By icarus

Table of Contents

The Road Ahead

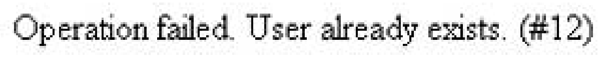
In the previous segment of this article, I discussed the various library functions related to user and group management, and also showed you some examples of how they could be used in real-world Web application development.
While the API for user and group management does form the core of the patUser library, it’s not all there is on display. patUser also includes a number of utility functions that can come in handy for certain specialized tasks. These functions include identifying users and groups by different criteria (such as name or email address); keeping track of the URLs visited by the user so as to generate a user trail; maintaining user account statistics for auditing purposes; and providing exception-handling routines for more graceful error handling.
All these (and more) are discussed in the following pages - so keep reading!

Making Exceptions

First up, error handling. You’ll have seen, from the previous sections of this article, that many of patUser’s functions return either true or false to indicate success or failure; this value can then be used in your script to display an appropriate status message.
However, patUser also goes further, allowing you to display a more verbose message indicating the cause of error if one occurred. A built-in error stack keeps track of all the errors that have occurred, and the library include a number of functions designed to help you read and modify this stack.
The two functions you’re most likely to use are getLastErrorCode() and getLastErrorMessage(), which return the last error code and message respectively. Consider the following example, which demonstrates:

```
setAuthDbc($db);
// set tables
$u->setAuthTable("users");
// add user
$u->addUser( array("username" => "tom", "passwd" => "tom") );
// add user again
$u->addUser( array("username" => "tom", "passwd" => "tom") );
// get message and display
echo "Operation failed. " . $u->getLastErrorMessage() . " (#" .
$u->getLastErrorCode() . ")";
?>
```

Since there’s a subtle error in the script above, patUser will not be able to execute the call to addUser() successfully. An error code will be generated internally and added to the error stack; this code can be viewed and retrieved via the calls to getLastErrorCode() and getLastErrorMessage(). Here’s an example of what you might see:



```
Operation failed. User already exists. (#12)
```

Obviously, this is a little more informative than the standard "User could not be added" message.
You can also use the shortcut getLastError() method, which returns an array containing both error code and message. The following example, which is equivalent to the one above, illustrates:

```
setAuthDbc($db);
// set tables
$u->setAuthTable("users");
// add user
$u->addUser( array("username" => "tom", "passwd" => "tom") );
// add user again
$u->addUser( array("username" => "tom", "passwd" => "tom") );
// get message and display
$e = $u->getLastError();
echo "Operation failed. " . $e[’message’] . " (#" . $e[’code’] .
  ")";
  ?>
```

You can obtain a complete list of all the errors in the stack via the getAllErrorCodes() and getAllErrorMessages() methods, which return arrays of all the error codes and messages generated by patUser during the execution of the script, or again use the equivalent shortcut method getAllErrors() to get both codes and messages in one pass. A complete list of possible error codes and what they mean is available in the patUser documentation.

The History Channel

patUser allows you to track user movement through your application with a set of history functions, which maintain a buffer of all the URLs visited and provide you with hooks to return to any of them. As you might imagine, this comes in handy if you need to provide users with controls to go back to the previous page, or to show them their browsing history.
The primary method to accomplish this is the keepHistory() method, a call to which should be included on every page of your application. The function accepts two parameters, the size of the history buffer and the title of the current page, and stores the current URL and supplied title in the session for easy retrieval.
Here's a sample of what that history buffer might look like:

```
Array
(
[0] => Array
(
[url] => /add_user.php?
[title] => Add User
)
[1] => Array
(
[url] => /patuser/modify_user.php?
[title] => Modify User
)
[2] => Array
(
[url] => /scripts/review.php?
[title] => Verify User
)
)
```

Once the history buffer has been created, data from it may be retrieved via the getHistory() method, which returns an array holding a list of all the URLs and pages visited by the user in the current session. The following example illustrates, using the sample data above as reference.

```
<?php
// include classes
include("../include/patDbc.php"); include("../include/patUser.php");
// initialize database layer
$db = new patMySqlDbc("localhost", "db211", "us111", "secret");
// initialize patUser
$u = new patUser(true);
// connect patUser to database
$u->setAuthDbc($db);
// set tables
```

```
$u->setAuthTable("users");
// get history
$history = $u->getHistory();
// iterate over array
// display history with clickable URLs
echo "<h2>Your history:</h2>";
echo "<ol>";
foreach ($history as $h)
{
echo "<li><a href=" . $h['url'] . ">" . $h['title'] . "</a>"; } echo
"</ol>";
?>
```

An additional goHistory() method makes it possible to send the browser back to a specific page in the buffer via HTTP redirection. Consider the following script, which accepts a number and redirects the browser back that many pages (note that the goHistory() method must be provided with negative values as input):

```
<?php
// include classes
include("../include/patDbc.php"); include("../include/patUser.php");
// initialize database layer
$db = new patMySqlDbc("localhost", "db211", "us111", "secret");
// initialize patUser
$u = new patUser(true);
// connect patUser to database
$u->setAuthDbc($db);
// set tables
$u->setAuthTable("users");
// go back i pages
$u->goHistory($_GET['i'] * -1);
?>
```

Natural Selection

You’ve already seen, in my examination of the getUsers() and getGroups() methods, that you can apply selection criteria (as the second argument to those methods) to return a result subset containing only the records you need. Here’s an example:

```
<?php
$list = $u->getUsers(
array("uid", "username"),
array( array("field" => "sex", "value" => "F", "match" =>
"exact") ) );
?>
```

The code snippet above would return a list of only those users who were female.
There are three components to the second argument in the snippet above: the field, the value, and the selection criteria. The first two are fairly obvious: the "field" key specifies the field against which to match, while the "value" specifies the value against which records are to be compared. The critical element, though, is the third key, "match", which defines how the comparison between the "field" and the "value" will actually be performed.
patUser permits comparison using any of the following conditions:
"match" => "exact" field value equals "value" when compared in a case-sensitive manner
"match" => "greater" field value must be greater than "value"
"match" => "lower" field value must be lower than "value"
"match" => "greaterorequal" field value must be greater than or equal to "value"
"match" => "lowerorequal" field value must be lower than or equal to "value"
"match" => "contains" field value contains "value"
"match" => "like" field value equals "value" when compared in a case-insensitive manner
"match" => "startswith" field value starts with "value"
"match" => "endswith" field value starts with "value"
"match" => "between" field value is between the "value" array (x,y)
The following snippet would return the IDs of all users whose username contains an "e",

```
<?php
$list = $u->getUsers(
array("uid"),
array( array("field" => "username", "value" => "e", "match" =>
"contains") ) );
?>
```

while this one would return usernames and email addresses of all those users who have logged in more than 11 times and are over 18 years old:

```
<?php
$list = $u->getUsers(
array("username", "email"),
array(
array("field" => "count_logins", "value" => "11",
"match" => "greater"),
array("field" => "age", "value" => "18", "match" =>
"greater") ) );
?>
```

No Distinguishing Marks

That isn’t all, though - patUser also comes with two other functions, identifyUser() and identifyGroup(), which allow you to identify users and groups by other criteria. Consider the following example, which demonstrates:

```
setAuthDbc($db);
// set tables
$u->setAuthTable("users");
// get UID
$uid = $u->identifyUser(array("email" => "tom@some-domain.com"));
echo
$uid;
?>
```

This method is essentially a simpler version of the getUsers() method; it checks the user database for matching records and returns the user ID of the found user if available.
You can add as many search criteria as you like, as in the following (equivalent) example:

```
setAuthDbc($db);
// set tables
$u->setAuthTable("users");
// get UID
$uid = $u->identifyUser(array("email" => "tom@some-domain.com",
"sex" => "M", "age" => 21, "time_online" => 0)); echo $uid;
?>
```

Similarly, there’s also an identifyGroup() method, which can be used to quickly look up a group ID, given the group name (or any other group attribute). Take a look:

```
setAuthDbc($db);
// set tables
$u->setGroupTable("groups");
// get GID
$gid = $u->identifyGroup(array("name" => "Operations"));
echo $gid;
?>
```

If no user or group can be found matching the specified criteria, both functions will return false.

Big Brother Is Watching

patUser also allows you to record statistical information about user logins via its very cool addStats() and updateStats() methods. There are five different pieces of data you can store: the user’s first login, last login, total number of logins, total number of pages visited and total time on your site. The data thus collected is stored in the "users" table as a component of each user record, and can be viewed using regular SQL queries.
In order to enable this feature, you need to tell patUser which pieces of data you want to track by calling addStats()at the top of your PHP scripts, once for each item. Once that’s done, you can update the database with the latest statistics at any time by calling updateStats(). 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);
// decide which data to save
$u->addStats("last_login");
$u->addStats("count_logins");
$u->addStats("count_pages");
$u->addStats("time_online");
// update statistics
$u->updateStats();
?>
```

If you’re using a single init() function to initialize your patUser object, the calls to addStats() should probably go into that function, so that statistics generation is enabled every time patUser starts up. And if you’re not using patUser’s default tables, you can have the data saved to your own custom fields, simply by specifying each field name as the second argument in your calls to addStats().
These statistics can come in handy for database administrators to track user logins and gauge site popularity via time spent online and number of logins; it can also provide an audit trail for internal usage tracking and monitoring.

Endgame

And that’s about it for this series. Over the last few pages, I spent lots of time and energy driving you through the landscape of the patUser library. I explained the important utility functions in the library, discussing error handling, history tracking, user identification and statistics generation within the context of not-so-real-life illustrative examples.
You should now (hopefully!) know enough about patUser to begin using it in your daily development activities. The authors have spent a fair amount of time thinking about the most common requirements of a user management library, and the end result of their efforts is a class that’s both sophisticated and flexible enough to be used for a wide variety of purposes. I strongly encourage you to exploit this power for your own projects - I can tell you from experience that it will substantially reduce the amount of timeyou spend bulding user management modules for your Web applications.
In case you’d like to learn more about patUser and other, related projects, you should consider visiting the following links:
The official pat Web site, at http://www.php-tools.de/
patUser documentation, at http://www.php-tools.de/site.php?&file=patUserDocumentation.xml
A discussion of building template-based Web sites with patTemplate, at http://www.devshed.com/Server_Side/PHP/patTemplate
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!