

Blog Export: Rasmus' Toys Page, <http://toys.lerdorf.com/>

Wednesday, September 28, 2011

ZeroMQ + libevent in PHP

While waiting for a connection in Frankfurt I had a quick look at what it would take to make ZeroMQ and libevent co-exist in PHP and it was actually quite easy. Well, easy after Mikko Koppanen added a way to get the underlying socket fd from the ZeroMQ PHP extension.

To get this working, install the PHP ZeroMQ extension and the PHP libevent extension.

First, a little event-driven server that listens on loopback port 5555 and waits for 10 messages and then exits.

Server.php

Posted by Rasmus in PHP at 23:10

Blog Export: Rasmus' Toys Page, <http://toys.lerdorf.com/>

Saturday, May 22. 2010

Writing an OAuth Provider Service

Last year I showed how to use pecl/oauth to write a Twitter OAuth Consumer. But what about writing the other end of that? What if you need to provide OAuth access to an API for your site? How do you do it?

Luckily John Jawed and Tjerk have put quite a bit of work into pecl/oauth lately and we now have full provider support in the extension. It's not documented yet at php.net/oauth, but there are some examples in svn. My particular project was to hook an OAuth provider service into a large existing Kohana-based codebase. After a couple of iterations this should now be trivial for others to do with the current pecl/oauth extension.

Step 1 - Create a Consumer Key registration page

In order for an application to communicate with your service you assign it what is essentially a user id and password. Except in OAuth terms these are known as the Consumer Key (CK) and shared secret. Your main job here is to make sure the CK and secret are unique and unguessable which means you need a decent entropy source. On Linux you have `/dev/random`, and the non-blocking potentially slightly weaker `/dev/urandom` mechanism. You can check how much entropy is available with:

```
cat /proc/sys/kernel/random/entropy_avail
```

In general something like the following should give you a decent random string of characters that you can use for your CK and secret:

```
function new_consumer_key() {
    $fp = fopen('/dev/urandom','rb');
    $entropy = fread($fp, 32);
    fclose($fp);
    // in case /dev/urandom is reusing entropy from its pool, let's add a bit more entropy
    $entropy .= uniqid(mt_rand(), true);
    $hash = sha1($entropy); // sha1 gives us a 40-byte hash
    // The first 30 bytes should be plenty for the consumer_key
    // We use the last 10 for the shared secret
    return array(substr($hash,0,30),substr($hash,30,10));
}
```

You can of course read more entropy and use a longer hash, like sha256 or whirlpool if you want longer keys.

Step 2 - The OAuth endpoints

It would probably be a good idea to skim the OAuth Spec at this point. We need two main end points for the OAuth sequence of requests. The first is called the "request token" end point. In my case I named it `/v1/oauth/request_token`. And since I am sliding this into a Kohana-based system, I wrote a controller (`application/controllers/v1.php`) which contained this in the constructor:

```
try {
    $this->provider = new OAuthProvider();
    $this->provider->consumerHandler(array($this,'lookupConsumer'));
    $this->provider->timestampNonceHandler(array($this,'timestampNonceChecker'));
    $this->provider->tokenHandler(array($this,'tokenHandler'));
    $this->provider->setParam('kohana_uri', NULL); // Ignore the kohana_uri parameter
    $this->provider->setRequestTokenPath('/v1/oauth/request_token'); // No token needed for this end point
    $this->provider->checkOAuthRequest();
}
```

Blog Export: Rasmus' Toys Page, <http://toys.lerdorf.com/>

```
} catch (OAuthException $E) {
echo OAuthProvider::reportProblem($E);
$this->oauth_error = true;
}
```

This makes the pecl/oauth extension do all the heavy lifting for us. We register a couple of callback functions. `lookupConsumer` will look up the CK and check if it is valid. `timestampNonceChecker` will check whether the timestamp of the request is sane and falls within the window of our Nonce checks. And this function will, of course, also check whether the provided Nonce has been used already to prevent replay attacks. And finally the `tokenHandler` callback will check whether a request or access token is valid. The `setParam('kohana_uri',NULL)` call tells the extension to ignore the `kohana_uri` parameter in the URLs because this was injected by an nginx rewrite rule and wasn't part of the original request.

The `lookupConsumer` callback happens early on. My version looks like this:

```
public function lookupConsumer($provider) {
    $consumer = ORM::Factory("consumer", $provider->consumer_key);
    if($provider->consumer_key != $consumer->consumer_key) {
        return OAUTH_CONSUMER_KEY_UNKNOWN;
    } else if($consumer->key_status != 0) { // 0 is active, 1 is throttled, 2 is blacklisted
        return OAUTH_CONSUMER_KEY_REFUSED;
    }
    $provider->consumer_secret = $consumer->secret;
    return OAUTH_OK;
}
```

Just a simple lookup in the data model and if found, the shared secret is attached so it can be used to check the signature of the request.

So, now we need to write code for the different stages of the OAuth request. I have an `oauth` method with a switch:

```
public function oauth($action=NULL) {
    if($this->oauth_error) return;

    switch($action) {
    case 'request_token':
        $token = Token_Model::create($this->provider->consumer_key);
        $token->save();
        // Build response with the authorization URL users should be sent to
        echo 'login_url=https://'.Kohana::config('config.site_domain').
            '/session/authorize&oauth_token='.$token->tok.
            '&oauth_token_secret='.$token->secret.
            '&oauth_callback_confirmed=true';
        break;
    }
```

Remember, the CK and the signature of the request has been checked already in the constructor, so by the time we get here, as long as `$this->oauth_error` is false, we can generate a request token and respond with a urlencoded set of parameters that include the unauthorized request token and secret we just generated along with a login url that the 3rd-party application will redirect the user to in order for them to authorize the request token to be used on their behalf.

Step 3 - Authorizing the request token

Back to your regular web UI, you now need to add a landing page for authorizing request tokens. Make sure the language on the page makes it clear to the user that they are authorizing a 3rd-party application to act on their behalf. It is a good idea to make them re-enter their password and explicitly click a button to do this authorization.

Blog Export: Rasmus' Toys Page, <http://toys.lerdorf.com/>

Step 4 - The Access token

Once the user has authorized the request token, the 3rd-party app needs to exchange the authorized request token for an access token. My switch case for this looks something like this:

```
case 'access_token':
    $access_token = Token_Model::create($this->provider->consumer_key, 1);
    $access_token->save();
    $this->token->state = 2; // The request token is marked as 'used'
    $this->token->save();
    // Now we need to find the user who authorized this request token
    $utoken = ORM::factory('utoken', $this->token->tok);
    if(!$utoken->loaded) {
        echo "oauth error - token rejected";
        break;
    }
    // And swap out the authorized request token for the access token
    $new_utoken = Utoken_Model::create(
        array('token'      => $access_token->tok,
            'user_id'     => $utoken->user_id,
            'application_id'=> $utoken->application_id,
            'access_type' => $utoken->access_type));
    $new_utoken->save();
    $utoken->delete();
    echo "oauth_token={$access_token->tok}&oauth_token_secret={$access_token->secret}";
    break;
```

As you can see I have a couple of data models that deal with tokens here. Token_Model is the token itself while UToken_Model links users to tokens. A token is either a request token (type 0) or an access token, (type 1) and they can be in various states. The tokenHandler callback keeps track of whether a token is used correctly. It looks like this:

```
public function tokenHandler($provider) {
    $this->token = ORM::Factory("token", $provider->token);
    if(!$this->token->loaded) {
        return OAUTH_TOKEN_REJECTED;
    } else if($this->token->type==1 && $this->token->state==1) {
        return OAUTH_TOKEN_REVOKED;
    } else if($this->token->type==0 && $this->token->state==2) {
        return OAUTH_TOKEN_USED;
    } else if($this->token->type==0 && $this->token->verifier != $provider->verifier) {
        return OAUTH_VERIFIER_INVALID;
    }
    $provider->token_secret = $this->token->secret;
    return OAUTH_OK;
}
```

Once we have handed out an access token, the 3rd-party app can store this and sign API requests with the associated shared secret and send that signature along with the consumer key. As you can see in the above token handler, there is a revocation check. At any point the user can go in through the Web UI and revoke an access token for a specific 3rd-party application.

Step 5 - An actual API call

Now that we have completed the OAuth dance, we can write API end points. Because all the oauth magic happens in our controller constructor, we are left to just implement our own api logic in our api methods. Mine look something like this:

```
public function user($cmd, $arg1=NULL, $arg2=NULL) {
    if(!$this->checkAccess()) return;
    $res = array();
    switch($cmd) {
    case 'settings':
        foreach($this->user->settings as $key=>$val ) {
            $res[$key] = $val;
        }
        $this->jsonResult($res);
        break;
    }
```

The checkAccess function uses the access token to look up the user who authorized it and sets \$this->user the same way your login flow will load the user once authenticated.

For the most part, the new OAuth Provider support in pecl/oauth takes the pain out of writing an OAuth Provider. It is smart about detecting the URL of the end points. You only have to indicate what the request_token (or 2-legged) endpoint is called, and it takes it from there. You can also pass the endpoint explicitly to checkOAuthRequest at each stage of the OAuth dance, but then you have to separate the steps more than I have here. I find this approach nice and compact and it lets me focus on the API part. I also like that it supports the OAuth ProblemReporting extension. This makes it a bit easier to track down problems. And by problems I mean signature mismatches. Everyone who has ever done anything with OAuth has struggled with them, and you will too. With the ProblemReporting extension you will get the raw string that the provider used to generate the signature. You just have to match that against what your consumer signed and you should be able to track down signature problems quickly.

Let me know if you do something interesting with the pecl/oauth extension. We may still be missing some use cases and right now there is decent momentum to fix things, so get in there and try it out.

Posted by Rasmus in PHP at 21:50

Blog Export: Rasmus' Toys Page, <http://toys.lerdorf.com/>

Monday, April 27, 2009

Using pecl/oauth to post to Twitter

I have seen a lot of questions about OAuth and specifically how to do OAuth from PHP. We have a new pecl oauth extension written by John Jawed which does a really good job simplifying OAuth.

I added Twitter support to Slowgeek.com the other day and it was extremely painless. The goal was to let users have a way to have Slowgeek send a tweet on their behalf when they have completed a Nike+ run. Here is a simplified description of what I did.

First, I needed to get the user to authorize Slowgeek to tweet on their behalf. This is done by asking Twitter for an access token and secret which will be stored on Slowgeek. This access token and secret will allow us to act on behalf of the user. This is made a bit easier by the fact that Twitter does not expire access tokens at this point, so I didn't need to worry about an access token refresh workflow.

First, a bit of DB groundwork that has nothing to do with OAuth itself. I needed a place to store the access tokens and secrets and relate them to the existing user ids. My Slowgeek user table has a numeric u_id field for each user, so that is what I am using as my primary key here:

```
CREATE TABLE twitter (  
  u_id int(10) not null,  
  name char(32) default NULL,  
  state smallint default 0,  
  token varchar(64) default NULL,  
  secret varchar(64) default NULL,  
  description varchar(255) default NULL,  
  status varchar(140) default NULL,  
  location varchar(80) default NULL,  
  followers smallint default 0,  
  mtime timestamp default CURRENT_TIMESTAMP on update CURRENT_TIMESTAMP,  
  PRIMARY KEY (u_id)  
) TYPE=MyISAM;
```

And the related DB code using PDO.

So, I now have a mechanism for storing Twitter-related data. Now for the real oauth work. First I registered my application with Twitter to get a consumer key and consumer secret. You do that at http://twitter.com/oauth_clients/new.

Because I am updating the status I needed read/write access for this app. Now the PHP code. It will probably be easier to read if you copy and paste it out of the iframe into your favourite editor.

The last thing this script did after it had gotten the access token and secret was to call `verify_credentials.json` to get the user's credentials.

That gives us json back which we can decode with `json_decode()`. There is some interesting stuff in your Twitter record. Here is mine from a few minutes ago:

Blog Export: Rasmus' Toys Page, <http://toys.lerdorf.com/>

stdClass Object

```
(
  [favourites_count] => 0
  [profile_text_color] => 666666
  [description] => Breaking the Web
  [screen_name] => rasmus
  [utc_offset] => -28800
  [profile_background_image_url] => http://static.twitter.com/images/themes/theme9/bg.gif
  [profile_link_color] => 2FC2EF
  [following] =>
  [profile_sidebar_fill_color] => 252429
  [url] =>
  [name] => Rasmus Lerdorf
  [time_zone] => Pacific Time (US & Canada)
  [protected] =>
  [status] => stdClass Object
  (
    [truncated] =>
    [in_reply_to_status_id] => 1642930101
    [text] => @DonMacAskill Floating point values are approximations in all computer languages #php #broken
    [in_reply_to_user_id] => 813491
    [favorited] =>
    [in_reply_to_screen_name] => DonMacAskill
    [id] => 1643107564
    [source] => Nambu
    [created_at] => Tue Apr 28 21:57:56 +0000 2009
  )

  [profile_sidebar_border_color] => 181A1E
  [notifications] =>
  [profile_background_tile] =>
  [followers_count] => 2112
  [friends_count] => 71
  [profile_background_color] => 1A1B1F
  [profile_image_url] => http://s3.amazonaws.com/twitter_production/profile_images/52489510/rl_normal.jpg
  [location] => Sunnyvale, California
  [id] => 928961
  [statuses_count] => 577
  [created_at] => Sun Mar 11 15:39:19 +0000 2007
)
```

The `$debug = $oauth->getLastResponseInfo();` after an oauth call will always give you info about the last call. For this verify_credentials call the LastResponseInfo is (with the actual token and secret deleted):

```
[u_id] => 1823955881
[name] => rasmus
[state] => 2
[token] => XXX
[secret] => XXX
[description] => Breaking the Web
[status] => @cdamian I don't like things strapped to my chest. My brain is a perfectly good built-in heart rate monitor.
[location] => Sunnyvale, California
[followers] => 2080
[utype] =>
[mtime] => 2009-04-26 09:07:08
```

And now you can actually send a tweet. This assumes the filename is 'twitter.php'. This shows how to POST a tweet to

Blog Export: Rasmus' Toys Page, <http://toys.lerdorf.com/>

twitter. Two things to note here. Because Twitter requires a status update to be sent as a POST request, we have to use OAUTH_AUTH_TYPE_FORM when we instantiate the oauth object here. And second, I have a CSRF-preventing crumb in the POST data. The idea here is to tie the POST body to the current user's login cookie so the bad guys can't spoof a form post to our twitter.php script.

I am hoping the code is for the most part self-explaining. You can also have a look at the great pecl/oauth examples.

Posted by Rasmus in PHP at 15:20

Thursday, March 19, 2009

Select * from World

I have been having a lot of fun with two Yahoo! technologies that have been evolving quickly. YQL and GeoPlanet. The first, YQL, puts an SQL-like interface on top of all the data on the Internet. And the second, GeoPlanet, introduces the concept of a WOEID (Where-On-Earth ID) that you can think of as a foreign key for your geo-related SQL expressions.

First some example YQL queries to get you used to this concept of treating the Internet like a database. Go to the YQL Console and paste these queries into the console to follow along.

```
select * from geo.places where text="SJC"
```

This looks up "SJC" in GeoPlanet and returns an XML result containing this information:

```
12521722
Airport
Norman Y Mineta San Jose International Airport
United States
California
Santa Clara
```

```
Downtown San Jose
```

```
95110
```

```
37.364079
-121.920662
```

```
37.35495
```

Posted by Rasmus in PHP at 15:06

Blog Export: Rasmus' Toys Page, <http://toys.lerdorf.com/>

Thursday, May 8, 2008

SearchMonkey

One of the things I have been playing with lately is Yahoo!'s SearchMonkey project. It appeals to me on many different levels. The geeky name is a play on GreaseMonkey. But instead of writing plugins that run locally in the browser, SearchMonkey is a way to write plugins for the Yahoo! Search results page that change the appearance of the results themselves. Best explained with an example. Assume I am looking for a Japanese restaurant, and on my search results page I see:

That's ok, I guess. It tells me it is somewhere in Redwood City and that it is a neighborhood restaurant, whatever that means. Compare that to:

This gets me a real address and phone number plus a number of other useful bits of information. That is the first level SearchMonkey appeals to me on. The usefulness is obvious. My usefulness test is to see if I can explain it to my mother. Having her search for recipes and get pictures of dishes, ingredients and preparation times right on the search results page makes this an easy sell.

The second level this appeals to me on is the way it is implemented. Writing these SearchMonkey plugins becomes much simpler if the site you are writing the plugin for uses microformats of some sort. hCard, hCalendar, hReview, hAtom, xfn or generic structured eRDF or RDFa tags. The data can also be collected via a separate XML feed that can then be converted via XSLT in the SearchMonkey developer tool. The microformat data is collected and indexed and when you go to write a plugin and specify the url pattern you are writing the plugin for, it will find whatever indexed metadata it has for that url. If it doesn't have what you are looking for, you can still write a custom data scraper to get it, but that gets a bit more involved. I really like that the easy path is to add some sort of semantic markup to the pages. Yes, as Micah points out, this is not the (uppercase) Semantic Web, but it is still a push towards semantic markup. Having such a tangible and visible result of adding semantic tags is going to encourage people other than microformat geeks to do so. The more semantic markup we get, the better off the Web is.

The third part that appeals to me is the way the plugins are written. You write a little snippet of PHP. It is actually a method in a class you can't see, but its job is to return an associative array of data such as the title to display, the summary, extra links to show and whatever other key/value pairs you might want in the output. Because you have a full-featured scripting language available, you can write quite complicated logic in one of these plugins and pull whatever data you want from the site the plugin is written for.

You can also write an add-on to your plugin which is called an Infobar. It is a little bar that is shown below the plugin and from an Infobar you can access arbitrary external services. This example shows it well:

This one shows an OpenTable reservation link and a Yelp review, but almost anything can go there as long as you can squeeze it into the limited space you have.

The SearchMonkey is still in its infancy. It needs developer support. If you are in Silicon Valley, please come to the Developer Launch Party next week on Thursday May 15. See the link for details. If you aren't in the area, or even if you are, sign up for a developer account at <http://developer.yahoo.com/searchmonkey/preview.html> and help encourage the Web to become more semantic.

Posted by Rasmus in PHP at 14:50

Blog Export: Rasmus' Toys Page, <http://toys.lerdorf.com/>

Tuesday, January 30, 2007

Want a PHP job?

Want to work on some of the busiest and coolest web apps in the world?

Do you like Flickr, and want to work downtown San Francisco?

Or perhaps you are into music, movies or TV and want to work out of Santa Monica? Jumpcut? Or have you seen answers.yahoo.com? Address Book, Personals, Search, Premium Services, Hot Jobs? Want to do interesting things combining PHP and Flash?

Yes, I get a referral bonus, but I need more toys. You get a cool job though, so I think we are even.

Send me your resume and let me know what sort of stuff you are interested in or poke around on <http://careers.yahoo.com/> and let me know which job interests you and I will forward your resume to the appropriate hiring manager.

[edited to remove RSS ad test I had forgotten about]

Posted by Rasmus in PHP at 23:36

Blog Export: Rasmus' Toys Page, <http://toys.lerdorf.com/>

Sunday, March 5, 2006

Get your geo plugin here

PHP 5, JSON from pecl cvs, Map Tile API and the Yui Connection Manager.

Toss them in a bag and shake and you get something like this. Enter a city name, or even a pseudo-name like Philly or SFO and hit return. Each time you enter a new name you will get a map tile for that city.

It's not all that fancy, but it sure is easy to do:

33 lines total which includes about 6 lines of PHP which consists mostly of building the URL to the map tile service. Then the Javascript which has a callback function to read the form values and make the backend GET request and a simple function to take the JSON response and add the map tile to the document. And finally the actual HTML form.

With a slight tweak you can change it to make a nice geocode lookup field.

So, fire up your text editors and start writing some plugins for blog and forum packages and perhaps image gallery applications as well.

Posted by Rasmus in PHP at 15:50

Monday, February 27, 2006

The no-framework PHP MVC framework

March 1, 2006 - Disclaimer: Since a lot of people seem to me misunderstanding this article. It isn't about OOP vs. Procedural programming styles. I happen to lean more towards procedural, but could easily have gone more OOP. I simplified the code a bit for brevity, but have added a light OO layer back in the model now. Not that it makes a difference. What I was hoping to get across here is a simple example of how you can use PHP as-is, without additional complex external layers, to apply an MVC approach with clean and simple views and still have all the goodness of fancy Web 2.0 features. If you think I am out to personally offend you and your favourite framework, then you have the wrong idea. I just happen find most of them too complex for my needs and this is a proposed alternative. If you have found a framework that works for you, great.

So you want to build the next fancy Web 2.0 site? You'll need some gear. Most likely in the form of a big complex MVC framework with plenty of layers that abstracts away your database, your HTML, your Javascript and in the end your application itself. If it is a really good framework it will provide a dozen things you'll never need.

I am obviously not a fan of such frameworks. I like stuff I can understand in an instant. Both because it lets me be productive right away and because 6 months from now when I come back to fix something, again I will only need an instant to figure out what is going on. So, here is my current approach to building rich web applications. The main pieces are:

PHP 5
Yahoo! User Interface Library
JSON

MVC?

I don't have much of a problem with MVC itself. It's the framework baggage that usually comes along with it that I avoid. Parts of frameworks can be useful as long as you can separate the parts out that you need. As for MVC, if you use it carefully, it can be useful in a web application. Just make sure you avoid the temptation of creating a single monolithic controller. A web application by its very nature is a series of small discrete requests. If you send all of your requests through a single controller on a single machine you have just defeated this very important architecture. Discreteness gives you scalability and modularity. You can break large problems up into a series of very small and modular solutions and you can deploy these across as many servers as you like. You need to tie them together to some extent most likely through some backend datastore, but keep them as separate as possible. This means you want your views and controllers very close to each other and you want to keep your controllers as small as possible.

Goals for this approach

Clean and simple design

HTML should look like HTML

Keep the PHP code in the views extremely simple: function calls, simple loops and variable substitutions should be all you need

Secure

Input validation using `pecl/filter` as a data firewall

When possible, avoid layers and other complexities to make code easier to audit

Fast

Avoid `include_once` and `require_once`

Use APC and `apc_store/apc_fetch` for caching data that rarely changes

Stay with procedural style unless something is truly an object

Avoid locks at all costs

Example Application

Here is the example application I will be describing.

It is a form entry page with a bit of Javascript magic along with an sqlite backend. Click around a bit. Try to add an entry, then modify it. You will see the server->client JSON traffic displayed at the bottom for debug purposes.

The Code

This is the code layout. It uses AJAX (with JSON instead of XML over the wire) for data validation. It also uses a couple of components from the Yahoo! user interface library and PHP's PDO mechanism in the model.

The presentation layer is above the line and the business logic below. In this simple example I have just one view, represented by the add.html file. It is actually called add.php on the live server, but I was too lazy to update the diagram and it really doesn't matter. The controller for that view is called add_c.inc. I tend to name files that the user loads directly as something.html or something.php and included files as something.inc. The rest of the files in the presentation layer are common files that all views in my application would share.

ui.inc has the common user interface components, common.js contains Javascript helper functions that mostly call into the presentation platform libraries, and styles.css provides the stylesheet.

A common db.inc file implements the model. I tend to use separate include files for each table in my database. In this case there is a just single table called "items", so I have a single items.inc file.

Input Filtering

You will notice a distinct lack of input filtering yet if you try to inject any sort of XSS it won't work. This is because I am using the pecl/filter extension to automatically sanitize all user data for me.

View - add.html

Let's start with the View in add.html:

The main thing to note here is that the majority of this file is very basic HTML. No styles, or javascript and no complicated PHP. It contains only simple presentation-level PHP logic. A modulus operation toggles the colours for the rows of items, and a loop around a heredoc (

Posted by Rasmus in PHP at 14:39