_Feed a class with a mutable API. A description() method looks for *description* elements, in other words. The API is tied to element names, not the concept of what you're trying to extract. This is great when RSS and Atom are kind enough to agree on a *description* element in their respective namespaces, but it goes off course when they don't. For example, RSS 2.0 has the *pubDate* element whereas Atom 1.0 has the *created* and *modified* elements. What does this mean with Zend_Feed?

```

$feed = Zend_Feed::import('http://www.example.com/feed/');
$entry = $feed->current();
$dateModified = $entry->pubDate();
if (!$dateModified) {
    $dateModified = $entry->published();
}
if (!$dateModified) {
    $dateModified = $entry->modified();
}
if (!$dateModified) {
    $dateModified = $entry->updated();
}
if (!$dateModified) {
    $dateModified = $entry->created();
}
// Oh, crap. I forgot about Dublin Core... (there's DC 1.0 and DC 1.1)
// Load up Zend_Date and some detection so we can normalise these dates after

```

With Zend_Feed, we need to run a few (7 or so) literal requests - we could add in Feed type detection but that would mean we'd be moving towards a Strategy Pattern (watch the code grow then, and the unit tests). Finally, all feed types vary in how they present dates. These would have to be normalised from RFC822, RFC2822, ISO 8601 or that W3C standard. Assuming the feeds used one of these. Now rinse and repeat for anything you want from that feed...and add unit tests to ensure it all works for various feed types and versions. Ouch! You will also need to learn RSS and Atom in detail along with their common use extensions like Dublin Core, Atom Threaded, yada, yada, yada...

Zend_Feed_Reader, on the otherhand, is not a gateway to the feed's DOM. It just offers a simple lean API which is identical for every type and version of feed.

```
$feed = Zend_Feed_Reader::import('http://www.example.com/feed/');  
$entry = $feed->current();  
$dateModified = $entry->getDateModified();
```

In this case, the \$dateModified result is a Zend_Date object. You can format the date any way you want from here. Behind the scenes, getDateModified() does a ton of work (all unit tested) to decipher the current feed and locate the data you seek from all the possible alternatives that might exist. Once it's located, it's imported into Zend_Date using the correct standard.

Of course, this all means you'll have to endure a tiny easy to memorise API since you can only call getDateModified() - not pubDate(), created(), etc. Sorry for that. You'll also have to put up with the extra 660+ unit tests Zend_Feed_Reader has added to the Zend Framework suite to cover every feed type and version, every common combination of versions and XML extensions, and all the niggly things like normalisation and consistent API returns. My apologies. We even fantasised that RSS might implement the entire Atom 1.0 spec as an RSS module. Insanity.

The result is that we have two sets of methods. One operates at the feed level, the other at the entry level. Once you boil down all the possible alternative elements, that means we implement 12 methods at the feed level, and 13 methods at the entry level (seriously, that is all you need in most cases!). We also provide methods for accessing the current object's DOMELEMENT, DOMDOCUMENT and DOMXPath objects for those times you need something not provided by the current API, or you can use the Extension system (see below).

As for comparing the internal workings, Zend_Feed_Reader uses only one method for querying feeds. XPath. XPath might seem a surprising choice but, being a lazy programmer, it was the option that required the least code, made it easy to see what methods did internally since XPath queries (if you speak the query language) are easy to follow, and assisted in enforcing a uniform approach to parsing feeds. This means Zend_Feed_Reader may be (and probably is) slower than Zend_Feed. Remember we also sift across alternatives to get what you're looking for - so it could be many XPath queries per API call. We've added internal class caches so repetitive queries can skip XPath, and you can set your own persistent cache, to improve performance overall by minimising network requests. Zend_Feed_Reader also supports HTTP 1.1 Conditional GET requests to save on bandwidth, processing of unchanged feeds and network use.

Internally, Zend_Feed_Reader also implements (it'll be much improved later after some refactoring) a plugin system where you can write custom "Extensions" that add methods to query feeds or entries for RSS/Atom module data not already bundled with Zend_Feed_Reader. This will cover cases where a custom RSS or Atom module (for example, weather, podcasting or geolocation data) is included in a feed and you want access to that data easily. I hope to bundle some of the more popular standards with Zend_Feed_Reader - but I'll see what time I have left before ZF 1.9.

In the meantime - enjoy the new component. It's led a quiet existence to date so I'm eager to get more feedback on usage. Kawsar has allegedly sacrificed fingers to complete massive emails containing feedback. Personally, the most invaluable QA tool available at this point in time are warm bodies sitting at a desk, using this component, and complaining to me when it doesn't do what they expected!

Now, I only covered some basics - for a full overview of what Zend_Feed_Reader does and is capable of you should read the documentation. It's currently not in the HTML online manual, but the Docbook XML source is still quite readable:

http://framework.zend.com/svn/framework/standard/trunk/documentation/manual/en/module_specs/Zend_Feed_Rea

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 12:34