User Authentication with patUser (part 1)

By icarus
2003-04-23

Printed from DevShed.com
URL: http://www.devshed.com/Server_Side/PHP/patUser1/

Time Is Money

Over the past couple of years, Web application development has matured to a point where building a Web application is not quite as random a process as it once was. Most developers are now familiar with what goes into building the fundamental skeleton of a Web application: a database to store content, page templates to simplify maintenance, sessions and cookies for temporary data storage, and an authentication and privilege system to manage users and user security levels.

While much has been said about the first few items on the list above, the last one - user management - is a topic that has received step-motherly treatment thus far. The reason is fairly simple: different applications have different requirements of their user authentication and privilege management systems, and it's hard to come up with a generic solution to the problem in such a diverse environment. Additionally, given the time constraints that are de rigeur for Web development projects, most Web developers don't have the time to take on the problem and see it through; they find it faster to simply custom-code for each requirement.

This reasoning is, however, flawed. If you think about it, a user management system must be capable of performing certain basic tasks - user identification, credential verification, privilege assignment, user tracking and auditing - which are fairly standard across different applications. It should, therefore, be possible to come up with a generic library to perform these tasks, one which provides developers with a reusable code base and eliminates the need to reinvent the wheel every time they build the user management component of a Web application. This library should be flexible enough to adapt to different needs, powerful enough to satisfy most requirements, and robust enough to meet the performance and scalability requirements of most of the current generation of Web applications.

Which brings me, rather nicely, to patUser.

Power User

There are a number of reasons why you might want to add user authentication to your Web site. You might want to restrict access to certain pages only to a specific group of privileged users. You might want to customize the content on your site as per user preferences. Or you might just want to track user movement between the pages of your site. Regardless of why you want to add it, you should know how to go about doing it reliably and efficiently.

patUser, in the author's words, is a "user management class, that helps you with authentication, group and permission management, statistics and more". Developed by Stephan Schmidt, it is freely available for download and is packaged as a single PHP class that can be easily included in your application.

Very simply, patUser provides application developers with a set of APIs that ease the task of managing and authenticating users within the framework of a Web application. It includes a number of well-thought-out functions designed to manage users, organize users into groups,

assign privileges to users and groups, and track user logins for audit purposes. The model on which patUser based is simple enough that it can be quickly understood and exploited for rapid deployment, yet flexible enought that it can be easily extended to build fairly sophisticated user management systems for Web applications.

If you're in the business of building Web applications, and if those applications are designed to support multiple users with different privileges, you're going to find patUser invaluable to your development cycle. Written as a PHP class, patUser can easily be integrated into any PHP-based Web application, and can substantially reduce the amount of time you spend on designing, building and deploying multi-user applications. You'll find it functional, powerful and (if you're the kind who likes fiddling) easily extensible, and it'll soon be a standard component of every application you write.

Before proceeding further, you should visit the patUser home page at http://www.php-tools.de/ and download a copy of the latest version (2.1 at the time of writing). The package contains the main class file, documentation outlining the exposed methods and variables, and some example scripts.

Dump Truck
Now that the hard sell is over and you're (hopefully) all set up with patUser, let's get things started. The first thing you need to do is set up a MySQL database to hold your user data - this database will be used by patUser to perform its various tasks.

The patUser distribution comes with an SQL dump file which you can use to create the necessary tables - you'll find it in the "sql/" directory of the distribution, and it looks something like this:

```
#
# Table structure for table 'groups'
#

CREATE TABLE groups (
   gid int(11) NOT NULL auto_increment,
   name varchar(50),
   UNIQUE gid (gid)
);

#
# Table structure for table 'permissions'
#

CREATE TABLE permissions (
   id int(11) DEFAULT '0' NOT NULL,
   id_type enum('group','user') DEFAULT 'group' NOT NULL,
   part varchar(50),
   perm set('read','delete','modify','add')
);

#
# Table structure for table 'usergroups'
#

CREATE TABLE usergroups (
   uid int(11) DEFAULT '0' NOT NULL,
   gid int(11) DEFAULT '0' NOT NULL
);
```

```
#
# Table structure for table 'users'
#

CREATE TABLE users (
   uid int(10) unsigned NOT NULL auto_increment,
   username varchar(20) NOT NULL,
   passwd varchar(20) NOT NULL,
   email varchar(200),
   nologin tinyint(1) DEFAULT '0' NOT NULL,
   first_login datetime,
   last_login datetime,
   count_logins int(10) unsigned DEFAULT '0' NOT NULL,
   count_pages int(10) unsigned DEFAULT '0' NOT NULL,
   time_online int(11) DEFAULT '0' NOT NULL,
   PRIMARY KEY (uid),
   KEY username (username)
);
```

Each of the tables above plays an important role in the patUser model:

- The "users" table stores information about each user in the system - the user's login name and password, email address, and date of first and last login. Each user in this table is identified by a unique user ID.

- patUser allows you to organize users into groups - the "groups" table stores a list of these groups. As with users, each group has a name and a unique group ID.

- The "usergroups" table connects the records in the "users" and "groups" table together, specifying which users belong to which groups. patUser groups are not exclusive - a user may belong to more than one group at a time.

- The "permissions" table makes it possible to assign permissions for a user or group. These permissions can be used to restrict user activities on a page-by-page basis.

It's important to note, at this stage, that although the developers of patUser recommend using the schema above, there is no restriction on you, as a developer, creating and using your own set of tables. patUser's developers are well aware that different applications have different requirements and the schema above may not be suitable for all needs; therefore, they have designed the patUser library to be flexible enough for use with other, custom schemas as well (more on this shortly).

Zone Six
I'll begin with something simple - using patUser to restrict access to a Web page. In this scenario, only users who provide appropriate credentials - a login name and password - will be allowed access to the page; all other users will simply be presented with an error message.

Here's the code:

```
<?php

// include classes
include("../include/patDbc.php"); include("../include/patUser.php");
include("../include/patTemplate.php");
```

```
// initialize database layer
$db = new patMySqlDbc("localhost", "db111", "us111", "secret");

// initialize template engine
$tmpl = new patTemplate();
$tmpl->setBasedir("../templates");

// initialize patUser
$u = new patUser(true);

// connect patUser to database/template engines
$u->setAuthDbc($db);
$u->setTemplate($tmpl);

// check credentials before displaying page
$uid = $u->requireAuthentication("displayLogin");

// restricted page goes here

?>

<html>
<head>
<basefont face="Arial">
</head>

<body>

<center>
<h2>Welcome to Zone 6!</h2>
<u>This is a restricted zone. Trespassers will be vaporized.</u>
</center>

<p align="right">

Your user ID is <?=$uid?>

<br>

<a href="logout.php">Log out</a>

</body>
</html>
```

As you can see, the script above contains an HTML page which I'm assuming you want to restrict access to, surrounded with some fairly complicated code. The code is all explained a little further down, so don't worry too much about it just yet; instead, try accessing this page through your Web browser.

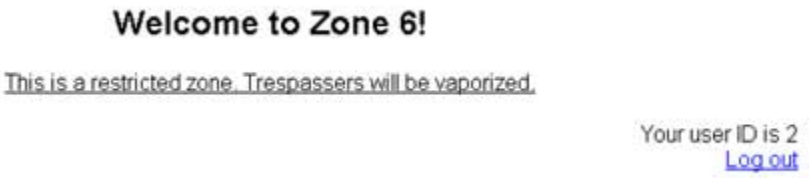
You should see something like this:

Try entering a random user name and password - since the database is currently empty, the system should barf and throw you back out with an error message.

Now, add a user to the "users" table,

```
INSERT INTO users (`uid`, `username`, `passwd`) VALUES ('', 'joe',
'joe');
```

and try logging in again with that username and password. This time, patUser should recognize that your credentials are valid and allow you access to the page.



Breaking It Down
Let's take a closer look at the code to see how this works.

1. The first step is to include the patUser class in your script:

```
<?php

include("../include/patUser.php");

?>
```

patUser uses the patDbc class to connect to the user database, and the patTemplate engine to render its pages - so let's include those as well here:

```
<?php
```

```
include("../include/patDbc.php");
include("../include/patTemplate.php");

?>
```

2. Next, the database layer needs to be initialized. This is accomplished by creating an instance of the patDbc class and providing it with the access parameters needed to connect to the MySQL database holding the user data. This object instance is stored in the PHP variable $db.

```
<?php

$db = new patMySqlDbc("localhost", "db111", "us111", "secret");

?>
```

3. Once the database connection has been initialized, the patTemplate engine also needs to be initialized. Here too, an instance of the patTemplate class is created, and provided with the location of the directory containing patUser's templates.

```
<?php

$tmpl = new patTemplate();
$tmpl->setBasedir("../templates");

?>
```

If you look in this directory, you'll see two templates, named "patUserLogin.tmpl" and "patUserUnauthorized.tmpl". The first of these templates contains the login box that is displayed whenever the system requires user credentials, while the second contains a generic "access denied" page that is displayed when an attempt is made to access a restricted page without appropriate privileges. Both these templates are included as part of the patUser distribution, and you are free to make changes to them so that they fit into the look and feel of your application.

4. Once both template engine and database connection are awake, it's time to initialize the patUser class itself.

```
<?php

$u = new patUser(true);

?>
```

The "true" argument to the class constructor tells patUser to use PHP's session management capabilities to store user data in the session.

Once the object instance has been initialized, the methods setAuthDbc() and setTemplate() are used to connect it to the database connection and template engine respectively. patUser will use this information to send queries to the database and render page templates as needed.

```
<?php

$u->setAuthDbc($db);
$u->setTemplate($tmpl);

?>
```

The four steps above are fairly standard for all scripts using the patUser class, and so it's a good idea to encapsulate them into a single function, say init(), and call that function at the start of every script so as to eliminate unnecessary repetition. A single init() function reduces redundancy, and also makes changes easier.

Once all the formalities are concluded, the requireAuthentication() method may be called to verify the user's credentials.

```
<?php

$u->requireAuthentication("displayLogin");

?>
```

This method tells patUser that what follows is a restricted page, and access should only be allowed if appropriate user credentials are available. In case these credentials are not available in the session, patUser will automatically display the contents of the "patUserLogin.tmpl" template, usually a login box. Once user credentials have been submitted, patUser will internally verify them and either permit access or display an error message.

The actual mechanics of the authentication are performed internally and automatically by the requireAuthentication() method, and you don't really need to worry about it - although if you're really interested, feel free to pop open the class file and inspect the code.

By encapsulating the business logic of user verification into a single function, patUser makes life much easier for the harried application developer - all that's needed is to add a single function call at the top of all such sensitive pages, and patUser automatically takes care of the rest.

A Different Realm

If you'd prefer not to futz around with templates, patUser also supports plain-vanilla HTTP authentication. In this case, the browser takes care of displaying a login box; patUser merely verifies the credentials entered by the user into that box.

In order to enable this type of authentication, all you need to do is forget to connect patUser to a patTemplate object. Since patUser will not have a template to use for the login box, it will automatically fall back to using HTTP authentication. Consider the following script, which illustrates:

```
<?php

// include classes
include("../include/patDbc.php"); include("../include/patUser.php");

// initialize database layer
$db = new patMySqlDbc("localhost", "db111", "us111", "secret");

// initialize patUser
$u = new patUser(true);

// connect patUser to database
$u->setAuthDbc($db);

// set realm for display
$u->setRealm("Highly Secure Zone");

// check credentials before displaying page
```

```
$uid = $u->requireAuthentication("displayLogin");

// restricted page goes here

?>

<html>
<head>
<basefont face="Arial">
</head>

<body>

<center>
<h2>Welcome to Zone 6!</h2>
<u>This is a restricted zone. Trespassers will be vaporized.</u>
</center>

<p align="right">

Your user ID is <?=$uid?>

<br>

<a href="logout.php">Log out</a>

</body>
</html>
```

Here's what the result might look like:

Note my introduction of a new method in the script above, the setRealm() method. This method is used to set the name of the realm for HTTP authentication, and appears in the login box displayed by the browser.

Icing On The Cake
In case your application database schema is already designed and integrating the patUser schema into it would lead to redundancies or complications, you can configure patUser to use a different set of tables than its default. This is accomplished by means of the setAuthTables() and setAuthFields() methods, which allow you to remap patUser's default table and column references to custom values of your own.

In order to better understand this, consider the following table, which contains user authentication information:

```
CREATE TABLE zx_users (
  uid int(10) unsigned NOT NULL auto_increment,
  uname varchar(20) NOT NULL default '',
  upswd varchar(20) NOT NULL default '',
```

```
  PRIMARY KEY  (uid)
) TYPE=MyISAM;
```

Now, while this table contains a subset of the information in patUser's "users" table, it does not have the same table and column names. I can, however, still integrate it with patUser, simply by remapping patUser to use the new table and columns. The following example demonstrates:

```
<?php

// include classes
include("../include/patDbc.php"); include("../include/patUser.php");
include("../include/patTemplate.php");

// initialize database layer
$db = new patMySqlDbc("localhost", "db211", "us111", "secret");

// initialize template engine
$tmpl = new patTemplate();
$tmpl->setBasedir("../templates");

// initialize patUser
$u = new patUser(true);

// connect patUser to database/template engines $u->setAuthDbc($db);
$u->setTemplate($tmpl);

// tell patUser where to get user information from
$u->setAuthTable("zx_users"); $u->setAuthFields( array(
        "uid" => "uid",
        "username" => "uname",
        "passwd" => "upswd"
));

// check credentials before displaying page
$uid = $u->requireAuthentication("displayLogin");

// restricted page goes here

?>
```

As you can see, I've told patUser where to find the data by remapping the old column names to the ones in the new table.

Though the requireAuthentication() method displays a login box by default, this default behaviour can be altered by passing a different argument to the method. If, instead of displaying a login box, you would simply like a blank screen to be displayed when an unauthenticated user arrives at the page, you can use the following code:

```
<?php

$u->requireAuthentication("exit");

?>
```

As you might imagine, the "exit" argument tells the requireAuthentication() method to simply exit without displaying a login box. The end result? Users without appropriate authentication

credentials will see an empty page.

The return value of the requireAuthentication() method is a user ID (if the authentication is successful) or false (if it isn't); this user ID may then be used in subsequent PHP code. patUser also provides a getUid() method, which can be used at any time to retrieve the currently logged-in user's ID (you'll see this in action in subsequent examples).

Making A Graceful Exit
You can also use the isAuthenticated() utility function to check if a user is authenticated, as demonstrated below:

```
<?php

// include classes
include("../include/patUser.php");

// initialize patUser
$u = new patUser(true);

if ($u->isAuthenticated())
{
        echo "Welcome!";

        // or you could display the page
}
else
{
        echo "Leave now or I'll make you!";

        // or you could redirect the user to an error page
}

?>
```

You can also restrict the maximum number of login attempts with the setMaxLoginAttempts() method. Once that maximum number is reached, patUser will automatically display the contents of the template "patUserUnauthorized.tmpl", or you can redirect the user to a new URL, which you can set with the setUnauthorizedUrl() method.

Finally, now that you know how to handle logging in, how about logging out? Well, it's equally simple - patUser provides a logOut() method, which terminates the user's session and destroys all related user information. Consider the following example, which illustrates:

```
<?php

// include class
include("../include/patUser.php");

// initialize patUser
$u = new patUser(true);

// log out
$u->logOut();

// redirect to index page
header("Location: index.php");
```

```
?>
```

That's about all for the moment. In this article, I discussed the patUser class, explaining some of its methods and illustrating how they could be used in the context of a Web application to authenticate and verify user credentials. I showed you how Web pages could be secured by the simple addition of a single method call, how to modify patUser so it fits into your application's overall look and feel, and how to integrate a custom database schema into the patUser world.

Thus far, I've been assuming the existence of a correctly filled-in database for patUser to use, without wondering too much about how that database was created or maintained. In the second part of this article, those items will come to the fore, when I discuss built-in patUser functions to retrieve user data; add, modify and delete users and user information; organize users into groups; and manager user and group relationships. Make sure you come back for that one!

Note: Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!