# Use PHP to build a Twitter-like system on your site

## It's simple to add Twitter-like microblogging to any application using PHP

Thomas Myer                                                              February 24, 2009

Learn how to use PHP to add a Twitter-like interface to your applications. Specifically, we show you how to allow users to add posts, disseminate those posts to other users who want to receive them, and allow users to choose to follow the posts of other users.

If you've been paying any attention at all, you know that Twitter is one of the biggest sensations in the Web 2.0 world. For those who don't know, Twitter (a service available at Twitter.com) is a simple microblogging service that allows users to make posts (called *tweets*) of up to 140 characters that answer the question "What are you doing now?" Users can follow people they find interesting and have followers of their own. In this way, information can be published to a small following or disseminated far and wide.

A quick glance at any single Twitter account reveals that users typically produce tweets on a wide variety of subject matter, from the everyday (for example, "I'm having a sandwich") to the more sublime. Often, there are embedded links to images, media files, and blog postings. These URLs are frequently obfuscated by services like TinyURL, mostly to keep the total character length of the post at or under 140 characters.

There are lots of folks who have taken to Twitter and have made an art form out of the super-condensed format, even having conversations with other users (by directing their remarks to @user, for example). From this simple start, a whole galaxy of Twitter-enabled mobile applications and other tools have sprung up. There are even awards now for funniest, most sublime, and most fact-filled tweets, plus online applications that track the state of the various Twitter applications out there.

Many other sites and services, such as LinkedIn and Facebook, now allow their users to update their current status in a decidedly Twitter-like way. In other words, updating your status on Facebook involves using a condensed message, and, of course, the status usually answers the question "What are you doing right now?"

Adding a microblogging or status update tool to your own site doesn't require a lot of work, and it provides your users with a fun and simple way to communicate. The goal of this article is to show you how to do just that. But first, I have to make a few assumptions about you.

First, I assume that you know something about PHP and MySQL. I also assume that you have access to some kind of locally available Apache Web server that runs PHP and MySQL. For the purposes of this article, I develop on a MacBook Pro using Macintosh, Apache, MySQL, and PHP (MAMP), a freeware program that conveniently provides an entire development environment in one package. However, you should be able to develop on Microsoft® Windows® or Linux® without any difficulty. Finally, I assume that you have an existing application running right now that involves users of some kind and that you're going to be adding microblogging or tweeting to that application in some way. For that reason, I skip over some of the more user-centric parts of the application (for example, logging in, managing a profile, etc.) in favor of the posts.

# Designing the back end of the application

At its simplest, the Twitter service centers around two nouns: users and messages. If you've already built an application and want to add a Twitter-like service to it, you probably already have user management in place. If you don't, you need to have some way of uniquely identifying each user in a database table (a primary key, usually an integer), a username (also unique), an e-mail address, a password, etc.

Tweets (or posts) are stored in a posts table, with each post having a primary key (some kind of sequential integer), a foreign key relationship back to the user that made the post, the post itself (limited to a number of characters), and a date/time stamp.

The final piece of the puzzle is a database table that shows which users are following whom. All that is needed is some way to record a user ID and the follower ID, giving your application the ability to quickly build lists of followers and easily disseminate information to those who have signed up to follow someone else.

If you're following along, go ahead and set up your three database tables now. Use the SQL code shown in Listing 1 to create the first table, called users. (If you already have a users table in place, please ignore this.)

## Listing 1. The users table

```
CREATE TABLE `users` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`username` VARCHAR( 255 ) NOT NULL ,
`email` VARCHAR( 255 ) NOT NULL ,
`password` VARCHAR( 8 ) NOT NULL ,
`status` ENUM( 'active', 'inactive' ) NOT NULL
) ENGINE = MYISAM ;
```

The second table, called posts, is shown below.

## Listing 2. The posts table

```
CREATE TABLE `posts` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`user_id` INT NOT NULL ,
`body` VARCHAR( 140 ) NOT NULL ,
`stamp` DATETIME NOT NULL
) ENGINE = MYISAM ;
```

Listing 3 shows the last table, called following. Note that this table has two primary keys.

## Listing 3. The following table

```
CREATE TABLE `following` (
`user_id` INT NOT NULL ,
`follower_id` INT NOT NULL ,
PRIMARY KEY ( `user_id` , `follower_id` )
) ENGINE = MYISAM ;
```

Before you go any further, make a file called header.php and place all your connection strings for MySQL in it. If you already have a file that handles this, ignore me for now. Just be sure to include this file everywhere because you'll need it. Listing 4 shows what this file might look like for you.

## Listing 4. Sample header.php file

```
$SERVER = 'localhost';
$USER = 'username';
$PASS = 'password';
$DATABASE = 'microblogger';


if (!($mylink = mysql_connect( $SERVER, $USER, $PASS))){
 echo  "<h3>Sorry, could not connect to database.</h3><br/>
 Please contact your system's admin for more help\n";
 exit;
}

mysql_select_db( $DATABASE );
```

Please note that you are also free to add any other kinds of security checks to this header.php file. For example, you could check to see if a user ID has been set in a session variable (showing that the user has actually logged in). If a user isn't logged in, you can redirect that user to a login page. This article doesn't get into all of that, but it's fairly easy to add when you need it.

# Creating the entry form

Now that you have the back-end tables set up, it's time to consider the PHP that will handle any data inserts and updates. What you need right now are some simple functions that will:

1. Allow users to log in and add posts.
2. Disseminate those posts to anyone following that user.
3. Allow users to follow other users.

I usually work within the context of a Model-View-Controller (MVC) application framework such as CodeIgniter because it provides me with a serious set of tools for creating these kinds of applications. For example, I normally start by creating two models (one for users and the other

for posts) that allow me to interact with the users, posts, and following tables, then move on from there.

Because you may already be working within a different framework, I decided against that approach here. Instead, I opt for a simpler approach that is not framework-specific. Also, just for today, I have you cheat a little bit and add a record to your users table to create a series of test users that you'll make available to your application. I create three users and give them usernames of `jane`, `tommy`, and `bill`.

When that's done, create a simple PHP file called functions.php that will contain your major functionality. You're going to create a handful of functions in this file that allow actions within the context of your microblogging application.

The first function, shown in Listing 5, is a simple one that allows you to add content to the posts table.

## Listing 5. Function for adding content to the posts table

```
function add_post($userid,$body){
 $sql = "insert into posts (user_id, body, stamp)
   values ($userid, '". mysql_real_escape_string($body). "',now())";

 $result = mysql_query($sql);
}
```

To test this simple function, you need to add two more PHP files to the mix. The first is the index.php file, which contains a basic little form for right now — you'll add a bit more to the page later. The second is the PHP file that the form posts to, called add.php. Listing 6 is the markup for the index.php file. Please note that you are using a PHP session to hard-code a user ID value of 1, which is the user `jane` in my database. This is perfectly OK to do now, but will obviously need to be changed later.

## Listing 6. Markup for the index.php file

```
<?php
session_start();
include_once('header.php');
include_once('functions.php');

$_SESSION['userid'] = 1;
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
 <meta http-equiv="content-type" content="text/html; charset=utf-8" />
 <title>Microblogging Application</title>
</head>
vbody>

<?php
if (isset($_SESSION['message'])){
 echo "<b>". $_SESSION['message']."</b>";
 unset($_SESSION['message']);
}
?>
<form method='post' action='add.php'>
```

```
<p>Your status:</p>
<textarea name='body' rows='5' cols='40' wrap=VIRTUAL></textarea>
<p><input type='submit' value='submit'/></p>
</form>

</body>
</html>
```

Also, note that I left some space just above the form for a status message, which you'll set dynamically in add.php, as shown below.

## Listing 7. Adding posts to the database with the add.php file

```php
<?php
session_start();
include_once("header.php");
include_once("functions.php");

$userid = $_SESSION['userid'];
$body = substr($_POST['body'],0,140);

add_post($userid,$body);
$_SESSION['message'] = "Your post has been added!";

header("Location:index.php");
?>
```

There shouldn't be anything particularly surprising about this code. It simply takes the field called `body` from the form and the user ID set in the PHP session and then passes them to the `add_post()` function in the functions.php file. Then another session variable is set (the update message), and the user is redirected back to the index.php page.

If you test this little function, the only way you know that it works is by checking the posts table in the database. That isn't exactly user-friendly, is it? What you need is for the posts to update right on your home page. For that, you need to add a second function to the functions.php file and use it on your home page.

# Adding a list of updates

It's time to open the functions.php file and add a second function to it. This time, call the function `show_posts()`. It will do just that, show all the posts for a particular user ID, as shown below.

## Listing 8. The `show_posts()` function

```
function show_posts($userid){
 $posts = array();

 $sql = "select body, stamp from posts
  where user_id = '$userid' order by stamp desc";
 $result = mysql_query($sql);

 while($data = mysql_fetch_object($result)){
  $posts[] = array(  'stamp' => $data->stamp,
      'userid' => $userid,
      'body' => $data->body
     );
 }
 return $posts;

}
```

If you pass this particular function a user ID, it returns all the posts made by that user in reverse-chronological order, all bundled in a nice, multidimensional array. To use it, all you have to do is call that function on index.php and retrieve all the posts for that user. Because you're dealing with a small amount of data for each record, this kind of query scales pretty well.

Listing 9 is the code you add to the index.php page, right after the form you put in before. By using the `show_posts()` function in combination with the session variable, you can grab all the posts by the logged-in user. If there are no posts, show an error message of some kind. If there are posts, show them one at a time in a table — or, if you want to get fancy, do your own Cascading Style Sheets (CSS) thing.

## Listing 9. Showing posts on the index.php page

```
<?php
$posts = show_posts($_SESSION['userid']);

if (count($posts)){
?>
<table border='1' cellspacing='0' cellpadding='5' width='500'>
<?php
foreach ($posts as $key => $list){
 echo "<tr valign='top'>\n";
 echo "<td>".$list['userid'] ."</td>\n";
 echo "<td>".$list['body'] ."<br/>\n";
 echo "<small>".$list['stamp'] ."</small></td>\n";
 echo "</tr>\n";
}
?>
</table>
<?php
}else{
?>
<p><b>You haven't posted anything yet!</b></p>
<?php
}
?>
```

Figure 1 shows the basic interface you've built so far — not bad for just a few minutes' work.

## Figure 1. The basic interface

Your status:

Submit

| 1 | another test, my friends<br>2009-01-05 22:15:05 |

The easy part is over. You now have a basic application that allows users to post their status and see that status displayed. However, there's one important part missing: There's no one out there to see your status updates besides you. In the next section, you create a simple interface that lists all users in the system and allows logged-in users to actually follow other users and see their status updates mixed in their own.

# Following other users

It's time to add more substance to the functions.php file. You need a `show_users()` function that gives you a list of all the users in the system. You'll use this function to populate a user list.

### Listing 10. The `show_users()` function

```php
function show_users(){
 $users = array();
 $sql = "select id, username from users where status='active' order by username";
 $result = mysql_query($sql);

 while ($data = mysql_fetch_object($result)){
  $users[$data->id] = $data->username;
 }
 return $users;
}
```

Now that you have the `show_users()` function, you can create a users.php file that runs it and displays a list of all the users in the system, each with a link that says **follow** next to the user name.

### Listing 11. A users.php file that runs the `show_users()` function

```php
<?php
session_start();
include_once("header.php");
include_once("functions.php");

?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
 <meta http-equiv="content-type" content="text/html; charset=utf-8" />
 <title>Microblogging Application - Users</title>
</head>
<body>

<h1>List of Users</h1>
<?php
$users = show_users();
```

```
if (count($users)){
?>
<table border='1' cellspacing='0' cellpadding='5' width='500'>
<?php
foreach ($users as $key => $value){
 echo "<tr valign='top'>\n";
 echo "<td>".$key ."</td>\n";
 echo "<td>".$value ." <small><a href='#'>follow</a></small></td>\n";
 echo "</tr>\n";
}
?>
</table>
<?php
}else{
?>
<p><b>There are no users in the system!</b></p>
<?php
}
?>
</body>
</html>
```

To access this list of users, add a link to users.php on the index.php file, right above the form:

```
<p><a href='users.php'>see list of users</a></p>
```

What you end up with is an easy-to-use table of user names, each with a **follow** link.

## Figure 2. List of users



Before moving on to the next phase, it makes sense to write a small function that tells you who the current user is already following. That way, users can use this list to determine if they want to follow or unfollow another user.

Go back to the functions.php file and add a function called `following()`, shown in Listing 12. You pass the current user ID to this function to get back every user ID that this user is following.

## Listing 12. The `following()` function

```
function following($userid){
 $users = array();

 $sql = "select distinct user_id from following
   where follower_id = '$userid'";
 $result = mysql_query($sql);

 while($data = mysql_fetch_object($result)){
  array_push($users, $data->user_id);

 }

 return $users;

}
```

You can now run this function on users.php and check to see if a particular user ID is in the array. If it is, use the unfollow link. If it isn't, then default to follow. Listing 13 shows the revised code.

## Listing 13. Revised users.php file, showing follow and unfollow links

```php
<?php
$users = show_users();
$following = following($_SESSION['userid']);

if (count($users)){
?>
<table border='1' cellspacing='0' cellpadding='5' width='500'>
<?php
foreach ($users as $key => $value){
 echo "<tr valign='top'>\n";
 echo "<td>".$key ."</td>\n";
 echo "<td>".$value;
 if (in_array($key,$following)){
  echo " <small>
  <a href='action.php?id=$key&do=unfollow'>unfollow</a>
  </small>";
 }else{
  echo " <small>
  <a href='action.php?id=$key&do=follow'>follow</a>
  </small>";
 }
 echo "</td>\n";
 echo "</tr>\n";
}
?>
```

The next step is simple: Create the action.php file used in the follow and unfollow links. This file accepts two GET parameters: one for the user ID and the other for follow or unfollow. As shown in Listing 14, this file is simple and short like the add.php file.

## Listing 14. The action.php file

```php
<?php
session_start();
include_once("header.php");
include_once("functions.php");

$id = $_GET['id'];
$do = $_GET['do'];

switch ($do){
 case "follow":
  follow_user($_SESSION['userid'],$id);
  $msg = "You have followed a user!";
 break;

 case "unfollow":
  unfollow_user($_SESSION['userid'],$id);
  $msg = "You have unfollowed a user!";
 break;

}
$_SESSION['message'] = $msg;

header("Location:index.php");
?>
```

As you can see, you take two very different actions — either `follow_user()` or `unfollow_user()`— depending on which link you selected before. You then set a message and redirect users back to the index.php page where they'll see not only their own messages but recent messages added by the users they follow. Or, in the case of an unfollow, that user's messages disappear from the listing. You need to add that last bit of code to index.php a little later. Right now, it's time to add the `follow_user()` and `unfollow_user()` functions to functions.php.

You have to be a bit careful with both of these functions. You can't just blindly follow or unfollow a user simply because someone clicks that link. First, you have to check if there's a relationship in the following table. If there is, then you can either ignore the request (in the case of the follow) or act on it (in the case of the unfollow). To simplify things, write a `check_count()` function you can use in either case, as shown below.

## Listing 15. The `check_count()` function

```
function check_count($first, $second){
 $sql = "select count(*) from following
   where user_id='$second' and follower_id='$first'";
 $result = mysql_query($sql);

 $row = mysql_fetch_row($result);
 return $row[0];

}

function follow_user($me,$them){
 $count = check_count($me,$them);

 if ($count == 0){
  $sql = "insert into following (user_id, follower_id)
    values ($them,$me)";

  $result = mysql_query($sql);
 }
}


function unfollow_user($me,$them){
 $count = check_count($me,$them);

 if ($count != 0){
  $sql = "delete from following
    where user_id='$them' and follower_id='$me'
    limit 1";

  $result = mysql_query($sql);
 }
}
```

The next step is easy: Display a list of other users the user is following on the home page. You already have a `show_users()` function, but that shows *all* users. You can easily repurpose this function by adding a nonrequired argument. This argument is a user ID that you use to limit the list of users to those who have that particular ID in the following position.

What's going on in the newly rewritten code shown in Listing 16 is a simple check on the incoming `$user_id` argument. If the user ID is greater than zero, use a query to pull out any user IDs

being followed by that ID. Use the `implode()` function to turn that array of values into a comma-separated list. Then insert that string — which looks something like `and id in (1,2,3...n)`— into the existing SQL query, thereby limiting the user listing to those the user is following.

## Listing 16. Rewritten code to limit the list of users pulled by the query

```php
function show_users($user_id=0){

 if ($user_id > 0){
  $follow = array();
  $fsql = "select user_id from following
    where follower_id='$user_id'";
  $fresult = mysql_query($fsql);

  while($f = mysql_fetch_object($fresult)){
   array_push($follow, $f->user_id);
  }

  if (count($follow)){
   $id_string = implode(',', $follow);
   $extra =  " and id in ($id_string)";

  }else{
   return array();
  }

 }

 $users = array();
 $sql = "select id, username from users
  where status='active'
  $extra order by username";


 $result = mysql_query($sql);

 while ($data = mysql_fetch_object($result)){
  $users[$data->id] = $data->username;
 }
 return $users;
}
```

Next, you add the code shown in Listing 17 to the home page to display all those followed users.

## Listing 17. Revising index.php to show users being followed

```php
<h2>Users you're following</h2>

<?php
$users = show_users($_SESSION['userid']);

if (count($users)){
?>
<ul>
<?php
foreach ($users as $key => $value){
 echo "<li>".$value."</li>\n";
}
?>
</ul>
<?php
}else{
?>
```

```
<p><b>You're not following anyone yet!</b></p>
<?php
}
?>
```

## Adding posts from other users

To add posts from other users to a user's timeline, you need only reuse some previously written code. For example, you already know how to get a list of users that the current user is following. You also know how to pull out all the posts by a certain user. You merely need to tweak the latter function to be able to accept a list of users rather than a single user.

All you need to do now is move the first function higher up in the index.php file so you can take advantage of it sooner, then use the list of user IDs you get from the function to pull out a limited number of posts from their timelines — you don't want all of them, just five or so. Remember, you want to place the posts by those other users in reverse-chronological order (most recent on top).

First things first: Add a limit argument to the `show_posts()` function, setting it to zero by default. If that limit goes higher than zero, you add a limit to your SQL statement for retrieving posts. The other thing you do is make the `$userid` argument into an array you parse into a field of commas, which you then pass to the SQL statement. This is a bit of extra work, but it pays off handsomely because all of the posts will then display in reverse order, as you can see.

### Listing 18. Updating `show_posts()` to accept an array of users

```
function show_posts($userid,$limit=0){
 $posts = array();

 $user_string = implode(',', $userid);
 $extra =  " and id in ($user_string)";

 if ($limit > 0){
  $extra = "limit $limit";
 }else{
  $extra = '';
 }

 $sql = "select user_id,body, stamp from posts
  where user_id in ($user_string)
  order by stamp desc $extra";
 echo $sql;
 $result = mysql_query($sql);

 while($data = mysql_fetch_object($result)){
  $posts[] = array(  'stamp' => $data->stamp,
       'userid' => $data->user_id,
       'body' => $data->body
     );
 }
 return $posts;

}
```

Now go back to the index.php file and work on passing in more than one user ID to `show_posts()`, as shown below. It's a simple thing, really, since you already gathered the users. You just pull out the keys using `array_keys()` and add your session variable to the mix. At minimum, this sends

in an array with one value in it (the currently logged-in user's ID). At most, it sends the logged-in user's ID and the ID of every user that user is following.

### Listing 19. Passing in an array of users to the `show_posts()` function

```
$users = show_users($_SESSION['userid']);
if (count($users)){
 $myusers = array_keys($users);
}else{
 $myusers = array();
}
$myusers[] = $_SESSION['userid'];

$posts = show_posts($myusers,5);
```

# Conclusion

In this article, you learned how to build a simple PHP-based microblogging service similar to Twitter and the Facebook status update tool. With any luck, you can take what you learned here, add it to your application, and tailor it to your needs.

# Related topics

- Get yourself a Twitter account.
- Sign up for Mr. Tweet to get suggestions for whom to follow on Twitter.
- Here's the makeuseof.com list of 15 Twitter resources.
- PHP.net is the central resource for PHP developers.
- Check out the "Recommended PHP reading list."
- Browse all the PHP content on developerWorks.
- Follow developerWorks on Twitter.
- Expand your PHP skills by checking out IBM developerWorks' PHP project resources.
- Using a database with PHP? Check out the Zend Core for IBM, a seamless, out-of-the-box, easy-to-install PHP development and production environment that supports IBM DB2 V9.