

---

# **Delivering User-Centric Experiences with Hybrid Auth**

Taking the next step in our exploration of the authentication and authorization features of different open standards, we will now look at an extension that combines two of those standards: the OpenID OAuth hybrid.

In this chapter, we'll cover the specifics of how to implement this extension, explore why you may want to work with it, and delve into the extensive control you gain from using it. Let's start by defining what the hybrid auth extension is and who some of its implementers are.

## **The OpenID OAuth Hybrid Extension**

At this point, we have already discussed the value of implementing the OpenID authentication model to allow users to sign in using a variety of existing accounts, as well as the extensive amount of data and control that the OAuth standard gives you.

Now that we understand the individual specifications, we'll look into combining them to enable us to take advantage of the best parts of each in a single implementation: hybrid auth.

With the OpenID OAuth hybrid extension, we can allow a relaying party to capture a user's identity information using the OpenID implementation and then employ OAuth functionality so that the relaying party can request permission from the user to capture and set his privileged data on his behalf, giving us a much more comprehensive amount of social information from the implementation.

Next let's take a look at the implementers of hybrid auth.

## Current Implementers

There are a few major companies that currently implement the OpenID OAuth hybrid extension, two of which are Yahoo! and Google. These companies' implementations allow you to authenticate using their associated membership databases and then access their users' privileged information using the OAuth authorization aspects.

## When Should I Use OpenID Versus Hybrid Auth?

You might be wondering when it's appropriate to use a standard OpenID approach versus a hybrid OpenID/OAuth model. There are different answers to this question depending on your particular project flow. Obviously, the hybrid auth approach will deliver a lot of extra personal information from a user, but it also incurs additional overhead and technical depth. OpenID will enable a user to sign in to a site with a single login, which may not originate from the service that she is signing in to; then, using the OAuth specification, you can extract her extensive profile and personalization information. At the end of the day, you have to ask yourself what the best approach is for you and your service.

## Questions to Ask Yourself Before Choosing

Before you embark upon a particular implementation path, you should ask yourself some questions about what you need in your specific application and what's available to you from a particular provider that you are trying to integrate.

### Does the provider I am working with support hybrid auth? Where can I find out?

The first question that you should ask yourself before embarking upon any approach is "What does my provider support?" Clearly, if you're working with a service that's an OpenID provider but does not offer OAuth, you should not be looking into a hybrid auth approach.

When working with a standard OpenID implementation, you simply need to find out two things:

- Is the company that I am trying to allow a login for an OpenID provider?
- What is that company's discovery URL? (For a refresher on this topic, see the "OpenID Providers" on page 453 section in Chapter 11.)

If the first answer is "yes," and you have the discovery URL, you're ready to begin integrating OpenID authentication into your site.

Now, if you're looking into a hybrid auth approach, you'll not only need to answer the preceding questions about OpenID, but also a number about OAuth and hybrid auth, such as:

- Does the provider I’m working with support OAuth?
- Does the provider I’m working with support hybrid auth to allow me to obtain a preapproved request token from OpenID in order to exchange it for an OAuth access token?

If the answers to the preceding questions are also “yes,” then you are ready to leverage the powerful authentication and authorization functionality of a hybrid auth implementation.

If you answered “I don’t know” to any of the preceding questions, then you should check the provider’s documentation for the OpenID and OAuth specifications. Specifically, you’ll need an OpenID method for obtaining a preapproved request token at the end of the authentication process in order to implement a hybrid auth approach.

### **What information about the user am I trying to obtain?**

Your next question when deciding between the different standards should be, “What information about the user do I actually need?” This is an important question for a few reasons:

- If you simply want to offload user authentication to another provider without having to store user credentials, or want to avoid the privacy concerns associated with storing those details, then there is absolutely no reason for you to incur the overhead and effort of implementing OAuth. If, on the other hand, you want to leverage an existing user social graph, then the OAuth implementation is your best option for accessing a great deal of data.
- The provider that you are working with will most likely have a “Terms of Use” document that outlines what information about its users you are allowed to store in your own systems, and for how long. If you try to store every piece of information about a user permanently (through the OAuth process), you will more than likely be violating the provider’s terms of use.

Beyond the terms of use and authentication offload considerations, you should also be concerned about incurring an overhead that you simply may not need. We’ve touched on this already, but the point should be stressed. So, let’s take a look at this factor in a little more detail by comparing the pros and cons of each implementation.

## **Pros and Cons: Standard OpenID**

First, we’ll look at the pros and cons of using a straight OpenID implementation without the second, more extensive, OAuth steps that we will explore momentarily in the hybrid auth pros and cons list.

Pros:

- You can offload the authentication of a user to an OpenID provider such as Yahoo! or Google. Using this method, you can take advantage of the provider's large membership and security systems to log your users in to your site.
- You will not need to store user login credentials in your own database systems; rather, you simply map the OpenID user on the provider site with whatever information your application or site stores about that user.
- The straight OpenID approach is more lightweight than the hybrid auth implementation.

#### Cons:

- OpenID is simply an authentication service for verifying a user account state, not an authorization system like OAuth, which allows an application or service to perform actions on the user's behalf once authorized. What this means is that a simple OpenID integration will not be able to make signed requests to the provider site to get, set, or delete a user's social information.
- The support for OpenID extensions—such as Simple Registration, Attribute Exchange, and PAPE—is inconsistent from provider to provider. Some providers support all of the most popular extensions, while others support none. In addition, the personal information that you can obtain through such extensions varies among providers. Some providers may return a user's email address, full name, profile image, and similar information, while others may return only a single piece of data such as the email address.

To summarize, OpenID is the best choice when you are simply looking to implement an authentication system to log users in, without needing to gain access to the vast majority of their social profile or graph. Extensions like Attribute Exchange and Simple Registration offer you a means to access additional basic data about the user if needed.

## Pros and Cons: Hybrid Auth

Now let's take a look at the more extensive implementation—the OpenID OAuth hybrid approach—to see how it compares to the straight OpenID integration that we just discussed.

#### Pros:

- Since you are using OAuth for your authorization model, you will have access to a much wider array of information from user profiles. In addition to the extensive data comprising the profile systems, OAuth providers usually supply several social APIs that allow you to get, set, and delete a significant quantity of the user's profiles, connections, and activities.
- By storing the access token provided from the OAuth process, applications and services may generally run headless requests\* to process user data for the duration that the access token is valid.

- Since you are already leveraging the OpenID process, you also gain all of its pros, with the exception of its lightweight implementation.

Cons:

- When you integrate the OAuth libraries, you incur quite a large overhead in your implementation code base. Instead of the process completing when the user authenticates the application (like in OpenID), once the user signs in to authorize your application, you still need to perform all of the token exchange steps required to obtain a valid OAuth access token.
- To integrate OAuth in the hybrid auth approach, you have to set up an application with the provider in order to obtain the consumer key and secret needed for the OAuth process. This means that you will also need to implement a mechanism to manage the keys for the application.

When we put all of these factors together, we can see how having an OpenID OAuth hybrid approach can be especially beneficial. Even though integrating the OAuth libraries makes for a more heavyweight implementation, having the user's authorization for your application or website to fetch, update, and delete information on her behalf can be very powerful. The only limits to our hybrid auth approach depend on the number of social APIs that the provider makes available for accessing and modifying user data.

## The OpenID OAuth Hybrid Auth Flow

Let's take a look at the flow that makes up the OpenID OAuth hybrid extension. By breaking down the different exchanges that take place in this overall flow, we will be able to see how the individual OpenID and OAuth processes combine to generate this model.

As with the separate OpenID and OAuth flows, there are three participants in the OpenID OAuth hybrid flow that we will be working with and describing throughout this chapter:

### *The user*

This is the end user who is attempting to sign in to a site or service using one of the OpenID providers and allow the application to access and/or set his personal information on his behalf.

\* Within the OAuth process, when an application stores a valid access token for a user once he authorizes the application and then at a later date makes calls to modify his data without involving him again, such as through a cron job, this is referred to as a *headless request*.

### *The relaying party*

This is the hybrid auth consumer site that implements the OpenID login to the provider in order to allow a user to authenticate his account, and the OAuth authorization to access and set additional information for that user.

### *The hybrid auth provider*

This is the site or service that contains the membership database that the relaying party will authenticate against to log in and authorize the user to access and set his personal information.

Now that we're reacquainted with the players in this exchange, let's start our hybrid auth overview by looking at the first two steps of the process, which mirror our initial OpenID steps from Chapter 11.

## **Step 1–2: Perform Discovery (OpenID Steps 1–2)**

The first steps of the hybrid auth process will seem very familiar to you from the OpenID authentication flow overview, so we'll just briefly touch on them:

1. Request login with an OpenID identifier.
2. Perform discovery on that identifier to establish an endpoint URL from which the auth process may be displayed to the user.

At step 1, the user will provide the relaying party with the OpenID identifier of the provider that he wants to use to authenticate with (i.e., which site he wants to sign in using). Through this exchange, the relaying party will normalize and perform discovery on the identifier URL before authentication begins.

The relaying party will make a request to the provider, sending it the normalized URL from the previous step. The provider will determine whether the OpenID identifier is valid and, if so, it'll return the endpoint URL to which the user should be redirected in order to sign in and accept the permissions that the application is requesting.

## **Step 3: Request User Authentication Permissions**

Once the relaying party has performed discovery on a given OpenID provider identifier (and assuming everything went according to plan), it should now have the endpoint URI to which to forward users so they can accept the application permissions to access their data.

Using this endpoint, we will now go through the process of pushing the user through the auth screens to either allow or deny the relaying party from accessing his private data, as shown in Figure 12-1.

The process is fairly simple step: the user is presented with a page on the provider site that displays the permissions that the relaying party would like to access, such as the

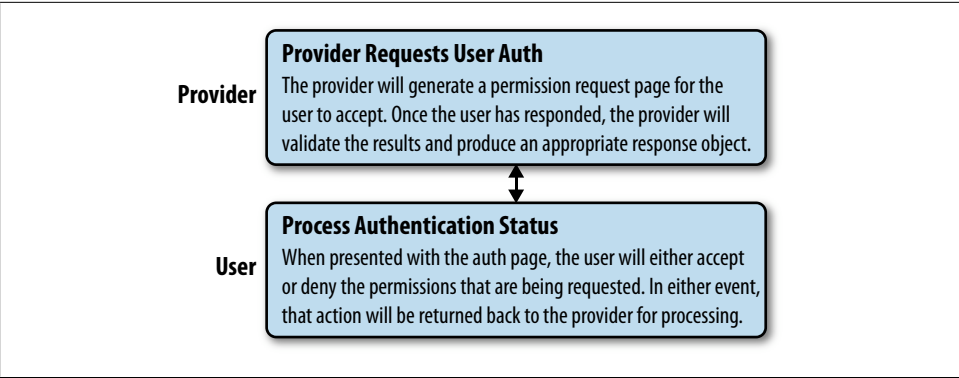


Figure 12-1. Hybrid auth, step 3: Provider requests user authentication

user’s profile, friends, activities, or Attribute Exchange values from the OpenID process.

The user will either grant or deny that request and then be forwarded back to the relaying party to either process his user information or not (depending on his choice).

The initial request that the relaying party makes to the provider site includes a number of OpenID parameters (as we discussed in Chapter 11) as well as two or three OAuth-specific hybrid parameters tacked on the end to signal that it wants to trigger a hybrid auth request. Table 12-1 lists these parameters.

Table 12-1. Hybrid request parameters

Request parameter	Description
openid.ns	The OpenID namespace URI to be used. For instance, this should be <i>http://specs.openid.net/auth/2.0</i> for OpenID 2.0 transactions.
openid.mode	The transaction mode to be used during the auth process. The possible values are <i>checkid_immediate</i> or <i>checkid_setup</i> .  If the user should be able to interact with the OpenID provider, then <i>checkid_setup</i> should be used.
openid.claimed_id (optional)	The claimed OpenID identifier, provided by the user.
openid.identity (optional)	The local OpenID provider identifier.  If <i>http://specs.openid.net/auth/2.0/identifier_select</i> is used as the identity, then the provider should choose the correct identifier for the user.
openid.assoc_handle (optional)	A handle for an association between the relaying party (implementing site) and the OpenID provider that should be used to sign the request.
openid.return_to (optional)	The location where the user should be returned, with the OpenID response, after authentication has taken place.  Many web-based providers may require this field. If this field is not included, it indicates that the relaying party does not want to return the user after authentication.
openid.realm (optional)	The URL pattern for the domain that the user should trust. For instance, <i>*.mysite.com</i> .

Request parameter	Description
	If <code>openid.return_to</code> is omitted from the request, <code>openid.realm</code> is a required parameter.
<code>openid.ns.oauth</code>	The OpenID OAuth extension namespace. This value should be set to <i>http://specs.openid.net/extensions/oauth/1.0</i> .
<code>openid.oauth.consumer</code>	The consumer key provided when you create a new OAuth application with the provider.
<code>openid.oauth.scope</code> (optional)	Scopes (the data that the application would like to access from the user) that may be required for the OAuth process.  Some providers may bind scopes directly to the consumer key once the application is created, in which case this parameter is not required.

Many of the parameters required for the request are familiar from the OpenID authentication flow. For the hybrid auth flow, the three additional request parameters are:

- `openid.ns.oauth`
- `openid.oauth.consumer`
- `openid.oauth.scope`

The user will be forwarded to the provider site to accept the permissions being requested from him, which brings us to the next step in the process.

## Step 4: Provide OpenID Approved/Failed State and Hybrid Extension Parameters

The next step in the hybrid auth process is for the provider to deliver an approved or failed state back to the relaying party so that the relaying party knows whether it can exchange the preapproved OAuth request token for an access token to complete the hybrid process.

The components of this step are identical to those of the OpenID process. The user has gone through the authentication process in step 3 and will now be forwarded to the callback with the response state from the provider, as displayed in Figure 12-2.

The OpenID provider will first process the user authentication and generate a response to the provider site. This response will either include an approved state or one of several possible failed states, depending on the outcome of the user authentication.

If the provider returns an approved state, the response object (generally sent via query string parameters) will include all parameters required to complete the OpenID process as well as any OpenID extension responses.

Besides the OpenID response, the parameter passed to the `return_to` location that we really care about for the OAuth piece of the puzzle is `openid.oauth.request_token`. This is the preapproved OAuth request token from the provider that we will need to exchange for an access token next, in step 5.



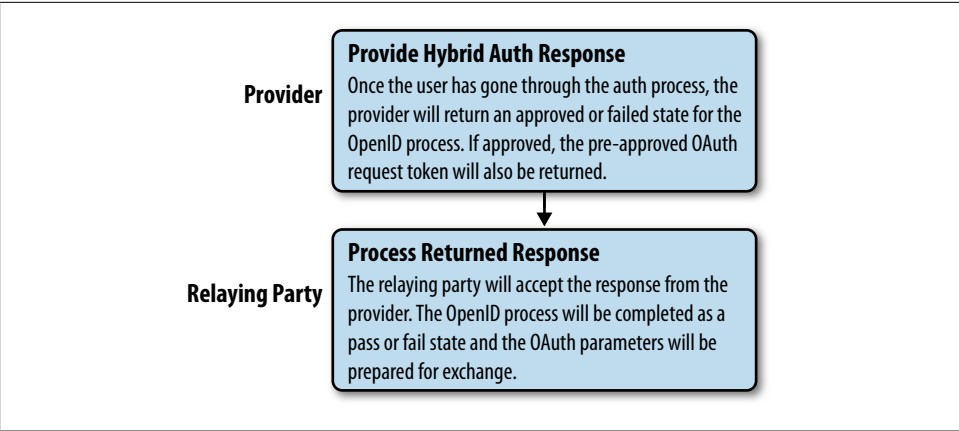


Figure 12-2. Hybrid auth, step 4: Provider returns OpenID approved/failed response

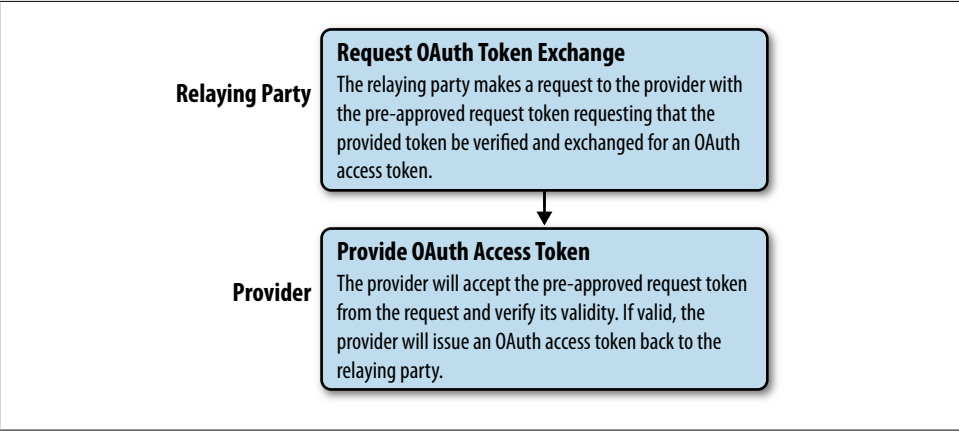


Figure 12-3. Hybrid auth, step 5: Relaying party exchanges the preapproved request token for an access token from the provider

### Step 5: Exchange the Preapproved Request Token for an Access Token

Assuming that the OpenID provider’s response was an approved state containing the preapproved request token, we can now go through the process of exchanging that request token for an access token. This will allow us to make requests to the provider for privileged user resources.

The exchange between the relaying party and provider in this step looks something like Figure 12-3.

The relaying party will issue a request to the provider to exchange the preapproved request token for an access token. With the exception of the differences in creating a

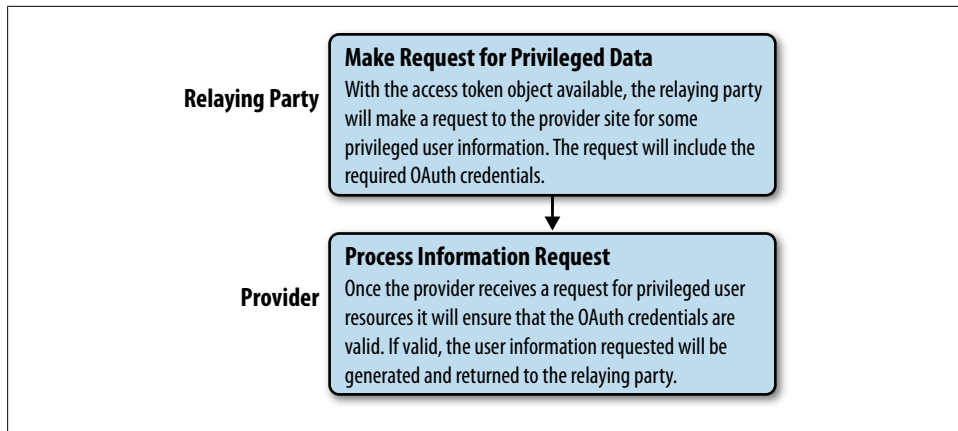


Figure 12-4. Hybrid auth, step 6: Relaying party makes privileged user data requests through the provider

request token object, this step is identical to the request token/access token exchange in the standard OAuth flow.

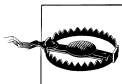
The provider will check to ensure that the request and token are valid and then return an access token string to the relaying party. The relaying party can then turn that string into an access token object and use it to make signed requests for the user's privileged data.

## Step 6: Make Signed Requests for Privileged User Data

The last step of the hybrid auth process involves the relaying party taking the access token object generated from the token exchange in step 5 and making signed requests to the provider in order to obtain privileged resources from a user, such as his profile information, friend data, or activity stream updates.

The process looks similar to Figure 12-4.

The relaying party will generate a signed HTTP request to a URI endpoint on the provider site that is set up to return the resources that we are looking for. This request will include the access token parameters from the object we generate.



Different providers accept the OAuth access token parameters in different ways. Some may require that data be sent via the HTTP headers, while others may accept the data in the POST body. You should check with the provider you are working with to ensure that you are passing through the token information in the way that it expects and requires.

The provider will receive that request, validate the access token, and issue the requested information as a response object back to the relaying party, provided that the OAuth scopes associated with the request are sufficient for accessing the requested data.

## Implementation Example: OpenID, OAuth, and Yahoo

We saw in Chapter 11 how OpenID works in a simple end-to-end example, so in this chapter let's take that example further and see how OpenID works in conjunction with OAuth to perform a standard task: using a provider such as Yahoo! to capture an end user's full profile.

We've already seen how we can capture a user's simple profile data using standard OpenID with the Simple Registration and Attribute Exchange extensions, but tacking on OAuth to the process will allow us to capture much more data than we can through OpenID alone. In this implementation example, we'll apply these two technologies as follows:

- OpenID authentication will allow our end users to sign in through different providers.
- OAuth 1.0A authorization will enable us to capture or set any private user information that we may need in our application.

For this example, we'll cut out two of the OpenID extensions (Simple Registration and PAPE) that we used in our straight OpenID example in Chapter 11.

### Application Setup: Getting Your OAuth Keys for the Hybrid Auth Process

Since we are now using a hybrid auth approach, we need to create an OAuth application with the provider of our choice (in this case, Yahoo!) to obtain the required OAuth consumer and secret keys.

For this example, we will follow these steps to set up a new OAuth application through the Yahoo! Developer Network (YDN):

1. Go to <https://developer.apps.yahoo.com/projects> to see your current YDN projects.
2. Click the New Project button at the top of your project listing.
3. In the screen that comes up, select the option for a standard OAuth application, not a YAP (Yahoo! Application Platform) application.
4. You will be presented with a blank application form asking you to fill in details about your application. Enter the data, ensuring that the application URL and domain match the location where the hybrid auth code will be running from.
5. Under Access Scopes, select "This app requires access to private user data" to indicate that we want to capture privileged user information. There are several personal information scopes that you can select, but for this example we require only read access to Profiles within the Social Directory section.
6. Click to agree to the terms of service and then click the button to get your API key. Unless you previously verified the domain, you might now have to go through a verification step. Simply follow the instructions on screen.

7. Once verification is complete, you will be presented with the consumer key and consumer secret that we'll use during the hybrid auth process.

Now that you have the keys for the hybrid auth process, we'll look at a couple of practical implementation examples.

## Implementing Hybrid Auth Using PHP



The full code for this sample is available at [https://github.com/jcleblanc/programming-social-applications/tree/master/chapter\\_12/hybrid-php](https://github.com/jcleblanc/programming-social-applications/tree/master/chapter_12/hybrid-php).

Since we now have everything we need for hybrid auth, we can begin to explore a full end-to-end example of how OpenID 2.0 and OAuth 1.0A fit together into this process.


We will build upon the base example that was introduced in Chapter 11, taking the user through the process of entering an OpenID provider identifier URL and granting the application permission to capture her personal profile information, and then finally we'll capture her data and display it on screen.

Our first task in this process is to look at the discovery form.

### The discovery form

Let's start out by exploring the HTML that will compose the OpenID form in which the user inputs the OpenID URL that she wants to sign in to. As mentioned in our Chapter 11 example, in a production-level product, you should never require a user to input the OpenID discovery URL for the service that she is trying to sign in to. One proper method is to display the logo of the company (or companies) for which you offer a sign-in option and then initiate the OpenID process for the user's selected provider without requiring her to enter any further information.

For the purposes of testing different services, though, we will build out a form that does require the user to know her preferred provider's discovery URL. This form is stored as *index.html*.



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-  
<html xmlns="http://www.w3.org/1999/xhtml">  
<head>  
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />  
<title>OpenID / OAuth Hybrid Sample Application</title>  
</head>  
<body>  
<style type="text/css">  
form{ font:12px arial,Helvetica,sans-serif; }  
#openid_url{ background:#FFFFFF url(http://wiki.openid.net/f/openid-16x16.gif)  
no-repeat scroll 5px 50%;  
padding-left:25px; }
```

```

</style>

<form action="auth.php" method="GET">
  <input type="hidden" value="login" name="actionType">
  <h2>Sign in using OpenID / OAuth</h2>
  <input type="text" style="font-size: 12px;" value="" size="40"
    id="openid_url" name="openid_url"> &nbsp;
  <input type="submit" value="Sign in"> <br>
  <small>(e.g. http://username.myopenid.com)</small><br /><br />
</form>

</body></html>

```

Much like our standard OpenID example from Chapter 11, in our hybrid approach we build out a form with a single input box to allow the end user (or developer) to enter the provider discovery URL that she would like to use. Once she enters that URL, the end user simply submits the form to begin the authentication process. There isn't really much more to the form than that, since we removed the options to utilize the OpenID PAPE policies.

### The common includes, functions, and globals

Once the form is submitted, the user is pushed to *auth.php* to begin the OpenID authentication process. Before diving into *auth.php*, however, we're going to look at *includes.php*, which is used throughout the OpenID process to provide general variables, definitions, and commonly used functions for the authentication flow.

```

<?php
require_once "Auth/OpenID/Consumer.php"; //openid consumer code
require_once "Auth/OpenID/FileStore.php"; //file storage
require_once "Auth/OpenID/AX.php"; //attribute exchange
require_once "OAuth.php"; //oauth library

define('APP_ROOT', 'http://www.mysite.com/auth/');
define('FILE_COMPLETE', 'complete.php');
define('STORAGE_PATH', 'php_consumer');

define('CONSUMER_KEY', 'YOUR KEY');
define('CONSUMER_SECRET', 'YOUR KEY');
define('APP_ID', 'YOUR APPLICATION ID');

$debug = true;
$base_url = 'http://www.mysite.com/auth/complete.php';
$request_token_endpoint = 'https://api.login.yahoo.com/oauth/v2/get_request_token';
$authorize_endpoint = 'https://api.login.yahoo.com/oauth/v2/request_auth';
$oauth_access_token_endpoint = 'https://api.login.yahoo.com/oauth/v2/get_token';

/*****
 * Function: Run CURL
 * Description: Executes a CURL request
 * Parameters: url (string) - URL to make request to
 *              method (string) - HTTP transfer method
 *              headers - HTTP transfer headers
 */

```

```

*           postvals - post values
*****/
function run_curl($url, $method = 'GET', $headers = null, $postvals = null){
    $ch = curl_init($url);

    if ($method == 'GET'){
        curl_setopt($ch, CURLOPT_URL, $url);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    } else {
        $options = array(
            CURLOPT_HEADER => true,
            CURLINFO_HEADER_OUT => true,
            CURLOPT_VERBOSE => true,
            CURLOPT_HTTPHEADER => $headers,
            CURLOPT_RETURNTRANSFER => true,
            CURLOPT_POSTFIELDS => $postvals,
            CURLOPT_CUSTOMREQUEST => $method,
            CURLOPT_TIMEOUT => 3
        );
        curl_setopt_array($ch, $options);
    }

    $response = curl_exec($ch);
    curl_close($ch);

    return $response;
}

/*****
* Function: Get Consumer
* Description: Creates consumer file storage and OpenID consumer
*****/
function get_consumer() {
    //ensure file storage path can be created
    if (!file_exists(STORAGE_PATH) && !mkdir(STORAGE_PATH)){
        print "Could not create FileStore directory '". STORAGE_PATH . "' .
            Please check permissions.";
        exit(0);
    }

    //create consumer file store
    $store = new Auth_OpenID_FileStore(STORAGE_PATH);

    //create and return consumer
    $consumer =& new Auth_OpenID_Consumer($store);
    return $consumer;
}
?>

```

This single file contains a lot of functionality for us to examine, so let's start at the top with all of our includes and definitions.

We start the file with the OpenID includes that we will need for the process, including our OpenID consumer, file storage, and Attribute Exchange. In addition to these, we

have an include for *OAuth.php*, which, if you remember from Chapter 9, is the OAuth file we need to create the authorization requests.

Our next block contains all of the standard definitions we'll use in the OpenID OAuth hybrid flow. We've seen the first three before in our OpenID example: our application root, the file to load after the end user has authenticated the application, and the file storage location for the OpenID consumer object. New to the file are our OAuth definitions—that is, our consumer key, consumer secret, and application ID. Once the OpenID authentication process has completed, we will exchange the preapproved request token that we obtain for an OAuth access token that allows us to capture private user data.

Next, we have our block of common variables for the program. These include our debug mode flag (for displaying some general signature and request information at the end of the application) and the base URL for our *complete.php* file (where we will exchange the verified request token from OpenID for the OAuth access token). We then have the endpoints that we will use for the OAuth process: the fetch request token, authorization, and fetch access token endpoints.

Last, we have two functions that we will use throughout the program flow, *run\_curl(...)* and *get\_consumer(...)*. The *run\_curl()* function will allow us to make GET, PUT, and POST HTTP requests to the services that we need to leverage. The *get\_consumer(...)* function allows us to capture a new OpenID consumer object and build a file storage mechanism.

### The authentication request

Now that we have an overview of the include file that we will be using, let's take a look at the *auth.php* file to which the initial form will forward the user in order to begin the OpenID authentication process.

```
<?php
error_reporting(E_ERROR);
session_start();

require_once "includes.php"; //configurations and common functions

/*****
 * Function: Make Request
 * Description: Builds out the OpenID request using the defined
 *              request extensions
 *****/
function make_request(){
    //get openid identifier URL
    if (empty($_GET['openid_url'])) {
        $error = "Expected an OpenID URL.";
        print $error;
        exit(0);
    }

    $openid = $_GET['openid_url'];
```

```

$consumer = get_consumer();

//begin openid authentication
$auth_request = $consumer->begin($openid);

//no authentication available
if (!$auth_request) {
    print "Authentication error; not a valid OpenID.";
}

//add openid extensions to the request
$auth_request->addExtension(attach_ax()); //attribute exchange

//generate redirect url
$return_url = sprintf("http://%s%s/%s", $_SERVER['SERVER_NAME'],
    dirname($_SERVER['PHP_SELF']),
    FILE_COMPLETE);
$trust_root = sprintf("http://%s%s/", $_SERVER['SERVER_NAME'],
    dirname($_SERVER['PHP_SELF']));
$redirect_url = $auth_request->redirectURL($trust_root, $return_url);

//attach oauth extension parameters to redirect url
$hybrid_fields = array(
    'openid.ns.oauth' => 'http://specs.openid.net/extensions/oauth/1.0',
    'openid.oauth.consumer' => CONSUMER_KEY
);
$redirect_url .= '&'.http_build_query($hybrid_fields);

//if no redirect available display error message, else redirect
if (Auth_OpenID::isFailure($redirect_url)) {
    print "Could not redirect to server: " . $redirect_url->message;
} else {
    header("Location: " . $redirect_url);
}
}

/*****
 * Function: Attach Attribute Exchange
 * Description: Creates attribute exchange OpenID extension request
 *              to allow capturing of extended profile attributes
 *****/
function attach_ax(){
    //build attribute request list
    $attribute[] = Auth_OpenID_AX_AttrInfo::make(
        'http://axschema.org/contact/email', 1, 1, 'email');
    $attribute[] = Auth_OpenID_AX_AttrInfo::make(
        'http://axschema.org/media/image/default', 1, 1, 'picture');

    //create attribute exchange request
    $ax = new Auth_OpenID_AX_FetchRequest;

    //add attributes to ax request
    foreach($attribute as $attr){
        $ax->add($attr);
    }
}

```



```

        //return ax request
        return $ax;
    }

    //initiate the OpenID request
    make_request();
?>

```

If you’ve completed the straight OpenID example from Chapter 11, this file will look familiar to you, with a few exceptions. Let’s start with the `make_request()` function.

We start the `make_request()` function by running several data checks. We check whether the user specified an OpenID URL, and if so, capture that URL and construct a new OpenID consumer object.

Next, we begin the authentication process with the provided OpenID URL. If it isn’t valid, we display an appropriate message to the end user.

We then jump into OpenID extensions. Instead of using Simple Registration, PAPE, and Attribute Exchange as we did in the Chapter 11 example, we set up a request only for the Attribute Exchange extension in our code here.

The Attribute Exchange function, `attach_ax()`, simply specifies that we want to capture the end user’s email address and photo, creates a new OpenID Attribute Exchange fetch request, adds the attributes to the request object, and then returns the request object.



Even when you’re using hybrid auth, which gives you the power to access a wider array of private user data, OpenID extensions can still come in handy. For example, in many instances accessing a user’s email address can be tricky (or can require an additional HTTP request) through standard OAuth. If you attach that request in the OpenID authentication process through the Attribute Exchange extension, however, it can save you another request down the road.

Now we need to build the redirect URL that we will call following the authentication process. We create our return URL (the *complete.php* file) and our trust root (to ensure that we are not implementing any redirect attacks) and then construct the redirect URL by calling the `redirectURL(...)` method against our OpenID object.

Next—and this is primarily where this file differs from the straight OpenID example—we set up the OAuth extension parameters that will be attached to the OpenID authentication process. This step is how we indicate that we will need to be issued a preapproved request token after the authentication request so that we can exchange it for an OAuth access token. We attach the OAuth extension spec URI and our OAuth request token that we obtained from our provider when we set up our OAuth application (as described in Chapter 11). These parameters are then appended to the OpenID redirect URI.

The last part of the file is the redirect. Instead of our straight OpenID example—where we were checking whether to redirect or print the authentication form—we now have a combined OpenID OAuth hybrid URI, so we’re going to redirect the user. We check to confirm that a redirect is possible and then initiate it.

At the end of the script, we initiate the request to the `make_request()` function. Any preprocessing or non-OpenID-related tasks should be inserted prior to this call.

The user is forwarded to the hybrid OpenID OAuth screen to accept the permissions for the application that will be requesting her information (just like we saw in Chapter 11) and, once she accepts, is forwarded on to our *complete.php* file (the callback).

### The authentication callback

Now, let’s take a look at the *complete.php* file to see how we capture the OpenID return values and transform them into an OAuth access token to make requests for private end-user data. This file contains a number of final steps that we must take in order to capture user data beyond what is provided from the OpenID process.

**Completing the OpenID process.** Our first task in the callback file is to run the final steps of the OpenID process. We do this by capturing and filtering the objects that were returned, and then running a complete check against an OpenID consumer.

```
<?php
session_start();
require_once("includes.php");

$filters = array(
    'openid_ax_value_email' => FILTER_SANITIZE_ENCODED,
    'openid_identity' => FILTER_SANITIZE_ENCODED,
    'openid_oauth_request_token' => FILTER_SANITIZE_ENCODED
);
$attributes = filter_var_array($_REQUEST, $filters);

$consumer = get_consumer();

//complete openid process using current app root
$return_url = sprintf("http://%s%s/complete.php",
    $_SERVER['SERVER_NAME'],
    dirname($_SERVER['PHP_SELF']));
$response = $consumer->complete($return_url);
```

We first set up the objects that we want to run filters against and specify which filters to run; then, we call `filter_var_array(...)` to return a filtered set of the GET and POST parameters.



When you’re working with authentication and authorization systems—or anything with user input parameters, for that matter—it’s a good idea to filter the unknown values prior to using them.

Following this, we create a new OpenID consumer object by calling `get_consumer()`. Next, we construct the *complete.php* absolute path to run a comparison against, and then initiate a request to `complete(...)` against the OpenID consumer to compare the complete URL. We then check the response state to determine whether we have a valid object that we can use to complete the hybrid auth process.

**Checking the OpenID response and processing the Attribute Exchange data.** The value of `$response->status` is the response state of the authentication process, providing us with feedback on whether authentication succeeded or not. We can use this status to control what to display to the end user, handle fail states, and trigger OpenID extension-processing and token-swapping functions.

```
//response state - authentication cancelled
if ($response->status == Auth_OpenID_CANCEL) {
    $response_state = 'OpenID authentication was cancelled';
//response state - authentication failed
} else if ($response->status == Auth_OpenID_FAILURE) {
    $response_state = "OpenID authentication failed: " . $response->message;
//response state - authentication succeeded
} else if ($response->status == Auth_OpenID_SUCCESS) {
    //get the identity url and capture success message
    $openid = htmlentities($response->getDisplayIdentifier());
    $response_state = sprintf('OpenID authentication succeeded:
        <a href="%s">%s</a>', $openid, $openid);

    if ($response->endpoint->canonicalID){
        $response_state .= '<br />XRI CanonicalID Included: '
            . htmlentities($response->endpoint->canonicalID);
    }

    //get attribute exchange return values
    $response_ax = new Auth_OpenID_AX_FetchResponse();
    $ax_return = $response_ax->fromSuccessResponse($response);
    foreach ($ax_return->data as $item => $value){
        $response_state .= "<br />AX returned <b>$item</b> with the value:
            <b>{$value[0]}</b>";
    }
}
```

There are three response states that we will be integrating into our complete file:

#### **Auth\_OpenID\_CANCEL**

The authentication process has been cancelled. In this case, we should give the user a way to use the application without having to log in, such as by delivering public-only profile content. For this example, we simply display a failure message to the user.

#### **Auth\_OpenID\_FAILURE**

Something went wrong during the authentication process. In this case, ideally you'd give the user a way to attempt reauthentication and provide some basic, user-readable error information. For this example, we simply display an error message to the user.

#### Auth\_OpenID\_SUCCESS

The authentication process succeeded. At this point, we can process the OpenID extensions and begin to convert the preauthorized request token to an OAuth access token.

For the purposes of this example, we'll assume an application SUCCESS state and proceed accordingly.

In the object that is returned from the OpenID authentication process, we first extract the display identifier for the end user by making a request to `getDisplayIdentifier()` against the response object. This is the unique identifier for the end user.

We then check if a canonical ID was returned in the object. As mentioned in the Chapter 11 OpenID example, if the `CanonicalID` field is available from the XRD (Extensible Resource Descriptor), we should use it as the key lookup field when storing information about the end user.

Next, we turn our attention to the Attribute Exchange extension response object. We obtain the Attribute Exchange information from the OpenID response object by creating a new fetch response object, instantiating a new instance of `Auth_OpenID_AX_FetchResponse()`. Then, we call the `fromSuccessResponse(...)` method against that fetch response object, passing in the OpenID response. This will give us the values that were returned from the Attribute Exchange request. In our case, these values should hold the user's email address and photo, assuming that the provider's Attribute Exchange process allows these response objects. We then iterate over the data that was returned to capture the values required.

Now that we have our Attribute Exchange data, we can focus on exchanging the pre-approved request token returned from the OpenID process for an OAuth access token that will enable us to request additional private end user information.

**Turning the OpenID preapproved request token into an OAuth access token.** We've already explored the OAuth process of exchanging a verified request token for an access token in Chapter 9. The hybrid process is not much different, since we should have been provided with a preapproved request token in the object that was returned from the OpenID authentication process.

We simply need to follow the same steps as we did in Chapter 9's OAuth process.

```
//if pre-approved request token available, start OAuth process at step 4
//reference: http://developer.yahoo.com/oauth/guide/request-token.html
if(isset($attributes['openid_oauth_request_token'])){
    $consumer = new OAuthConsumer(CONSUMER_KEY, CONSUMER_SECRET, APP_ID);
    $sig_method = new OAuthSignatureMethod_HMAC_SHA1();

    //manually generate request token object
    $req_token = new stdClass();
    $req_token->key = $attributes['openid_oauth_request_token'];
    $req_token->secret = '';

    //generate access token object
```

```

$acc_req = OAuthRequest::from_consumer_and_token($consumer, $req_token,
    "GET", $oauth_access_token_endpoint, array());
$acc_req->sign_request($sig_method, $consumer, $req_token);
$access_ret = run_curl($acc_req->to_url(), 'GET');

//if access token fetch succeeded, we should have oauth_token and
//oauth_token_secret. Parse and generate access consumer from values
$access_token = array();
parse_str($access_ret, $access_token);
$access_consumer = new OAuthConsumer($access_token['oauth_token'],
    $access_token['oauth_token_secret'], NULL);

```

The first thing that we do is to check that a request token was indeed returned from the authentication process. If you remember, at the top of the *complete.php* file we ran the following line of code to filter the GET and POST parameters passed to this page:

```
$attributes = filter_var_array($_REQUEST, $filters);
```

Now we'll start extracting some of the parameters from that object. The preapproved request token that we will exchange for an OAuth access token is available under the value `openid_oauth_request_token`, so we check to ensure that this value is present before we attempt the OAuth process.

If the value is available, we construct a new OAuth consumer object using the OAuth consumer key, secret, and application ID we obtained when we set up our OAuth application (as described in Chapter 9). We also set up the signature method object (HMAC-SHA1) that we will be using in our OAuth requests.

Now we have a request token value but not a request token object like what would be available in a standard OAuth process, so we need to create one ourselves. We create a new standard class, set a key value to the request token key from our attributes object, and then set a blank value for the secret (since we don't have one).

With our request token now in hand, we can begin the exchange process to obtain an access token that will allow us to access and set private end-user information on her behalf. We create a new OAuth request object using the `from_consumer_and_token(...)` method from the `OAuthRequest` class, passing in a few values, including:

- Our OAuth consumer object that we created (`$consumer`).
- The request token object that we manually constructed (`$req_token`).
- The HTTP request method required for the request (GET).
- The provider URI to which we make a request to obtain the access token (`$oauth_access_token_endpoint`).
- Optional parameters to attach to the request. We don't have any, so this will be a blank array.

We then sign the request using the signature method object that we set up, and make a cURL GET request to fetch the access token object from the provider.

Assuming that all went according to plan, we should have everything that we need to construct the final access token to enable us to make private data requests.

We create an empty array for the access token and then populate it with the return values from our request. Once we've done that, we create a new OAuth consumer object, passing in a few values:

- The access token (`$access_token['oauth_token']`).
- The access token secret (`$access_token['oauth_token_secret']`).
- The callback URL. We don't need one here, so we set this value to `NULL`.

With that, we can begin constructing requests to access private end-user data far beyond what we would get from a straight OpenID authentication process.

**Making requests with the OAuth access token.** Let's start putting together a request to access profile information for a Yahoo! user. We have our access token consumer object that enables us to make those private requests, so let's see how to actually get that data.

```
//build profile GET request URL
$guid = $access_token['xoauth_yahoo_guid'];
$url = sprintf("http://%s/v1/user/%s/profile",
    'social.yahooapis.com',
    $guid
);

//build and sign request
$request = OAuthRequest::from_consumer_and_token($consumer,
    $access_consumer,
    'GET',
    $url,
    array());
$request->sign_request(new OAuthSignatureMethod_HMAC_SHA1(),
    $consumer,
    $access_consumer
);

//make GET request
$resp = run_curl($request->to_url(), 'GET');

//if debug mode, dump signatures & headers from OpenID / OAuth process
if ($debug){
    $debug_out = array('OpenID Request' => $response_state,
        'Access token' => $access_token,
        'GET URL' => $url,
        'GET response' => htmlentities($resp));

    print_r($debug_out);
}
}
```

Some providers that support OAuth will attach additional identification parameters with their access tokens to denote a particular user on their network. You can use these parameters to make private data requests for that specific user's data. In Yahoo's case, the `xoauth_yahoo_guid` parameter (globally unique identifier) provides the alphanumeric identifier for the user on Yahoo! networks.



Unsure whether the providers you are working with attach additional values in their access token? Simply dump the object and look for parameters that are not standard for an access token, such as any that do not start with `oauth_`.

This `xoauth_yahoo_guid` is what we'll start with to construct our data request to the Yahoo! social APIs. To utilize this parameter, we build out the URI to which we will make the request to obtain the user's profile. This URI includes the user's GUID, as follows:

*`http://social.yahooapis.com/v1/user/GUID/profile`*

Next, we need to create a new OAuth request object and sign it. We make a request to `from_consumer_and_token(...)` in the `OAuthRequest` class, passing in our basic consumer object, the access consumer, the HTTP request type, the URL to make the request to, and a blank array to signify that we aren't passing in any additional parameters. We then sign that request using HMAC-SHA1 by calling the `sign_request(...)` method.

Last, we simply need to make a cURL request to the social URI endpoint we specified in order to obtain our profile object. The response from the cURL request (the `$resp` variable) should be the profile object of the user, provided that there wasn't an error during the request.

If we set a debug flag in our test program, we go ahead and dump out a series of objects to screen.



Signature and object verification is a key element of OAuth debugging, so it's good to get used to seeing these signature and response objects.

Finally, if the provider you are working with requires that you create an XRDS file to allow verification of your domain, you will also need to go through the step of creating that file, as described in the "Bypassing Domain Discovery Errors in OpenID" on page 453 section in Chapter 11.

## Implementing Hybrid Auth Using Python



The full code for this sample is available at [https://github.com/jcleblanc/programming-social-applications/tree/master/chapter\\_12/hybrid-python](https://github.com/jcleblanc/programming-social-applications/tree/master/chapter_12/hybrid-python).

Now that we've looked at a hybrid OpenID OAuth implementation in PHP, let's explore a similar implementation using Python.

In this example, we'll take the OpenID example that we explored in Chapter 11 and modify it by introducing a request to fetch a preapproved request token when having the user authorize the application. We will then exchange this preapproved request token for an OAuth access token, which will allow us to make requests on the user's behalf.

We'll start this process by identifying the library dependencies we need to build out this project. Then we'll proceed by jumping into the YAML file that we will use to load the example on Google App Engine, which will run our program.



If you are running this Yahoo! example in App Engine on localhost, you should be aware that if you create an application on Yahoo! (and numerous other services) to obtain the necessary OAuth keys, you may be required to verify your domain. This verification will fail on localhost and may prevent you from completing your application. You should deploy applications running within App Engine prior to executing so that your production environment can be verified.

### Library dependencies

Before we begin this project, we need to define the libraries that we'll use to complete the OAuth OpenID hybrid auth process.

**OpenID.** This example uses the Python OpenID 2.0 library created by Janrain. To install this library, please see the section “Getting the required OpenID library” on page 482 in the Python example in Chapter 11.

**OAuth.** For this example, we use the Python OAuth 1.0A library created by Leah Culver. You can obtain this library by following these instructions:

1. The OAuth code libraries are all available at <http://oauth.net/code/>. Go to that location and scroll down to the Python section.
2. Click the Leah Culver “library in Python 2.3” link to go to the file hosting location, or alternately go directly to <http://oauth.googlecode.com/svn/code/python/oauth/>.
3. Download the *OAuth.py* file and place it in the project directory. In this example, that's the *oauth* directory.



We'll use this OAuth file in this example to exchange the preapproved request token from the OpenID process for an access token, which we'll then use to make private data requests on the user's behalf.

### The markup file

The markup file that App Engine will use to run this example is fairly simple, comprising only three files. For the sake of this example, we store the YAML file as *app.yaml*.

```
application: openid-oauth-hybrid
version: 1
runtime: python
api_version: 1

handlers:
- url: /index.py
  script: index.py
- url: /auth.py
  script: auth.py
- url: /complete.py
  script: complete.py
```

Here are the three files that make up the application:

#### *index.py*

The form loader that enables the user to input the OpenID provider URI that he wants to use to authorize the application

#### *auth.py*

Creates the initial OpenID request with the required OAuth hybrid extension fields and then redirects the user to authorize the application

#### *complete.py*

Completes the OpenID process, collects the information from all extensions, and then exchanges the preapproved request token for an OAuth access token to make private data requests on the user's behalf

Now that we understand the core of the program, let's start with the request form, *index.py*, which allows the user to select the provider that he would like to use.

### The request form

Using *index.py*, we will build out the form in which the user will input the OpenID provider service he would like to connect through and authorize to get and set information on his behalf via OAuth requests. This form is simply HTML content that has the required sections. Unlike the OpenID example in Chapter 11, we have removed the OpenID PAPE extension options, since they're unnecessary in this example. Let's see what this form looks like.

```
print '''\
Content-type: text/html; charset=UTF-8
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
```

```

    "http://www.w3.org/TR/html4/strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>OpenID Sample Application</title>
</head>
<body>
<style>
form{ font:12px arial,Helvetica,sans-serif; }
#openid_url { background:#FFFFFF url(http://wiki.openid.net/f/openid-16x16.gif)
              no-repeat scroll 5px 50%; padding-left:25px; }
</style>

<form action="auth.py" method="GET">
  <input type="hidden" value="login" name="actionType">
  <h2>Sign in using OpenID</h2>
  <input type="text" style="font-size: 12px;" value="" size="40"
        id="openid_url" name="openid_url"> &nbsp;
  <input type="submit" value="Sign in"> <br>
  <small>(e.g. http://username.myopenid.com)</small>
</form>

</body></html>
'''

```

The form is simply an input box allowing the user to enter his OpenID provider URL. When he clicks the submit button, the user will be forwarded to *auth.py* to begin OpenID discovery.

Now let's start looking at the foundation of the OpenID process, beginning with the file that holds the common variables that will be used for the OpenID authentication and OAuth authorization processes that we will send the user through.

### Common variables

The common variables file, *common.py*, will simply store the variables and URLs that are needed throughout our hybrid process.

```

import os

#set oauth consumer key and consumer secret keys
consumer_key = 'djoyJmk9TWI3cno3UWJnM9Y29uc3VtZXJzZWNyZXQmeD1kYg--'
consumer_secret = '28041f6af657807faa9b9727e0a0669e0d'

#oauth access token endpoint (Yahoo!)
oauth_access_token_endpoint = 'https://api.login.yahoo.com/oauth/v2/get_token'

#trust root and return to urls for openid process
trust_root = 'http://%s/' % (os.environ['HTTP_HOST'])
return_to = 'http://%s/complete.py' % (os.environ['HTTP_HOST'])

```

These include:

`consumer_key`

The OAuth consumer key issued when you created a new OAuth application with the provider

`consumer_secret`

The OAuth consumer secret issued when you created a new OAuth application with the provider

`oauth_access_token_endpoint`

The URI to exchange a preapproved request token (from the OpenID authentication process) for an access token (for OAuth private data requests)

`trust_root`

The trust root URI (current application folder) needed for the OpenID process

`return_to`

The callback URI (the *callback.py* file) where the hybrid auth process should forward the user after authorization

Now let's check out the *auth.py* file that the user will be forwarded to once he submits the request form.

### The authentication request

The authentication request file, *auth.py*, enables us to do the following:

- Perform discovery on the OpenID provider that the user specified to determine whether it is a valid provider.
- Set up the OpenID request itself and attach any desired OpenID extensions (in this case, the Attribute Exchange extension) to that request.
- Attach any OAuth request variables to the URI to which the user will be forwarded in order to accept the application permissions.
- Forward the user to the provider to accept the application permissions, which detail the type of information that the application will be able to access on his behalf.

To dig a little deeper, let's break down the *auth.py* file into a few subcategories, starting with the step to perform discovery on the OpenID provider URI and build the initial OpenID request object.

**Performing discovery and building an OpenID consumer object.** This is our first step in the construction of an OpenID OAuth hybrid authorization request. For this piece, we'll perform discovery on the OpenID URI that the user specified in the aforementioned request form. Basically, we're validating whether the URI that the user provided is valid and, if so, we will use it as the redirect URI to send him to in order to have him accept the application permissions. We'll then add our extensions on top of this request and apply the OAuth hybrid parameters to the redirect callback location contained within the request object we obtained via discovery.

```

import cgi
import openid

import common

from openid.consumer import consumer
from openid.extensions import ax

'''
' Function: Main
' Description: Initiates the OpenID authentication process
'''
def main():
    #get query parameters
    params = cgi.FieldStorage()

    #check if an OpenID url was specified
    if not params.has_key('openid_url'):
        print_msg('Please enter an OpenID Identifier to verify.', 'text/plain')
    else:
        #capture OpenID url
        openid_url = params['openid_url'].value

        #create a base consumer object
        oidconsumer = consumer.Consumer({}, None)

        try:
            request = oidconsumer.begin(openid_url)
        except:
            print_msg('Error in discovery: ' + openid_url, 'text/plain')
        else:
            if request is None:
                print_msg('No OpenID services found', 'text/plain')
            else:

```

First, we import the libraries and files that we will need for this script. Besides the standard library imports, we also include our previously mentioned *common.py* file and the OpenID consumer and Attribute Exchange extension libraries.

To start building out our `main()` function, we first get the query parameters that were entered into the form, which in this case are simply the OpenID URI. We then check if that URI exists. If it doesn't exist, we print out an appropriate failure message. If it does exist, we capture the value.

We then create a new base OpenID consumer object that will give us the functionality required to perform discovery. With this object, we call the `begin(...)` method, passing in the provided OpenID URI to begin discovery. If there was an error, we print out an appropriate message. If the process was successful, we make sure that the request object is not `None`. If it is, we print out the appropriate message. If it's not, we continue with the next step, attaching any desired OpenID extensions and the OAuth parameters needed for the hybrid auth process.

**Attaching extensions and OAuth hybrid parameters.** Now we'll attach everything that we need for the OpenID request and OAuth hybrid pieces of the process.

```
#attribute exchange extension request
ax_request = ax.FetchRequest()
ax_request.add(ax.AttrInfo('http://axschema.org/contact/email',
    required=False, alias='email'))
ax_request.add(ax.AttrInfo('http://axschema.org/media/image/default',
    required=False, alias='picture'))
request.addExtension(ax_request)

#add oauth hybrid extension parameters to redirect url
redirect_url = request.redirectURL(common.trust_root, common.return_to)
redirect_url += '&openid.ns.oauth=http%3A%2F%2Fspecs.openid.net%2F'
                extensions%2Foauth%2F1.0
                &openid.oauth.consumer=' + common.consumer_key

#print_msg(redirect_url, 'text/plain')
print "Location: " + redirect_url
```

We start with the OpenID Attribute Exchange extension. We create a new request object by calling `FetchRequest()`, which will hold all of the particular fields that we will be requesting. To that request object, we add a request to get the user's email address and default image (neither of which is required), just like we did in the Chapter 11 OpenID example. Once we've defined all our extensions, we add that Attribute Exchange object to the overall OpenID request object.



Even though using OAuth will give you access to a vast amount of user information—far more than what you could obtain with OpenID—using the Attribute Exchange extension can provide you with certain data that you can't access with OAuth or data that would require an extra HTTP request, depending on how the provider has set up its social API structures.

Next, we construct the redirect URL where we will be sending the user to accept the application permissions. We build this URI by making a call to `redirectURL(...)`, passing in our `trust_root` (the current absolute folder path to the application) and the `return_to` location (the callback file).

Now that we have the URL, we attach two additional query string parameters to signify that we want to obtain a preapproved request token at the end of the hybrid auth process:

`opened.ns.oauth`

The OpenID OAuth hybrid extension namespace URI. This should be a URL-encoded version of the namespace: *http://specs.openid.net/extensions/oauth/1.0*.

`opened.oauth.consumer`

The consumer key issued when we set up our application with the provider. This will allow the provider to attach the appropriate application to the auth request.

With those parameters attached to the redirect URL, we redirect the user to that provider location to proceed through the hybrid auth process.

Before we move on to the next file, the callback location, there are just a few helper and initialization aspects at the bottom of this script to go over.

**Helpful function and initialization.** There are a few last pieces of this file that we should talk about: the `print_msg` function and the program initialization functionality.

```
'''
' Function: Print Message
' Description: Print a message with a provided content type
' Inputs: msg (string) - The message to be displayed
'         type (string) - The content type to use (e.g. text/plain)
'''
def print_msg(msg, type):
    if msg is not None:
        print 'Content-Type: %s' % (type)
        print ''
        print msg

#initiate load of main()
if __name__ == '__main__':
    main()
```

The `print_msg(...)` function accepts a string message and a string content type, which it will use to print out a message to the user.

At the bottom of the file, we simply initiate the loading of `main()` to start the program execution.

At this point, the user should be going through the hybrid auth process, accepting the permissions of the application. Once he accepts them, he will be forwarded to our authentication callback file, *complete.py*.

### The authentication callback

The authentication callback, stored in this example as *complete.py*, is where the user is forwarded once he has granted the application permission to access his personal information. The crux of this content is exchanging the preapproved request token that will be passed back from the hybrid auth process for an access token that we can use to obtain the user's private information, much like we saw back in Chapter 9.

More specifically, this file will process the following steps:

1. Extracting all passed parameters from the auth process.
2. Completing the OpenID process to capture the OpenID display identifier for the user as well as all attributes from the Attribute Exchange extension.
3. Exchanging the preapproved request token for an OAuth access token.
4. Using the access token to make a private data request to get the current user's profile information.

Let's break down this file into several different sections to explore how we will accomplish each task, starting with extracting all passed data and creating a new OpenID consumer object in order to complete the OpenID process.

**Capturing response objects and preparing the OpenID consumer request object.** At the beginning of the file, we capture and prepare the information that will be used throughout the rest of the hybrid auth complete script.

At the top, we include several standard libraries as well as the OAuth library, OpenID consumer and extension libraries, and the common variables needed to run the program. We also add the `dotdict` class at the top of the file. This class will provide dot-notation functionality within a dictionary object, which the OAuth library will use during the token exchange part of the process. Basically, `dotdict` allows you to not only refer to an item in the dictionary like:

```
token['item1']
```

but also using the dot-notation method:

```
token.item1
```

With that done, we then jump into the `main()` function and begin to capture all required values passed to the form and create any required consumer objects.

```
import cgi
import openid
import urllib
import os
import oauth.oauth as oauth

import common

from openid.consumer import consumer
from openid.extensions import ax

'''
' Class: Dot Notation Insertion
' Description: Adds dot notation capabilities to a dictionary
'''
class dotdict(dict):
    def __getattr__(self, attr):
        return self.get(attr, None)
    __setattr__ = dict.__setitem__
    __delattr__ = dict.__delitem__

'''
' Function: Main
' Description: Completes OpenID authentication process and prints results
'''
def main():
    #create a base consumer object
    oidconsumer = consumer.Consumer({}, None)

    #print page content type
```

```

print 'Content-Type: text/plain'
print ''

#parse query string parameters into dictionary
params = {}
string_split = [s for s in os.environ['QUERY_STRING'].split('&') if s]
for item in string_split:
    key,value = item.split('=')
    params[key] = urllib.unquote(value)

```

We start the `main()` function by creating a new base OpenID consumer object. This will allow us to complete the OpenID process by providing us with a response stating whether it was successful or not.

Next, we print out the page content-type headers for displaying debugging information (basically, variable dumps) back to the person running the script.

We then loop through all query string parameters (split on the ampersand) and split the strings again on the equals sign to get the key and value. We store the results in a dictionary object.

We can then move on to completing the OpenID process and extracting the user's OpenID display identifier and the Attribute Exchange extension values.

**Completing the OpenID process and extracting the data.** Using the variables that we created in the last chunk of code, we can complete the OpenID process and extract any information returned.

```

#complete OpenID authentication and get identifier
info = oidconsumer.complete(params, common.return_to)
display_identifier = info.getDisplayIdentifier()

#build attribute exchange response object
ax_response = ax.FetchResponse.fromSuccessResponse(info)
if ax_response:
    ax_items = {
        'email': ax_response.get('http://axschema.org/contact/email'),
        'picture': ax_response.get('http://axschema.org/media/image/default')
    }

    #print attribute exchange object
    print 'Attribute Exchange Response Object:'
    print ax_items

#print openid display identifier
print '\n\nOpenID Display Identifier: \n' + display_identifier

```

We start by using the OpenID consumer object that we created and call the `complete(...)` function on it, passing in the query string parameters dictionary object that we created and the `return_to` URL (this file location) from the `common.py` file. This should provide us with the return object full of information about the user.

From the returned object, we call `getDisplayIdentifier()` to extract the unique OpenID display identifier URI for the user.



We then start pulling out the data from the Attribute Exchange extension request. We fetch the response object by calling `FetchResponse.fromSuccessResponse(...)` within the `ax` library, passing in the returned OpenID object. This will provide us with the required set of return elements for Attribute Exchange. If that return object exists, we create a new dictionary object, `ax_items`, composed of the email and picture that we requested.

Last, we print out the Attribute Exchange object and display identifier. This is basically just a debugging step so that when you first set up the example, you can assess whether the data was returned correctly.

We can now focus on determining whether the status was returned correctly and exchanging the preapproved request token from the OpenID OAuth extension parameters for a valid OAuth access token to capture user information.

**Checking the OpenID status and obtaining the access token.** With the extracted OpenID data, we can check the OpenID status response and then begin the token exchange process for OAuth.

```
#check the openid return status
if info.status == consumer.FAILURE and display_identifier:
    message = "\n\nOpenID Response:\nVerification failed"
elif info.status == consumer.CANCEL:
    message = '\n\nOpenID Response:\nVerification cancelled'
elif info.status == consumer.SETUP_NEEDED:
    message = '\n\nOpenID Response:\nSetup needed'
elif info.status == consumer.SUCCESS:
    message = '\n\nOpenID Response:\nSuccess'

#build base consumer object with oauth keys and sign using HMAC-SHA1
base_consumer = oauth.OAuthConsumer(common.consumer_key,
    common.consumer_secret)
signature_method_hmac_sha1 = oauth.OAuthSignatureMethod_HMAC_SHA1()

#build dictionary of pre-approved request token and blank secret to
#exchange for access token
req_token = dotdict({'key': params['openid.oauth.request_token'],
    'secret': ''})

#create new oauth request and sign using HMAC-SHA1 to access token endpoint
#to exchange request token for access token
oauth_request = oauth.OAuthRequest.from_consumer_and_token(base_consumer,
    token=req_token, verifier=None,
    http_url=common.oauth_access_token_endpoint)
oauth_request.sign_request(signature_method_hmac_sha1, base_consumer,
    req_token)

#make request to exchange request token for access token string
token_read = urllib.urlopen(oauth_request.to_url())
token_string = token_read.read()

#parse access token string into parameters and extract user guid
token_params = cgi.parse_qs(token_string)
```

```

guid = token_params['xoauth_yahoo_guid'][0]

#create new access token object to make permissioned requests
access_token = oauth.OAuthToken.from_string(token_string)

```

First, we check the status of the OpenID exchange object. Possible values are:

`consumer.FAILURE`

The authentication process failed at some point.

`consumer.SUCCESS`

The authentication process succeeded. At this point, we should begin processing all of the responses.

`consumer.CANCEL`

The authentication process was cancelled and not completed.

`consumer.SETUP_NEEDED`

Additional setup is required in the authentication request.

*Any other response*

If we encounter any other response, we should gracefully handle this as a “Something went wrong” case.

`SUCCESS` is the response that we’ll assume here in order to continue with our token exchange process. We call `OAuthConsumer(...)` within the `oauth` library, passing in the consumer key and secret for our OAuth application from the `common.py` file. We then create a new HMAC-SHA1 signature object by calling `OAuthSignatureMethod_HMAC_SHA1()` within the `oauth` library so that we can sign the requests that we will be making.

Now we need to go through the process of exchanging the preapproved request token for an access token, but we only have the request token key that was passed through the query string parameters to this file. To get the request token object, we create a dictionary object comprising a `key` parameter with the value of the preapproved request token, and a `secret` parameter with a blank value (since it isn’t needed).

Next, we create the token exchange request. To accomplish this, we make a call to `OAuthRequest.from_consumer_and_token(...)` within the `oauth` library, passing in the base consumer object, request token object, a blank value for the verifier, and the access token endpoint on the provider where the switch will occur. We then sign the request by calling `sign_request(...)`, passing in the signature method object (HMAC-SHA1), base consumer object, and the request token object.



The `oauth_verifier` is not required for the request object because we have a preapproved request token from the hybrid auth process.

We can then make the request to do the token exchange. We open the OAuth request URL, which we obtain by calling `to_url()` within the `oauth_request` object, and then read the contents returned into a variable string.

Next, since the token string that is returned mimics query string parameters (`item1&item2&...`), we call `cgi.parse_qs(...)` to create a dictionary out of the parameters returned. From the parameters, we can then extract the globally unique identifier (GUID) of the user that Yahoo! returned in the object.



The primary reason for creating an object out of the parameters is to extract any extra data that the service might attach in the token string that we can use, such as user identifiers. A simple way to check for additional parameters in a service is to examine the token string returned from the request token/access token exchange process.

Now we just need to construct an access token object out of the string that was returned from the token exchange process. To accomplish this, we call `OAuthToken.from_string(...)` in the `oauth` library and pass in the token string.

Now that we have the access token object in hand, we can begin making authenticated requests to the provider to obtain protected user resources.

**Making signed requests for protected user resources.** The process for making a signed request using the access token is fairly straightforward. Just as we did with the request token/access token exchange, we simply need to create a request object, sign it, and then use that to make the requests.

```
#create url to Yahoo! servers to access user profile
url = 'http://%s/v1/user/%s/profile' % ('social.yahooapis.com', guid)

#create new oauth request and sign using HMAC-SHA1 to get profile of user
oauth_request = oauth.OAuthRequest.from_consumer_and_token(base_consumer, token=access_token, http_method='GET', url=url)
oauth_request.sign_request(signature_method_hmac_sha1, base_consumer, access_token)

#make request to get profile of user
profile_read = urllib.urlopen(oauth_request.to_url())
profile_string = profile_read.read()

#print profile response object
message += '\n\nProfile Response Object:\n' + profile_string
else:
    message = '\n\nOpenID Response:\nVerification failed.'
print message

if __name__ == '__main__':
    main()
```

We first construct the URL to which we want to make requests to obtain privileged user data from the provider site. In this case, the URL used is for extracting the current user's full profile.

The next step is to construct an OAuth request object by calling `OAuthRequest.from_consumer_and_token(...)`, passing in the base consumer object, access token object, HTTP method (in this case, GET), and the URL to which we want to make a request. We then call `sign_request(...)` to sign the request, passing in the HMAC-SHA1 signature object, base consumer, and access token.

Next, we open the resource URL by calling `oauth_request.to_url()` and store the response string as a separate string. This is the response string from the request and should contain the user's full profile.

Finally, we simply print out the resource object so that we can explore the string that was returned to us.

## Conclusion

Now that we've concluded our exploration of OpenID, OAuth, and the hybrid auth extension, we can clearly see how combining specifications delivers both the rich capabilities of an authentication sign-in model and the extensive amount of data and viral channels that are available through the authorization standards.

In this chapter, we have identified the questions you should ask yourself before implementing the hybrid auth extension and delved into practical implementation examples to see how these two standards are combined.

We have now experienced firsthand the extensive capabilities that are available in both specifications, and discovered the potential they offer us for creating efficient and powerful login and personalization systems.

---

# Web Development Core Concepts

Throughout this book, we have talked about many open source and web service standards. This appendix will explore the core concepts behind much of the technology, standards, and services that we have worked with in each chapter.

## A Brief Tour of Open Source Standards

Most major Internet-based companies implement some measure of open source standards. Whether it's authentication and authorization technologies such as OpenID and OAuth, or the simple metadata tagging of the Open Graph protocol to integrate a Facebook Like button, the industry has embraced the value of open source standards and what they offer to its businesses.

In the sections that follow, we'll try to answer a few of the most common questions that arise on the topic of open source standards.

### What Are the Benefits and Drawbacks of Using Open Source Standards?

*Why should we even consider using open source standards?* This is a question that usually comes up when a company first begins embarking along the open source path, and it's one of the most valid to ask before starting integrations.

Let's look at some of the benefits and drawbacks of open standards to help implementers decide if it is the correct approach for them.

#### Benefits

The benefits are numerous, but we'll focus on just a few main points. Usually, open source initiatives that have any measure of success in the community have support from some major company. These companies often contribute heavily to the specifications and have a genuine business reason for doing so. In other words, these companies (e.g., Yahoo, Facebook, Google) are driving the future of the specification to fill some specific business need. This means that for the most part you, as an implementer, will be using

a fully vetted specification that is geared toward improving that particular business need.

In addition to major company contributions, open source specifications tend to have a large contributor network composed of people who either have a passion for the technology or have a specific reason for contributing. In either scenario, you will be drawing from a large development pool of engineers—who are often some of the best in their fields—without having to expend your own resources to develop a custom solution to the same problem. You will reap the rewards of a continually updating product that will go through regular release cycles. This means that the engineering time and effort required to implement the solution is drastically reduced, to the point where you'll simply need to integrate the technology into your web stack.

### **Drawbacks**

The main disadvantage to open source initiatives is the fact that the development of the project can be completely beyond the implementer's control. Even though an open source initiative may be in place to solve a specific problem, it tends to include a lot of additional features that different companies require for their particular implementations. While the projects generally attempt to maintain a platform-agnostic view, they still aim to address some specific concerns from many of the implementing companies. For an implementer who was not involved in the project construction, this means that the solution will not be custom built for his web stack and may contain feature bloat from added functionality he doesn't need for his particular integration.

## **Are Open Source Standards the Solution to Everything?**

The simple answer to this question is no. Open source is not a silver bullet that will solve all of your problems, but it's a good starting point.

The reason why open source standards are not the “end all, be all” solution is because they are built to solve a specific set of problems, and thus can sometimes have a very narrow focus in their implementation. Although the solution that is delivered by the open source initiative you're implementing may get you 90% of the way toward reaching your end goal, it's rare when you don't have to develop any tweaks or technology bridges to integrate the solution into your existing web stack and product.

You should think of open source as a great jumping-off point for your project, much like if someone came along and said, “Here, I have 90% of the work completed for the task you're working on.” Your main goal should be to see how that standard fits into your stack and integrate it.

Let's face it, when a company integrates an open source initiative, there is usually some core business reason behind the implementation, whether that can be directly monetized or not. For open source to be valuable for these companies, and thus reach the masses, these products need to make business sense. Open source initiatives rarely

discuss the business aspects of their implementations, so it's usually up to the implementing party to integrate that value.

## Web Service APIs

Web services that wish to make their data, tools, or features available for developers and clients tend to open up their system through *application programming interfaces* (APIs).

In the Web 2.0 world, APIs usually follow a RESTful architecture model, where the service will open up a series of URI endpoints to which developers make requests in order to get or set data from that web service system. Depending on the context of the data, these endpoints may be publicly available, or protected through authorization mechanisms such as OAuth.

In any case, these APIs communicate through the HTTP layer: the client will issue an HTTP request to the service, and the service will process the request and return standard response codes.

## HTTP Response Status Codes

When you're working with web services and the APIs of companies whose data you would like to use, it is important to understand the method these services use to communicate with you—*HTTP response status codes*.

When issuing requests to these services, you may receive any number of responses back in different address ranges. These codes will help you determine whether there were any major issues, if there was something expected by the service that was not present in the request, or if everything was sent and processed successfully.

In the response structure that a service sends back, you should be able to extract the HTTP response, which generally comprises the response code and a description that explains it.

There is a vast number of possible response codes, but the address ranges that will be returned will allow you to pinpoint any issues or decide whether you can process the response as a success case. These ranges include:

### *1xx range*

The request has been received and processing is in progress.

### *2xx range*

The request has been received, understood, accepted, and processed successfully.

### *3xx range*

The request has been received, but the client needs to take additional actions to complete the request.

4xx range

There was an error in the request sent by the client. The response from the server should indicate the specific issue encountered.

5xx range

The request was received and appears to be valid, but an error occurred while the server was attempting to process it.

Understanding common HTTP response status codes and approximate ranges will help you implement systems that can handle problems during processing, delivering a much less error-prone product.

## Understanding the Same-Origin Policy

The *same-origin policy* is important to keep in mind when you’re working with client-side scripting languages, such as JavaScript, and it is a core implementation in modern browsers. Its basic premise is that scripts that are running on a site should have access to that site’s properties and methods within reasonable limits, but those same scripts should not have access to properties and methods on other sites.

### How Is Origin Determined?

Browsers will compare two URLs to determine the origin based on a number of factors. These factors include the domain name, the application layer protocol, and in many browsers, the port used.

For example, say that we are making AJAX requests from a script on *http://www.site.com/page1.html* to access resources on other pages. Table A-1 outlines the results for several different URLs.

Table A-1. Results of making an AJAX request from a script on *http://www.site.com/page1.html*

URL requested	Outcome	Reason
<i>http://www.site.com/page2.html</i>	Success	Same host and protocol
<i>http://www.site.com:8888/page2.html</i>	Fail	Different port
<i>https://www.site.com/page2.html</i>	Fail	Different protocol (https)
<i>http://site.com/page1.html</i>	Fail	Different host (not an exact match)
<i>http://sub.site.com/page1.html</i>	Fail	Different host (subdomain)

Understanding these rules when you’re working with client-side scripting languages like JavaScript will save you a lot of headaches when development begins.



## Bypassing the Same-Origin Policy Requirements

When running into issues with the same-origin policy, developers implement some of the following methods to bypass these browser restrictions:

- Using a server-side proxy script. For instance, your JavaScript file can make a request to some PHP or Python script that it has access to, which in turn can facilitate server-to-server communication to some cross-domain source.
- Implement a Flash transport layer. In the same way that the server-side proxy works, some developers have used Flash to proxy requests to overcome the limits imposed by the same-origin policy.
- Generating iframes to pass data from a source site through to another location where the iframe is loaded.

There are numerous other ways to bypass the issue of communication between sites due to the same-origin policy restrictions, but these are a few of the most commonly implemented.

## REST Requests

*Representational Transfer* (REST) is a style of software architecture that defines a way for clients and servers to communicate using HTTP requests.

REST is a standard and widely implemented architecture that many companies employ to construct their web services so that developers can access their tools and data sources. REST offers a method—HTTP requests—for companies to embed the ability to create, update, retrieve, and delete resources within their systems.

There are a number of methods that are widely used within this architecture. Let's explore some of the ones that are implemented most often.

### GET Request

A GET request is one of the most straightforward request types. You are simply making a request to a URI endpoint and expecting some data to be returned to you.

In PHP, we can use a simple cURL request:

```
<?php
$url = 'http://www.example.com/request.php';

$ch = curl_init($url);
$options = array(
    CURLOPT_URL => $url,
    CURLOPT_RETURNTRANSFER => 1
);
curl_setopt_array($ch, $options);
$response = curl_exec($ch);
```

```
curl_close($ch);  
?>
```

Within the cURL option, we are specifying that we want a call a specific URL (CURLOPT\_URL) and would like to receive the response data from the request (CURLOPT\_RETURNTRANSFER).

In Python, we can simply use `urllib` from the standard library:

```
import urllib  
  
url = 'http://www.example.com/request.py'  
f = urllib.urlopen(url)  
response = f.read()
```

We open the specified URL and then read back the response from the request.

## POST Request

For the most part, HTTP POST requests are used to update existing resources on a server. You will be making a request to a service and passing along a POST payload, which the server then uses to denote the update resource or hold the content to be updated.

In PHP, we can use cURL to issue a POST request:

```
<?php  
$url = 'http://www.example.com/request.php';  
$postvals = 'firstName=John&lastName=Smith';  
  
$ch = curl_init($url);  
$options = array(  
    CURLOPT_CUSTOMREQUEST => 'POST',  
    CURLOPT_URL => $url,  
    CURLOPT_POST => 1,  
    CURLOPT_POSTFIELDS => $postvals,  
    CURLOPT_RETURNTRANSFER => 1  
);  
curl_setopt_array($ch, $options);  
$response = curl_exec($ch);  
curl_close($ch);  
?>
```

As with the GET request, we are making the call to a specified URL, but this time we are creating a POST string that we will send. In addition to the cURL options from the GET request, we use `CURLOPT_CUSTOMREQUEST` and `CURLOPT_POST` to state that we want to make a POST request and `CURLOPT_POSTFIELDS` to attach the data to send in the POST.

In Python, we can once again use `urllib` for the POST request:

```
import urllib  
  
url = 'http://www.example.com/request.py'  
postvals = {'first_name': 'John', 'last_name': 'Smith'}
```

```

params = urllib.urlencode(postvals)
f = urllib.urlopen(url, params)
response = f.read()

```

As with the GET request, we are making the request to a specific URL. This time, though, we create the object that we want to POST, encode it, and pass it along when we call `urlopen(...)`.

## PUT Request

Even though there are many web services that support only a subset of the RESTful architecture—GET and POST requests—fully *RESTful* services include PUT and DELETE requests as well.

PUT requests are a method for inserting new resources into a service. This request type is very similar to a POST request in technical implementation, and the two requests are often confused with each other.

In PHP, we will once again use a cURL request with some options set:

```

<?php
$url = 'http://www.example.com/request.php';
$postvals = 'firstName=John&lastName=Smith';

$ch = curl_init($url);
$options = array(
    CURLOPT_CUSTOMREQUEST => 'PUT',
    CURLOPT_URL => $url,
    CURLOPT_POST => 1,
    CURLOPT_POSTFIELDS => $postvals,
    CURLOPT_HTTPHEADER => array('Content-Length: ' . strlen($postvals)),
    CURLOPT_RETURNTRANSFER => 1
);
curl_setopt_array($ch, $options);
$response = curl_exec($ch);
curl_close($ch);
?>

```

This request mimics that of the POST request, except that we set a different `CURLOPT_CUSTOMREQUEST` value (PUT). In this example, we can also set an HTTP header with the content length of the POSTed fields if required.

In Python, this request is a little trickier because there are a few quirks to making requests with `urllib2`:

```

import urllib
import urllib2

url = 'http://www.example.com/request.py'
postvals = {'first_name': 'John', 'last_name': 'Smith'}
params = urllib.urlencode(postvals)

opener = urllib2.build_opener(urllib2.HTTPHandler)

```

```

request = urllib2.Request(url, data=params)
request.add_header('Content-Type', 'application/json')
request.get_method = lambda: 'PUT'
f = opener.open(request)
response = f.read()

```

The following code in the `urllib2` standard library helps us determine which HTTP method to use in a request:

```

def get_method(self):
    if self.has_data():
        return 'POST'
    else:
        return 'GET'

```

As you can see, it doesn't exactly support any types other than GET and POST, so we need to make a few adjustments. We first generate our URL that will be called, and then build and encode the parameters to send with the PUT request. We then create our HTTP handler opener and generate the request to our URL with the parameter object through the `urllib2.Request(...)` request. Then, we add a `Content-Type` header for the request. Now we need to override the method that `urllib2` will use in the request by manually setting the `get_method` to PUT. Finally, we open the URL location from the request object and read back the response.

## DELETE Request

As you may have guessed, an HTTP DELETE request is used to remove resources from some web service.

Since cURL is the de facto method for making HTTP requests in PHP, we will once again use it for our DELETE request:

```

<?php
$url = 'http://www.example.com/request.php';
$postvals = 'firstName=John&lastName=Smith';

$ch = curl_init($url);
$options = array(
    CURLOPT_CUSTOMREQUEST => 'DELETE',
    CURLOPT_URL => $url,
    CURLOPT_POST => 1,
    CURLOPT_POSTFIELDS => $postvals,
    CURLOPT_FOLLOWLOCATION => 1,
    CURLOPT_HEADER => 0,
    CURLOPT_RETURNTRANSFER => 1
);
curl_setopt_array($ch, $options);
$response = curl_exec($ch);
curl_close($ch);
?>

```

In this case, we will send along some POST values to denote the resource to be deleted, so this request will very much mimic a PUT or POST. We simply set the `CURLOPT_CUSTOMREQUEST` to `DELETE`.

As with our PUT request, in Python we will use `urllib2`:

```
import urllib
import urllib2

url = 'http://www.example.com/request.py?id=1234'

request = urllib2.Request(url)
request.get_method = lambda : 'DELETE'

f = urllib2.urlopen(request)
response = f.read()
```

Our Python request will follow a pattern similar to our PUT request—we will need to manually set the `get_method`, this time to `DELETE`. We then make our request with `urlopen(...)` and read back the server response.

## HEAD Request

The last request type that we will look at is the HTTP HEAD request. Think of this one like an HTTP GET request but without any response body returned. The value in making this request is in ensuring that a resource is available prior to using it.

For instance, let's say we scrape the content of a site and retrieve a series of image links, some of which have relative URLs in different formats, and others that are absolute URLs to the specific resource. We perform some logic to generate absolute URLs for the relative links so that we can use them from any site. We can then use HEAD requests to verify that the URLs that we have generated are even valid and will return a resource prior to implementing them on a page. Since these requests don't return back the actual image, we can perform this action quickly as opposed to having to obtain the resource from a GET or simply integrating the image on a page without first ensuring that the link is valid.

A HEAD request in PHP is a simple process and mimics the corresponding GET request in many ways:

```
<?php
$url = 'http://www.example.com/image.jpg';

$ch = curl_init($url);
$options = array(
    CURLOPT_URL => $url,
    CURLOPT_HEADER => 1,
    CURLOPT_NOBODY => 1,
    CURLOPT_RETURNTRANSFER => 1
);
curl_setopt_array($ch, $options);
$response = curl_exec($ch);
```

```
curl_close($ch);  
?>
```

The important cURL parameter to note here is `CURLOPT_NOBODY`. This parameter ensures that the request being made is a HEAD request because no body content will be returned.

The Python example looks like that of the GET request as well. For this instance, we will use the `urllib2` library instead of `urllib`:

```
import urllib2  
  
request = urllib2.Request('http://www.example.com/image.jpg')  
request.get_method = lambda : 'HEAD'  
  
f = urllib2.urlopen(request)  
response = f.info().gettype()
```

Much like we did for the PUT and DELETE requests, we simply need to override the `get_method` parameter prior to making our request. When obtaining the resource, instead of calling the `read()` method to get the response, we can simply call `info().gettype()` to obtain the content type of the resource being requested. If the content type is what is expected, then the resource exists.

## Microformats and the Semantic Web

A *microformat* is a web-based method for implementing page semantics through the use of common HTML or XHTML tags and the attributes within those tags (e.g., a `<span>` tag with an appropriate class). These tags convey metadata about the page or a particular object on the page.

For instance, with the defined *geo* microformat specification, we can use `<span>` tags with an appropriate class, `geo`, to denote that we are specifying a geographical location.

```
<span class="geo">  
  <span class="latitude">45.512280</span>,  
  <span class="longitude">-73.554390</span>  
</span>
```

Semantic markup is a core concept of technologies such as those we explored in the Open Graph protocol in Chapter 10. Microformats are simply a way of contributing to the semantic web.

This type of semantic markup helps when you are attempting to programmatically parse a site to extract as much usable information about its content as possible. If that page has all of its relevant information tagged through microformat specifications, then extracting its relevant data should be a simple task.

In addition to providing a way to extract data easily, tagging through microformat specifications allows the site developer to denote the relevant content of the page rather

than having a site parser attempt to infer that information itself. This will ensure a consistent experience when the content is integrated at another location.

There are a number of microformat specifications available to help you implement microformat tagging on your sites:

*hAtom*

To mark up Atom feeds within standard HTML.

*hCalendar*

To tag event-based information.

*hCard*

For contact information. This includes:

*adr*

For address-based information.

*geo*

For geographical coordinates such as latitude and longitude.

*hMedia*

To mark up audio and video content.

*hNews*

To denote news-based content on a page.

*hProduct*

For products.

*hRecipe*

For recipes and information relating to foodstuffs.

*hResume*

For resumes or CVs.

*hReview*

For any type of review content.

*rel-directory*

For distributed directory creation and inclusion.

*rel-enclosure*

For multimedia attachments to web pages.

*rel-license*

For the specification of a copyright license.

*rel-nofollow*

An attempt to discourage third-party content spam.

*rel-tag*

For decentralized tagging.

*xFolk*

For tagged links.

*XHTML Friends Network (XFN)*

For social-based relationship data.

*XOXO*

For lists and outlines.

Following these specifications will help you build semantically relevant pages and sites. For more information on microformat specifications, see <http://microformats.org/>.

## Installing Subversion (SVN)

Throughout this book, we have used Subversion to obtain the most up-to-date code repositories to build projects or work with new features. Having it installed will make your life a lot easier as you continue working through and extending these examples.

The following two sections describe the methods for obtaining and installing Subversion on Mac OS X and Windows, respectively.

### Installing on Mac OS X

We will use MacPorts in order to install Subversion on our systems. If you don't have MacPorts installed, you can obtain the *.dmg* install file from its project install page at <http://www.macports.org/install.php>.

The first thing we will do is to update MacPorts if needed. Open a terminal window and type in the following command:

```
sudo port -v selfupdate
```

Once MacPorts has updated, we will install the three Subversion dependencies. From the terminal window, enter:

```
sudo port install sqlite3
sudo port install apr-util
sudo port install neon
```

When those have finished installing, you can install Subversion with the following command:

```
sudo port install subversion
```

You are now ready to begin making Subversion requests from the terminal window.

### Installing on Windows

Installing Subversion on Windows is just a matter of following these few steps:

1. Go to the Apache Subversion project packages page at <http://subversion.apache.org/packages.html> and scroll to the bottom for the Windows packages.



2. For this installation, we will use the Win32Svn package. Click that option and then, on the page that loads, click the Download button to obtain the *.msi* installer file.
3. Once the *.msi* file has downloaded, double-click it to begin installation. You may go through the installation with the default choices.

Once the installation has completed, you will be able to run SVN commands from the Windows command prompt.

## Installing Apache HTTP Server

One helpful utility to have installed on your system is the Apache HTTP server. This will allow you to run scripts locally on your computer without requiring a web-based server environment.

In the initial chapters of this book, when we set up Apache Shindig and Partuza for OpenSocial development, we use a local HTTP server installation to run all of our scripts.

### Installing on Mac OS X

If you're using a Mac, Apache is already installed on your system; it's just probably not enabled. Since this is the typical case, we'll just go through the steps of enabling it rather than installing a whole new instance.

First, go to System Preferences on your Mac. Under Internet & Network, you'll see an option for Sharing, as shown in Figure A-1. Click this option.

In the Sharing panel that opens, simply ensure that the Web Sharing option is selected (Figure A-2) to enable the Apache HTTP server.

Now, if you go to *http://localhost* in a browser, you should see a default page for your server. The Apache HTTP server should load files from the default document location on your Mac, */Library/WebServer/Documents*.

### Installing on Windows

The Apache HTTP server does not come preinstalled on Windows systems, so we need to go through a few steps to download and install it:

1. In a browser, go to the Apache HTTP Server download page at *http://httpd.apache.org/download.cgi* and select the current stable release from the list.
2. From the list of available packages, pick the Win32 binary download link (this example uses the "Win32 Binary including OpenSSL 0.9.8o" version).



Figure A-1. The Sharing option under System Preferences in Mac OS X

3. Once you've downloaded the package, double-click the `.msi` file to begin the installation. You can either go through the installation process with the default choices or modify paths and attributes as needed.
4. Once you've installed the package, go to the HTTP server installation directory and then into the `bin` folder. Double-click the `httpd.exe` file to start the HTTP server.

You will now be able to go to `http://localhost` in a browser to view the default page for your server. Files for the server environment will be available in the `/htdocs` folder within the Apache HTTP server install directory.

## Setting Up Your PHP Environment

Now that we have our Apache HTTP server environment set up, we can begin obtaining and setting up PHP so that we can run files with the `.php` extension through `http://localhost`.

## Installing on Mac OS X

As with Apache HTTP server, PHP comes preloaded on a Mac. We simply need to go through a few steps to enable it and integrate it with the Apache HTTP server environment.

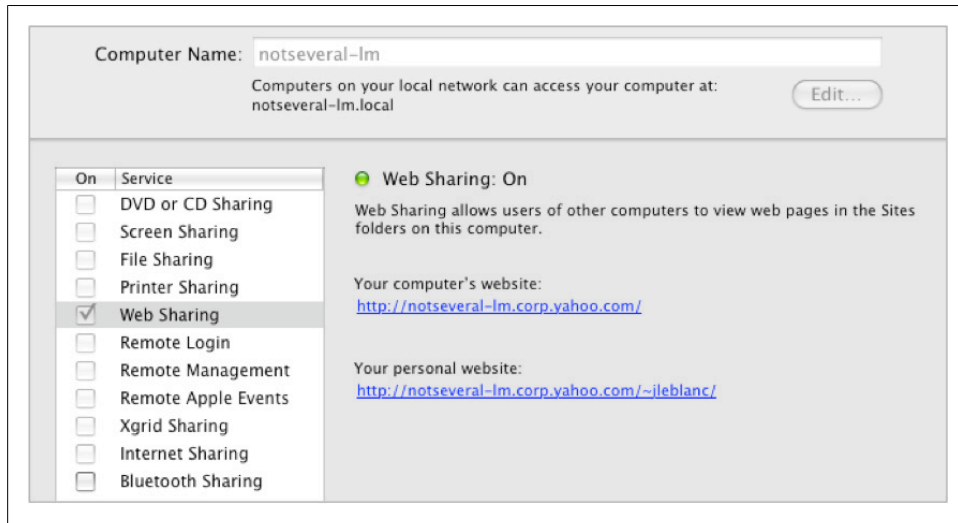


Figure A-2. Web Sharing option under Sharing settings on Mac OS X

First, we need to modify the HTTP server configuration file. The default location of the server should be `/private/etc/apache2`. We'll head to that location and then edit the `httpd.conf` file using the following command:

```
cd /private/etc/apache2
sudo vi httpd.conf
```

When you've opened the `httpd.conf` file, search for the following line:

```
#LoadModule php5_module libexec/apache2/libphp5.so
```

This is the line that allows PHP 5 to run within the HTTP server. To enable it, remove the initial pound sign (`#`), giving you:

```
LoadModule php5_module libexec/apache2/libphp5.so
```

Save and exit the `httpd.conf` file. Now we're going to move on to enabling PHP by setting up our `php.ini` configuration file. We'll go to our `/private/etc` folder and rename the default configuration file as follows:

```
cd /private/etc
sudo cp php.ini.default php.ini
```

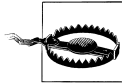
Now we simply need to restart the Apache HTTP server, which we can do by going to Web Sharing within System Preferences and disabling and reenabling the server.

Once we've done this, we should be able to use PHP files by navigating to them in a browser at `http://localhost/~username/file.php`.

## Installing on Windows

Once the Apache HTTP server has been installed, we can download PHP and make it available for the environment. Simply follow these steps to be able to run files from *http://localhost*:

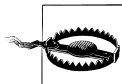
1. To download the source files for PHP, go to *http://windows.php.net/download/* and click the Zip option under the Thread Safe versions for the latest stable release.



Make sure that you download the Thread Safe version for PHP; otherwise, the required *.dll* files for Apache will not be present.

2. Once the files are downloaded, unzip them to *C:\php*.
3. Go to the *C:\php* directory and rename *php.ini-production* to *php.ini*. This will be the configuration file for PHP.
4. Now go to your Apache HTTP server installation directory and into the */conf* folder. Open *httpd.conf* for editing. This is the configuration file for the server.
5. Add the two following lines where applicable in the file (with the other *LoadModule* lines):

```
LoadModule php5_module "c:/php/php5apache2_2.dll"  
AddHandler application/x-httpd-php .php
```



Make sure that the appropriate *php5apache2\_2.dll* file is present within the *C:\php* directory before adding it. There may be permutations on the filename. The appropriate file should be a *.dll* that starts with *php5apache2*.

6. Restart your Apache server.

You will now be able to run *.php* files from the Apache HTTP server. For instance, if you create *test.php* in the HTTP server */htdocs* directory, you will be able to load it at *http://localhost/test.php* from your browser.

## Setting Up Your Python Environment

When it comes to hosting and running projects quickly, the Google App Engine environment is an excellent tool. For the most part, the Python examples in this text are all run from App Engine, using a simple YAML file for configuration.

To install App Engine:

1. Go to *http://code.google.com/appengine/* and sign up for an App Engine account.

2. Head to the download page and select your appropriate system download to install the App Engine SDK: <http://code.google.com/appengine/downloads.html>. This will be the environment in which you can run Python programs off localhost.

Using the SDK, you can simply load a YAML configuration file for the project, start the project, and then head to localhost with the appropriate project port in a browser window.

If you would like to run Python via a terminal window, you can install it simply by heading to the Python project download page at <http://www.python.org/download/>, picking the appropriate installer for your system, and then going through the installation steps from the installer.

Another option for working in a terminal window is ActivePython, available at <http://www.activestate.com/activepython>. The community edition is free and has a number of advantages over the standard download from <http://www.python.org>.



---

# Glossary

## activity

An activity is a piece of content (e.g., text, image, video) that a user shares through a consumable stream of activities. This stream may consist of the activities of a single user or an amalgamation of many people, sorted by criteria such as friendships or location (for example, a user's Facebook news feed contains a series of activities from her friends).

## AJAX

AJAX is an acronym for Asynchronous JavaScript and XML, although many implementations favor JSON over XML. It defines a series of interrelated web development technologies that allow developers to construct dynamic web applications.

More information: [http://en.wikipedia.org/wiki/Ajax\\_\(programming\)](http://en.wikipedia.org/wiki/Ajax_(programming)).

## connection

In terms of a social profile, a connection is a reciprocated relationship between two individuals. Both parties have to accept this “friendship” link to build a connection. Being part of a connection usually gives both individuals additional privileges to view and consume more information about each other from their respective profiles.

## container

*Container* is the term used to describe a social networking site that allows developers to build applications on top of it. Well-known social networking containers are Facebook, YAP, iGoogle, and Orkut.

## distributed web framework

Distributed web frameworks, as used in this text, refer to a series of open protocols and specifications that promote the syndication of content and entity cross-communication on the Web.

## gadget

In the context of OpenSocial, all applications that follow the gadget XML specification are considered gadgets.

## headless request

Headless requests are processes that run without requiring user interaction. In the case of services such as OAuth, the term *headless* refers to a process in which a user authorizes the application once and then the service, in subsequent requests, makes changes to or performs some action on the user's data without him being involved. The service does this by storing his access token and using it to make additional requests for the duration of time that the token is valid.

## IRI

An *Internationalized Resource Identifier* is a generalization of a URI. Instead of being limited to the use of the ASCII character set, an IRI may contain characters from the universal character set (Unicode/ISO 10646 ([http://en.wikipedia.org/wiki/ISO\\_10646](http://en.wikipedia.org/wiki/ISO_10646))).

More information: [http://en.wikipedia.org/wiki/Internationalized\\_Resource\\_Identifier](http://en.wikipedia.org/wiki/Internationalized_Resource_Identifier).



## LRDD

*Link-based Resource Descriptor Discovery* is an open protocol that defines methods for obtaining information about a resource through the use of a URI, much like how WebFinger is used to obtain information about a person through the use of her email address. The latest specification documentation for LRDD is available at <http://tools.ietf.org/html/draft-hammer-discovery-06>.

## owner

In the context of a social networking application, such as an OpenSocial gadget, the owner is the individual or company who created the application being used by a viewer.

## Partuza

Partuza is an example social networking site that is built off OpenSocial and Apache Shindig. The project home for Partuza is <http://code.google.com/p/partuza/>.

## REST/RESTful

*Representational State Transfer* is a specification that defines an HTTP communication architecture to which web-based services should conform. When a service implements the REST architecture, it is said to be *RESTful*.

More information: [http://en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer).

## RPC

A *remote procedural call* is a type of protocol that allows one computer to execute some program or script on another computer, even outside its own address space. This allows the client machine to leverage the existing programs on the server computer. The server will execute the requested script and return back the result of the execution to the client machine.

More information: [http://en.wikipedia.org/wiki/Remote\\_procedure\\_call](http://en.wikipedia.org/wiki/Remote_procedure_call).

## same-origin policy

Those working with client-side languages such as JavaScript should have an intimate

understanding of the same-origin policy. At a basic level, this policy states that scripts can access the methods and properties on their own domain but are restricted from accessing those methods and properties on other domains.

More information: [http://en.wikipedia.org/wiki/Same\\_origin\\_policy](http://en.wikipedia.org/wiki/Same_origin_policy).

## semantics/semantic web

The semantic web is essentially the utilization of specified metadata on a web page to allow machines to more effectively parse its content. Semantic markup allows for a much richer understanding of the page's content (i.e., what it is attempting to serve up to users) and helps construct a deep “web of data.”

More information: [http://en.wikipedia.org/wiki/Semantic\\_web](http://en.wikipedia.org/wiki/Semantic_web).

## Shindig

Shindig is an OpenSocial container that allows a developer or site owner to quickly begin hosting OpenSocial gadgets on his site. This is the de facto method for integrating OpenSocial gadgets in a service. Shindig is an Apache project whose home is located at <http://shindig.apache.org/>.

## social network

A social network is a site that allows users to engage with one another online and uses an extensive profile system to identify users within the system. One popular example is Facebook.

## social graph

*Social graph* is the term generally used to denote the links between two or more individuals within a *social network*. A user may have numerous interconnected social graphs that span many different social networks.

More information: [http://en.wikipedia.org/wiki/Social\\_graph](http://en.wikipedia.org/wiki/Social_graph).

## social entity

A social entity is a construct that resides within a user's *social graph*. Unlike the “per-





son” links that are traditionally associated with a social graph, an entity is a website, team, cause, or any number of other “things” that a user has indicated that she is a fan of or involved in. It is a generic container to denote additional information about a user other than friendship links.

#### viewer

In the context of a social networking application, such as an OpenSocial gadget, the viewer is the individual who is currently using the application constructed by the application *owner*.

#### XRDS

*eXtensible Resource Descriptor Sequence* is an XML format whose purpose is to provide metadata about some resource. The main use of this type of discovery, besides providing data about the resource, is to deliver services that are associated with the resource.

More information: <http://en.wikipedia.org/wiki/XRDS>.

#### XRI

*eXtensible Resource Identifier* is a scheme that defines a method for creating structured, self-describing identifiers that may be extended to many uses by including direct metadata about an object. The standard syntax and discovery format is domain, location, application, and transport independent, meaning that it can be shared among different domains, directories, and protocols.

More information: <http://en.wikipedia.org/wiki/XRI>.

#### viral

The term *viral* stems from application development, and generally means an application that maintains greater than a 1:1 growth. If each user of an application invites two people, then it will experience a viral growth trend, or in more popular parlance, “go viral.”

More information: [http://en.wikipedia.org/wiki/Viral\\_phenomenon](http://en.wikipedia.org/wiki/Viral_phenomenon).

#### YAML

YAML is an acronym for Yet Another Markup Language. Within the confines of Google App Engine, where YAML is referenced, it is used as an application configuration and control file.

More information: <http://en.wikipedia.org/wiki/YAML>.

#### YAP

YAP is an acronym for Yahoo! Application Platform. It is an OpenSocial 0.9-compliant container that showcases user-built applications across many of the Yahoo! sites and services.

More information: <http://developer.yahoo.com/yap>.