1

User Registration

Offering account registration and user logins is a great way of giving users a sense of individuality and serving tailored content. Such authentication is often at the very heart of many community-oriented and e-commerce web sites. Because this functionality is so useful, the first application I present is a user registration system.

From a functional perspective, the system will allow users to create accounts. Members must provide an e-mail address that they can use to validate their registration. Users should also be able to update their passwords and e-mail addresses and reset forgotten passwords. This is pretty standard functionality and what the web users of today have come to expect.

From an architectural standpoint, the directory holding your code should be logically organized. For example, support and include files should be kept outside of a publically accessible directory. Also, user records should be stored in a database. Since there are a large number of tools designed to view and work with data stored in relational databases such as MySQL, this affords transparency and flexibility.

Plan the Directory Layout

The first step is to plan the directory structure for the application. I'm going to recommend you create three main folders: One named `public_files` from which all publicly accessible files will be served, another named `lib` to store include files to be shared by any number of other files, and finally a `templates` folder to store presentation files. Although PHP will be able to reference files from anywhere in your setup, the web server should only serve files from the `public_files` folder. Keeping support files outside of the publicly accessible directory increases security.

Inside the `public_files` I also create `css` to store any style sheets, `js` for JavaScript source files and `img` for graphic files. You may want to create other folders to keep yourself organized. One named `sql` to store MySQL files would be a good idea, `doc` for documentation and development notes and `tests` to store smoke test or unit testing files.

Planning the Database

In addition to planning the directory layout, thought needs to be given to the database layout as well. The information you choose to collect from your users will depend on what type of service your site offers. In turn, this affects how your database tables will look. At the very least a unique user ID, username, password hash, and e-mail address should be stored. You will also need a mechanism to track which accounts have been verified or are pending verification.

```
CREATE TABLE WROX_USER (
    USER_ID    INTEGER UNSIGNED  NOT NULL  AUTO_INCREMENT,
    USERNAME   VARCHAR(20)       NOT NULL,
    PASSWORD   CHAR(40)          NOT NULL,
    EMAIL_ADDR VARCHAR(100)      NOT NULL,
    IS_ACTIVE  TINYINT(1)        DEFAULT 0,

    PRIMARY KEY (USER_ID)
)
ENGINE=MyISAM DEFAULT CHARACTER SET latin1
    COLLATE latin1_general_cs AUTO_INCREMENT=0;

CREATE TABLE WROX_PENDING (
    USER_ID       INTEGER UNSIGNED  NOT NULL,
    TOKEN         CHAR(10)          NOT NULL,
    CREATED_DATE  TIMESTAMP         DEFAULT  CURRENT_TIMESTAMP,

    FOREIGN KEY (USER_ID)
        REFERENCES WROX_USER(USER_ID)
)
ENGINE=MyISAM DEFAULT CHARACTER SET latin1
    COLLATE latin1_general_cs;
```

I have allocated 40 characters of storage for the password hash in `WROX_USER` as I will use the `sha1()` function which returns a 40-character hexadecimal string. You should never store the original password in the database — a good security precaution. The idea here is that a hash is generated when the user provides his or her password for the first time. The password given subsequently is hashed using the same function and the result is compared with what's stored to see if they match.

I set the maximum storage length for an e-mail address at 100 characters. Technically the standards set the maximum length for an e-mail address at 320 (64 characters are allowed for the username, one for the `@` symbol and then 255 for the hostname). I don't know anyone that has such a long e-mail address though and I have seen plenty of database schemas that use 100 and work fine.

Some other information you may want to store are first and last name, address, city, state/province, postal code, phone numbers, and the list goes on.

The `WROX_PENDING` table has an automatically initializing timestamp column, which lets you go back to the database and delete pending accounts that haven't been activated after a certain amount of time. The table's columns could be merged with `WROX_USER`, but I chose to separate them since the pending token is only used once. User data is considered more permanent and the `WROX_USER` table isn't cluttered with temporary data.

Writing Shared Code

Code that is shared by multiple files should be set aside in its own file and included using `include` or `require` so it's not duplicated, which makes maintaining the application easier. Where possible, code that might be useful in future applications should be collected separately as functions or classes to be reused. It's a good idea to write code with reusability in mind. `common.php` contains shared code to be included in other scripts in the application to establish a sane baseline environment at runtime. Since it should never be called directly by a user, it should be saved in the `lib` directory.

```
<?php
// set true if production environment else false for development
define ('IS_ENV_PRODUCTION', true);

// configure error reporting options
error_reporting(E_ALL | E_STRICT);
ini_set('display_errors', !IS_ENV_PRODUCTION);
ini_set('error_log', 'log/phperror.txt');

// set time zone to use date/time functions without warnings
date_default_timezone_set('America/New_York');

// compensate for magic quotes if necessary
if (get_magic_quotes_gpc())
{
    function _stripslashes_rcurs($variable, $top = true)
    {
        $clean_data = array();
        foreach ($variable as $key => $value)
        {
            $key = ($top) ? $key : stripslashes($key);
            $clean_data[$key] = (is_array($value)) ?
                stripslashes_rcurs($value, false) : stripslashes($value);
        }
        return $clean_data;
    }
    $_GET = _stripslashes_rcurs($_GET);
    $_POST = _stripslashes_rcurs($_POST);
    // $_REQUEST = _stripslashes_rcurs($_REQUEST);
    // $_COOKIE = _stripslashes_rcurs($_COOKIE);
}
?>
```

You may not always have control over the configuration of your server so it is wise to specify some common directives to make your applications more portable. Setting error reporting options, for example, lets you display errors while in development or redirect them in a production environment so they don't show to the user.

Magic quotes is a configuration option where PHP can automatically escape single quotes, double quotes, and backslashes in incoming data. Although this might seem useful, assuming whether this directive is on or not can lead to problems. It's better to normalize the data first and then escape it with `addslashes()` or `mysql_real_escape_string()` (preferably the latter if it's going to be stored in the database) when necessary. Compensating for magic quotes ensures data is properly escaped *how* you want and *when* you want despite how PHP is configured, making development easier and less error-prone.

Establishing a connection to a MySQL database is a common activity which makes sense to move out to its own file. `db.php` holds configuration constants and code to establish the connection. Again, as it is meant to be included in other files and not called directly, it should be saved in `lib`.

```
<?php
// database connection and schema constants
define('DB_HOST', 'localhost');
define('DB_USER', 'username');
define('DB_PASSWORD', 'password');
define('DB_SCHEMA', 'WROX_DATABASE');
define('DB_TBL_PREFIX', 'WROX_');

// establish a connection to the database server
if (!$GLOBALS['DB'] = mysql_connect(DB_HOST, DB_USER, DB_PASSWORD))
{
    die('Error: Unable to connect to database server.');
}
if (!mysql_select_db(DB_SCHEMA, $GLOBALS['DB']))
{
    mysql_close($GLOBALS['DB']);
    die('Error: Unable to select database schema.');
}
?>
```

The `DB_HOST`, `DB_USER`, `DB_PASSWORD` and `DB_SCHEMA` constants represent the values needed to establish a successful connection to the database. If the code is put into production in an environment where the database server is not running on the same host as PHP and the web server, you might also want to provide a `DB_PORT` value and adjust the call to `mysql_connect()` appropriately.

The connection handle for the database is then stored in the `$GLOBALS` super global array so it is available in any scope of any file that includes `db.php` (or that is included in the file that has referenced `db.php`).

Prefixing table names helps prevent clashes with other programs' tables that might be stored in the same schema and providing the prefix as a constant makes the code easier to update later if it should change, since the value appears just in one place.

Common functions can also be placed in their own files. I plan to use this `random_text()` function, for example, to generate a CAPTCHA string and validation token so it can be saved in a file named `functions.php`.

```
<?php
// return a string of random text of a desired length
function random_text($count, $rm_similar = false)
{
    // create list of characters
    $chars = array_flip(array_merge(range(0, 9), range('A', 'Z')));
    // remove similar looking characters that might cause confusion
    if ($rm_similar)
    {
        unset($chars[0], $chars[1], $chars[2], $chars[5], $chars[8],
            $chars['B'], $chars['I'], $chars['O'], $chars['Q'],
            $chars['S'], $chars['U'], $chars['V'], $chars['Z']);
    }

    // generate the string of random text
    for ($i = 0, $text = ''; $i < $count; $i++)
    {
        $text .= array_rand($chars);
    }

    return $text;
}
?>
```

An important rule when programming no matter what language you're using is to never trust user input. People can (and will) provide all sorts of crazy and unexpected input. Sometimes this is accidental, at other times it's malicious. PHP's `filter_input()` and `filter_var()` functions can be used to scrub incoming data, though some people still prefer to write their own routines, as the filter extension may not be available in versions prior to 5.2.0. If you're one of those people, then they can be placed in `functions.php` as well.

User Class

The majority of the code written maintaining a user's account can be encapsulated into one data structure, making it easy to extend or reuse in future applications. This includes the database interaction logic, which will make storing and retrieving information easier. Here's `User.php`:

```
<?php
class User
{
    private $uid;     // user id
    private $fields;  // other record fields

    // initialize a User object
    public function __construct()
    {
        $this->uid = null;
        $this->fields = array('username' => '',
                              'password' => '',
                              'emailAddr' => '',
                              'isActive' => false);
    }

    // override magic method to retrieve properties
    public function __get($field)
```

(continued)

(continued)

```
        {
            if ($field == 'userId')
            {
                return $this->uid;
            }
            else
            {
                return $this->fields[$field];
            }
        }

        // override magic method to set properties
        public function __set($field, $value)
        {

            if (array_key_exists($field, $this->fields))
            {
                $this->fields[$field] = $value;
            }
        }

        // return if username is valid format
        public static function validateUsername($username)
        {
            return preg_match('/^[A-Z0-9]{2,20}$/i', $username);
        }

        // return if email address is valid format
        public static function validateEmailAddr($email)
        {
            return filter_var($email, FILTER_VALIDATE_EMAIL);
        }

        // return an object populated based on the record's user id
        public static function getById($user_id)
        {
            $user = new User();
            $query = sprintf('SELECT USERNAME, PASSWORD, EMAIL_ADDR, IS_ACTIVE ' .
                'FROM %sUSER WHERE USER_ID = %d', DB_TBL_PREFIX, $user_id);
            $result = mysql_query($query, $GLOBALS['DB']);
            if (mysql_num_rows($result))
            {
                $row = mysql_fetch_assoc($result);
                $user->username = $row['USERNAME'];
                $user->password = $row['PASSWORD'];
                $user->emailAddr = $row['EMAIL_ADDR'];
                $user->isActive = $row['IS_ACTIVE'];
                $user->uid = $user_id;
            }
            mysql_free_result($result);
            return $user;
        }
```

```
    // return an object populated based on the record's username
    public static function getByUsername($username)
    {
        $user = new User();
        $query = sprintf('SELECT USER_ID, PASSWORD, EMAIL_ADDR, IS_ACTIVE ' .
            'FROM %sUSER WHERE USERNAME = "%s"', DB_TBL_PREFIX,
            mysql_real_escape_string($username, $GLOBALS['DB']));
        $result = mysql_query($query, $GLOBALS['DB']);
        if (mysql_num_rows($result))
        {
            $row = mysql_fetch_assoc($result);
            $user->username = $username;
            $user->password = $row['PASSWORD'];
            $user->emailAddr = $row['EMAIL_ADDR'];
            $user->isActive = $row['IS_ACTIVE'];
            $user->uid = $row['USER_ID'];
        }
        mysql_free_result($result);
        return $user;
    }

    // save the record to the database
    public function save()
    {
        if ($this->uid)
        {
            $query = sprintf('UPDATE %sUSER SET USERNAME = "%s", ' .
                'PASSWORD = "%s", EMAIL_ADDR = "%s", IS_ACTIVE = %d ' .
                'WHERE USER_ID = %d', DB_TBL_PREFIX,
                mysql_real_escape_string($this->username, $GLOBALS['DB']),
                mysql_real_escape_string($this->password, $GLOBALS['DB']),
                mysql_real_escape_string($this->emailAddr, $GLOBALS['DB']),
                $this->isActive, $this->userId);
            return mysql_query($query, $GLOBALS['DB']);
        }
        else
        {
            $query = sprintf('INSERT INTO %sUSER (USERNAME, PASSWORD, ' .
                'EMAIL_ADDR, IS_ACTIVE) VALUES ("%s", "%s", "%s", %d)',
                DB_TBL_PREFIX,
                mysql_real_escape_string($this->username, $GLOBALS['DB']),
                mysql_real_escape_string($this->password, $GLOBALS['DB']),
                mysql_real_escape_string($this->emailAddr, $GLOBALS['DB']),
                $this->isActive);
            if (mysql_query($query, $GLOBALS['DB']))
            {
                $this->uid = mysql_insert_id($GLOBALS['DB']);
                return true;
            }
```

(continued)

(continued)

```
            else
            {
                return false;
            }
        }
    }

    // set the record as inactive and return an activation token
    public function setInactive()
    {
        $this->isActive = false;
        $this->save(); // make sure the record is saved

        $token = random_text(5);
        $query = sprintf('INSERT INTO %sPENDING (USER_ID, TOKEN) ' .
            'VALUES (%d, "%s")', DB_TBL_PREFIX, $this->uid, $token);
        return (mysql_query($query, $GLOBALS['DB'])) ? $token : false;
    }

    // clear the user's pending status and set the record as active
    public function setActive($token)
    {
        $query = sprintf('SELECT TOKEN FROM %sPENDING WHERE USER_ID = %d ' .
            'AND TOKEN = "%s"', DB_TBL_PREFIX, $this->uid,
            mysql_real_escape_string($token, $GLOBALS['DB']));
        $result = mysql_query($query, $GLOBALS['DB']);
        if (!mysql_num_rows($result))
        {
            mysql_free_result($result);
            return false;
        }
        else
        {
            mysql_free_result($result);
            $query = sprintf('DELETE FROM %sPENDING WHERE USER_ID = %d ' .
                'AND TOKEN = "%s"', DB_TBL_PREFIX, $this->uid,
                mysql_real_escape_string($token, $GLOBALS['DB']));
            if (!mysql_query($query, $GLOBALS['DB']))
            {
                return false;
            }
            else
            {
                $this->isActive = true;
                return $this->save();
            }
        }
   }
}
?>
```

The class has two private properties: `$uid` which maps to the `WROX_USER` table's `USER_ID` column and the array `$fields` which maps to the other columns. They are exposed in an intuitive manner by overriding the `__get()` and `__set()` magic methods, but I still protect `$uid` from accidental change.

The static `getById()` and `getByUsername()` methods contain code responsible for retrieving the record from the database and populating the object. `save()` writes the record to the database and is smart enough to know when to execute an INSERT query or an UPDATE query based on if the user ID is set. All that's necessary to create a new user account is to obtain a new instance of a `User` object, set the record's fields, and call `save()`.

```
<?php
$u = new User();
$u->username = 'timothy';
$u->password = sha1('secret');
$u->emailAddr = 'timothy@example.com';
$u->save();
?>
```

It's the same logic to update an account; I retrieve the existing account, make my changes, and again save it to the database with `save()`.

```
<?php
$u = User::getByUsername('timothy');
$u->password = sha1('new_password');
$u->save();
?>
```

The `setInactive()` and `setActive()` methods handle the account activation. Calling `setInactive()` marks the account inactive, generates, an activation token, stores, the information in the database, and returns, the token. When the user activates their account, you accept the token and provide it to `setActive()`. The method will remove the token record and set the account active.

CAPTCHA

The word CAPTCHA stands for *Completely Automated Public Turing Test to Tell* **C***omputers and Humans Apart*. Besides being a painfully contrived acronym, CAPTCHAs are often used as a deterrent to keep spammers and other malicious users from automatically registering user accounts.

The user is presented with a challenge, oftentimes as a graphical image containing letters and numbers. He or she then has to read the text and enter it in an input field. If the two values match, then it is assumed an intelligent human being and not a computer is requesting the account sign-up.

It's not a perfect solution, however. CAPTCHAs cause problems for legitimate users with special accessibility needs, and some modern software can read the text in CAPTCHA images (see `www.cs.sfu.ca/~mori/research/gimpy/`). There are other types of challenges which can be presented to a user. For example, there are audio CAPTCHAs where the user enters the letters and numbers after hearing them recited in an audio file. Some even present math problems to the user.

CAPTCHAs should be considered a tool in the web master's arsenal to deter lazy miscreants and not a replacement for proper monitoring and security. Inconvenience to the visitor increases with the complexity of the challenge method, so I'll stick with a simple image-based CAPTCHA example here.

```
<?php
include '../../lib/functions.php';

// must start or continue session and save CAPTCHA string in $_SESSION for it
// to be available to other requests
if (!isset($_SESSION))
{
    session_start();
    header('Cache-control: private');
}

// create a 65x20 pixel image
$width = 65;
$height = 20;
$image = imagecreate(65, 20);

// fill the image background color
$bg_color = imagecolorallocate($image, 0x33, 0x66, 0xFF);
imagefilledrectangle($image, 0, 0, $width, $height, $bg_color);

// fetch random text
$text = random_text(5);

// determine x and y coordinates for centering text
$font = 5;
$x = imagesx($image) / 2 - strlen($text) * imagefontwidth($font) / 2;
$y = imagesy($image) / 2 - imagefontheight($font) / 2;

// write text on image
$fg_color = imagecolorallocate($image, 0xFF, 0xFF, 0xFF);
imagestring($image, $font, $x, $y, $text, $fg_color);

// save the CAPTCHA string for later comparison
$_SESSION['captcha'] = $text;

// output the image
header('Content-type: image/png');
imagepng($image);

imagedestroy($image);
?>
```

I recommend saving the script in the `public_files/img` folder (since it needs to be publically accessible and outputs a graphic image) as `captcha.php`. The image it creates is a 65×20 pixel PNG graphic with blue background and a white random text string five characters long, as seen in Figure 1-1. The string must be stored as a `$_SESSION` variable so you can check later to see if the user enters it correctly. To make the image more complex, you can use different fonts, colors, and background images.

Figure 1-1

Templates

Templates make it easier for developers to maintain a consistent look and feel across many pages, they help keep your code organized, and they move presentation logic out of your code, making both your PHP *and* HTML files more readable. There are a lot of different templating products available — some big (like Smarty, `http://smarty.php.net`) and some small (TinyButStrong, `www.tinybutstrong.com`). Each have their own benefits and drawbacks regardless if the solution is commercial, open source, or home-brewed. Sometimes the choice of which one to use will boil down to a matter of personal preference.

Speaking of personal preference, although I love the spirit of templating, I'm not a fan of most implementations. Despite all the benefits, modern templating systems complicate things. Some have their own special syntax to learn and almost all incur additional processing overhead. Truth be told, most projects don't need a dedicated template engine; PHP can be considered a template engine itself and can handle templating for even moderately large web projects with multiple developers if proper planning and organization is in place.

The setup that works best for me is to keep the core of my presentation in specific HTML files in a `templates` folder. This folder is usually outside of the web-accessible base (though the CSS, JavaScript and image files referenced in the HTML do need to be publically accessible) since I don't want a visitor or search engine to stumble upon a slew of content-less pages.

For now, here's a basic template that's suitable for the needs of this project:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
 <head>
  <title>
<?php
if (!empty($GLOBALS['TEMPLATE']['title']))
{
    echo $GLOBALS['TEMPLATE']['title'];
}
?>
</title>
  <link rel="stylesheet" type="text/css" href="css/styles.css"/>
<?php
if (!empty($GLOBALS['TEMPLATE']['extra_head']))
{
    echo $GLOBALS['TEMPLATE']['extra_head'];
}
?>
 </head>
 <body>
  <div id="header">
```

(continued)

(continued)

```
<?php
if (!empty($GLOBALS['TEMPLATE']['title']))
{
    echo $GLOBALS['TEMPLATE']['title'];
}
?>
  </div>
  <div id="content">
<?php
if (!empty($GLOBALS['TEMPLATE']['content']))
{
    echo $GLOBALS['TEMPLATE']['content'];
}
?>
  </div>
  <div id="footer">Copyright &copy;<?php echo date('Y'); ?></div>
  </div>
 </body>
</html>
```

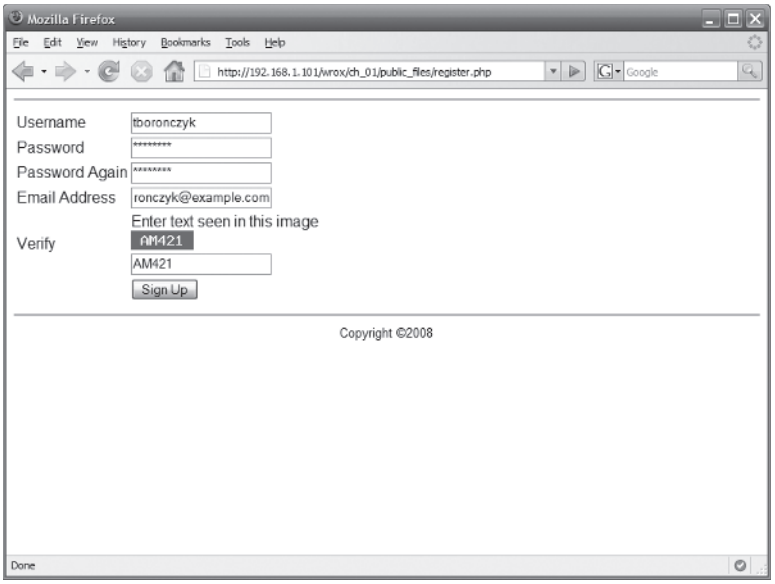
There should be some conventions in place to keep things sane. For starters, content will be stored in the `$GLOBALS` array in the requested script so it will be available in any scope within the included template file. I commonly use the following keys:

- `title` — Page title
- `description` — Page description
- `keywords` — Page keywords (the page's title, description and keywords can all be stored in a database)
- `extra_head` — A means to add additional HTML headers or JavaScript code to a page
- `content` — The page's main content

Occasionally I'll also include a menu or sidebar key depending on the project and planned layout, though your exact variables will depend on the template. So long as there are standard conventions written down and faithfully adhered to, a development team of any size can work successfully with such a template solution.

Registering a New User

With the directory structure laid out and enough of the support code written, the focus can now move to registering a new user. The following code can be saved in the `public_files` folder as `register.php`. Figure 1-2 shows the page viewed in a browser.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.101/wrox/ch_01/public_files/register.php
Google

Username	tboronczyk
Password	********
Password Again	********
Email Address	ronczyk@example.com
Verify	Enter text seen in this image AM421
	AM421
	Sign Up

Copyright ©2008

Figure 1-2

```
<?php
// include shared code
include '../lib/common.php';
include '../lib/db.php';
include '../lib/functions.php';
include '../lib/User.php';

// start or continue session so the CAPTCHA text stored in $_SESSION is
// accessible
session_start();
header('Cache-control: private');

// prepare the registration form's HTML
ob_start();
?>
<form method="post"
 action="<?php echo htmlspecialchars($_SEVER['PHP_SELF']); ?>">
 <table>
  <tr>
   <td><label for="username">Username</label></td>
   <td><input type="text" name="username" id="username"
    value="<?php if (isset($_POST['username']))
    echo htmlspecialchars($_POST['username']); ?>"/></td>
  </tr><tr>
   <td><label for="password1">Password</label></td>
   <td><input type="password" name="password1" id="password1"
    value=""/></td>
  </tr><tr>
```

(continued)

(continued)

```
      <td><label for="password2">Password Again</label></td>
      <td><input type="password" name="password2" id="password2"
       value=""/></td>
     </tr><tr>
      <td><label for="email">Email Address</label></td>
      <td><input type="text" name="email" id="email"
       value="<?php if (isset($_POST['email']))
       echo htmlspecialchars($_POST['email']); ?>"/></td>
     </tr><tr>
      <td><label for="captcha">Verify</label></td>
      <td>Enter text seen in this image<br/ >
      <img src="img/captcha.php?nocache=<?php echo time(); ?>" alt=""/><br />
      <input type="text" name="captcha" id="captcha"/></td>
     </tr><tr>
      <td> </td>
      <td><input type="submit" value="Sign Up"/></td>
      <td><input type="hidden" name="submitted" value="1"/></td>
     </tr><tr>
    </table>
   </form>
   <?php
   $form = ob_get_clean();

   // show the form if this is the first time the page is viewed
   if (!isset($_POST['submitted']))
   {
       $GLOBALS['TEMPLATE']['content'] = $form;
   }

   // otherwise process incoming data
   else
   {
       // validate password
       $password1 = (isset($_POST['password1'])) ? $_POST['password1'] : '';
       $password2 = (isset($_POST['password2'])) ? $_POST['password2'] : '';
       $password = ($password1 && $password1 == $password2) ?
           sha1($password1) : '';

       // validate CAPTCHA
       $captcha = (isset($_POST['captcha']) &&
           strtoupper($_POST['captcha']) == $_SESSION['captcha']);

       // add the record if all input validates
       if (User::validateUsername($_POST['username']) && password &&
           User::validateEmailAddr($_POST['email']) && $captcha)
       {
           // make sure the user doesn't already exist
           $user = User::getByUsername($_POST['username']);
           if ($user->userId)
           {
               $GLOBALS['TEMPLATE']['content'] = '<p><strong>Sorry, that ' .
                   'account already exists.</strong></p> <p>Please try a ' .
```

```
                'different username.</p>';
            $GLOBALS['TEMPLATE']['content'] .= $form;
        }
        else
        {
            // create an inactive user record
            $user = new User();
            $user->username = $_POST['username'];
            $user->password = $password;
            $user->emailAddr = $_POST['email'];
            $token = $user->setInactive();

            $GLOBALS['TEMPLATE']['content'] = '<p><strong>Thank you for ' .
                'registering.</strong></p> <p>Be sure to verify your ' .
                'account by visiting <a href="verify.php?uid=' .
                $user->userId . '&token=' . $token . '">verify.php?uid=' .
                $user->userId . '&token=' . $token . '</a></p>';
         }
    }
    // there was invalid data
    else
    {
        $GLOBALS['TEMPLATE']['content'] .= '<p><strong>You provided some ' .
            'invalid data.</strong></p> <p>Please fill in all fields ' .
            'correctly so we can register your user account.</p>';
        $GLOBALS['TEMPLATE']['content'] .= $form;
    }
}

// display the page
include '../templates/template-page.php';
?>
```

The first thing register.php does is import the shared code files it depends on. Some programmers prefer to place all the include statements in one common header file and include that for shorter code. Personally, however, I prefer to include them individually as I find it easier to maintain.

Other programmers may use chdir() to change PHP's working directory so they don't have to repeatedly backtrack in the file system to include a file. Again, this is a matter of personal preference. Be careful with this approach, however, when targeting older installations of PHP that use safe mode. chdir() may fail without generating any kind of error message if the directory is inaccessible.

```
<?php
// include shared code
chdir('../');
include 'lib/common.php';
include 'lib/db.php';
include 'lib/functions.php';
include 'lib/User.php';
...
?>
```

After importing the shared code files I call `session_start()`. HTTP requests are stateless, which means the web server returns each page without tracking what was done before or anticipating what might happen next. PHP's session tracking gives you an easy way to maintain state across requests and carry values from one request to the next. A session is required for keeping track of the CAPTCHA value generated by `captcha.php`.

I like to use output buffering when preparing large blocks of HTML such as the registration form, for greater readability. Others may prefer to maintain a buffer variable and repeatedly append to it throughout the script, like so:

```
<?php
$GLOBALS['TEMPLATE']['content'] = '<form action="'.
     htmlspecialchars(currentFile()) . '" method="post">';
$GLOBALS['TEMPLATE']['content'] .= '<table>';
$GLOBALS['TEMPLATE']['content'] .= '<tr>';
$GLOBALS['TEMPLATE']['content'] .= '<td><label for="username">Username</label>' .
 '</td>';
...
?>
```

I find that approach becomes rather cumbersome relatively fast. With output buffering, all I need to do is start the capturing with `ob_start()`, retrieve the buffer's contents with `ob_get_contents()`, and stop capturing with `ob_end_clean()`. `ob_get_clean()` combines `ob_get_contents()` and `ob_end_clean()` in one function call. It's also easier for the engine to fall in and out of PHP mode so such code with large blocks of output would theoretically run faster than with the buffer concatenation method.

No `$_POST` values should be received the first time a user views the page so the code just outputs the registration form. When the user submits the form, the `$_POST['submitted']` variable is set and it knows to start processing the input.

The validation code to check the use rname and password are part of the `User` class. The two password values are compared against each other and then the password's hash is saved for later storage. Finally, the user's CAPTCHA input is checked with what was previously stored in the session by `captcha.php`. If everything checks out, the record is added to the database.

The `verify.php` script referenced in the HTML code is responsible for taking in a user ID and activation token, checking the corresponding values in the database, and then activating the user's account. It must be saved in the publically accessible directory as well.

```
<?php
// include shared code
include '../lib/common.php';
include '../lib/db.php';
include '../lib/functions.php';
include '../lib/User.php';

// make sure a user id and activation token were received
if (!isset($_GET['uid']) || !isset($_GET['token']))
{
    $GLOBALS['TEMPLATE']['content'] = '<p><strong>Incomplete information ' .
        'was received.</strong></p> <p>Please try again.</p>';
        include '../templates/template-page.php';
        exit();
    }

    // validate userid
    if (!$user = User::getById($_GET['uid']))
    {
        $GLOBALS['TEMPLATE']['content'] = '<p><strong>No such account.</strong>' .
            '</p> <p>Please try again.</p>';
    }
    // make sure the account is not active
    else
    {
        if ($user->isActive)
        {
            $GLOBALS['TEMPLATE']['content'] = '<p><strong>That account ' .
                'has already been verified.</strong></p>';
        }
        // activate the account
        else
        {
            if ($user->setActive($_GET['token']))
            {
                $GLOBALS['TEMPLATE']['content'] = '<p><strong>Thank you ' .
                    'for verifying your account.</strong></p> <p>You may ' .
                    'now <a href="login.php">login</a>.</p>';
            }
            else
            {
                $GLOBALS['TEMPLATE']['content'] = '<p><strong>You provided ' .
                    'invalid data.</strong></p> <p>Please try again.</p>';
            }
        }
    }

    // display the page
    include '../templates/template-page.php';
    ?>
```

E-mailing a Validation Link

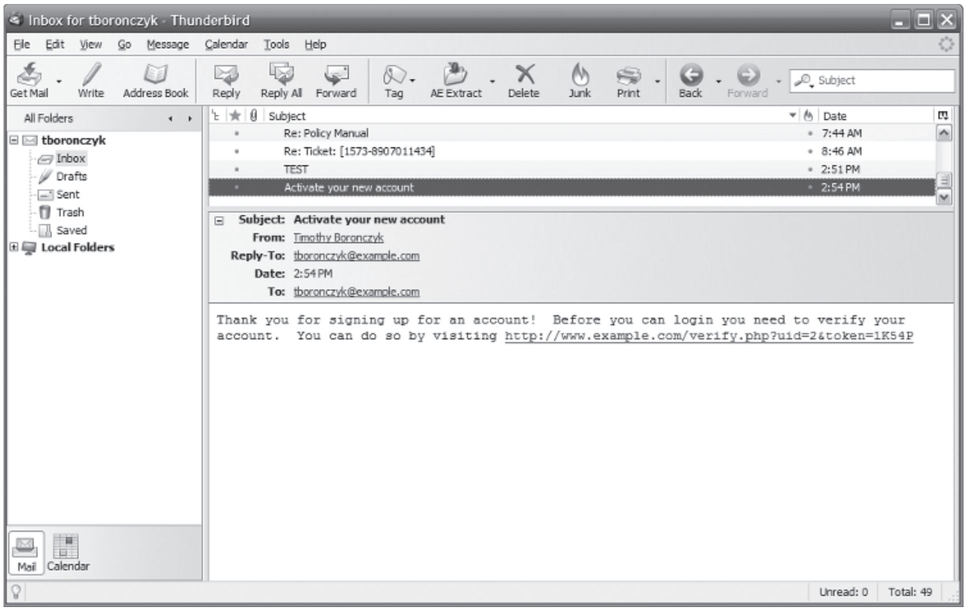
Right now `register.php` provides a direct link to verify the account, though in a production environment it's typical to send the link in an e-mail to the address provided. The hope is that legitimate users will supply legitimate e-mail accounts and actively confirm their accounts, and bulk spammers wouldn't.

The `mail()` function is used to send e-mails from within PHP. The first argument is the user's e-mail address, the second is the e-mail's subject, and the third is the message. The use of `@` to suppress warning messages is generally discouraged, though in this case it is necessary because `mail()` will return false *and* generate a warning if it fails.

The code you integrate into `register.php` to send a message instead of displaying the validation link in the browser window might look something like this:

```
<?php
...
// create an inactive user record
$user = new User();
$user->username = $_POST['username'];
$user->password = $password;
$user->emailAddr = $_POST['email'];
$token = $user->setInactive();

$message = 'Thank you for signing up for an account!  Before you '.
    ' can login you need to verify your account. You can do so ' .
    'by visiting http://www.example.com/verify.php?uid=' .
    $user->userId . '&token=' . $token . '.';

if (@mail($user->emailAddr, 'Activate your new account', $message))
{
    $GLOBALS['TEMPLATE']['content'] = '<p><strong>Thank you for ' .
        'registering.</strong></p> <p>You will be receiving an ' .
        'email shortly with instructions on activating your ' .
        'account.</p>';
}
else
{
    $GLOBALS['TEMPLATE']['content'] = '<p><strong>There was an ' .
        'error sending you the activation link.</strong></p> ' .
        '<p>Please contact the site administrator at <a href="' .
        'mailto:admin@example.com">admin@example.com</a> for ' .
        'assistance.</p>';
}
...
?>
```

Figure 1-3 shows the confirmation message sent as an e-mail viewed in an e-mail program.


Inbox for tboronczyk - Thunderbird
File Edit View Go Message Calendar Tools Help
Get Mail Write Address Book Reply Reply All Forward Tag AE Extract Delete Junk Print Back Forward
Subject
All Folders
tboronczyk
Inbox
Drafts
Sent
Trash
Saved
Local Folders
Mail Calendar
Unread: 0 Total: 49

Subject	Date
Re: Policy Manual	7:44 AM
Re: Ticket: [1573-8907011434]	8:46 AM
TEST	2:51 PM
Activate your new account	2:54 PM

Subject: **Activate your new account**
From: Timothy Boronczyk
Reply-To: tboronczyk@example.com
Date: 2:54 PM
To: tboronczyk@example.com

```
Thank you for signing up for an account!  Before you can login you need to verify your
account.  You can do so by visiting http://www.example.com/verify.php?uid=2&token=1K54P
```

Figure 1-3

Sending the message as a plain text e-mail is simple, while sending an HTML-formatted message is a bit more involved. Each have their own merits: plain text messages are more accessible and less likely to get blocked by a user's spam filter while HTML-formatted messages appear friendlier, less sterile and can have clickable hyperlinks to make validating the account easier.

An HTML-formatted e-mail message might look like this:

```
<html>
<p>Thank you for signing up for an account!</p>
<p>Before you can login you need to verify your account. You can do so by
visiting <a href="http://www.example.com/verify.php?uid=###&amp;token=xxxxx">
http://www.example.com/verify.php?uid=###&amp;token=xxxxx</a>.</p>
<p>If your mail program doesn't allow you to click on hyperlinks in a
message, copy it and paste it into the address bar of your web browser to
visit the page.</p>
</html>
```

However, if you sent it as the previous example then the e-mail would still be received as plain text even though it contains HTML markup. The proper `MIME` and `Content-Type` headers also need to be sent as well to inform the e-mail client how to display the message. These additional headers are given to `mail()`'s optional fourth parameter.

```
<?php
// assume the formatted message is stored as $html_message

// formatted mail requires a MIME and Content-Type header
$headers = array('MIME-Version: 1.0',
                 'Content-Type: text/html; charset="iso-8859-1"');

// additional headers are supplied as the 4th argument to mail()
mail($user->emailAddr, 'Please activate your new account', $html_message,
    join("\n", $headers));
?>
```

It's possible to have the best of both e-mail worlds by sending a mixed e-mail message. A mixed e-mail contains both plain-text and HTML-formatted messages and then it becomes the mail client's job to decide which portion it should display. Here's an example of such a multi-part message:

```
--==A.BC_123_XYZ_678.9
Content-Type: text/plain; charset="iso-8859-1"

Thank you for signing up for an account!

Before you can login you need to verify your account. You can do so by visiting
http://www.example.com/verify.php?uid=##&token=xxxxx.

--==A.BC_123_XYZ_678.9
Content-Type: text/plain; charset="iso-8859-1"

<html>
<p>Thank you for signing up for an account!</p>
<p>Before you can login you need to verify your account. You can do so by
visiting <a href="http://www.example.com/verify.php?uid=###&amp;token=xxxxx">
http://www.example.com/verify.php?uid=###&amp;token=xxxxx</a>.</p>
<p>If your mail program doesn't allow you to click on hyperlinks in a
message, copy it and paste it into the address bar of your web browser to
visit the page.</p>
</html>

--==A.BC_123_XYZ_678.9--
```

The correct headers to use when sending the message would be:

```
MIME-Version: 1.0
Content-Type: multipart/alternative; boundary="==A.BC_123_XYZ_678.9"
```

Note that a special string is used to mark boundaries of different message segments. There's no significance to `==A.BC_123_XYZ_678.9` as I've used — it just needs to be random text which doesn't appear in the body of any of the message parts. When used to separate message blocks, the string is preceded by two dashes and is followed by a blank line. Trailing dashes mark the end of the message.

Logging In and Out

With the ability to create new user accounts and verify them as belonging to a real people with valid e-mail addresses in place, the next logical step is to provide a mechanism for these users to log in and out. Much of the dirty work tracking the session will be done by PHP so all you need to do is store some identifying information in `$_SESSION`. Save this code as `login.php`.

```
<?php
// include shared code
include '../lib/common.php';
include '../lib/db.php';
include '../lib/functions.php';
include '../lib/User.php';

// start or continue the session
session_start();
header('Cache-control: private');

// perform login logic if login is set
if (isset($_GET['login']))
{
    if (isset($_POST['username']) && isset($_POST['password']))
    {
        // retrieve user record
        $user = (User::validateUsername($_POST['username'])) ?
            User::getByUsername($_POST['username']) : new User();

        if ($user->userId && $user->password == sha1($_POST['password']))
        {
            // everything checks out so store values in session to track the
            // user and redirect to main page
            $_SESSION['access'] = TRUE;
            $_SESSION['userId'] = $user->userId;
            $_SESSION['username'] = $user->username;
            header('Location: main.php');
        }
        else
        {
            // invalid user and/or password
            $_SESSION['access'] = FALSE;
            $_SESSION['username'] = null;
            header('Location: 401.php');
        }
    }
    // missing credentials
    else
    {
        $_SESSION['access'] = FALSE;
        $_SESSION['username'] = null;
        header('Location: 401.php');
    }
    exit();
}
```

(continued)

(continued)

```
// perform logout logic if logout is set
// (clearing the session data effectively logsout the user)
else if (isset($_GET['logout']))
{
    if (isset($_COOKIE[session_name()]))
    {
        setcookie(session_name(), '', time() - 42000, '/');
    }

    $_SESSION = array();
    session_unset();

    session_destroy();
}

// generate login form
ob_start();
?>
<form action="<?php echo htmlspecialchars($_SERVER['PHP_SELF']); ?>?login"
 method="post">
 <table>
  <tr>
   <td><label for="username">Username</label></td>
   <td><input type="text" name="username" id="username"/></td>
  </tr><tr>
   <td><label for="password">Password</label></td>
   <td><input type="password" name="password" id="password"/></td>
  </tr><tr>
   <td> </td>
   <td><input type="submit" value="Log In"/></td>
  </tr>
 </table>
</form>
<?php
$GLOBALS['TEMPLATE']['content'] = ob_get_clean();

// display the page
include '../templates/template-page.php';
?>
```

The code encapsulates the logic to both process logins and logouts by passing a parameter in the page address. Submitting a login form to `login.php?login` would processes the login logic. Linking to `login.php?logoff` will effectively log out the user by clearing all session data. The login form is shown in Figure 1-4.

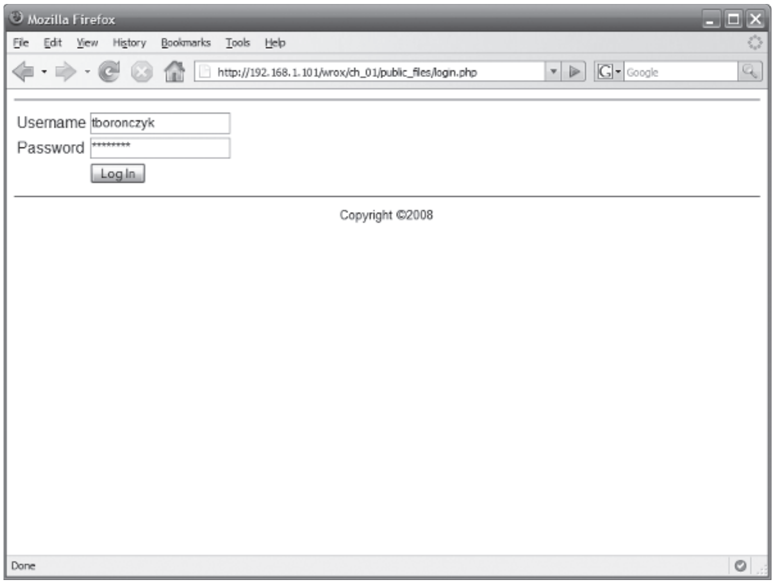

Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.101/wrox/ch_01/public_files/login.php
Google
Username tboronczyk
Password ********
Log In
Copyright ©2008
Done

Figure 1-4

To log a user in, the script accepts a username and password. The supplied username is passed to the `getByUsername()` method so the record can be retrieved from the database and the supplied password is hashed for comparison. If the credentials match, the user provided the correct username and password and is logged in by storing identifying information in the session and redirecting the browser to the main page. The session is cleared and the user is redirected to an error page (`404.php`) if authentication fails.

The script outputs HTML code for a login form if called without any parameters. This is convenient if you want to link to it from another page, or redirect back to the login form from the error page. But you're not restricted to using this form. Because an exit statement has been strategically placed after the login code, you can use the script to process any login form, whether it's in a page template or elsewhere. Just remember to pass the login parameter in the address.

The user is redirected to `401.php` if the login is not successful:

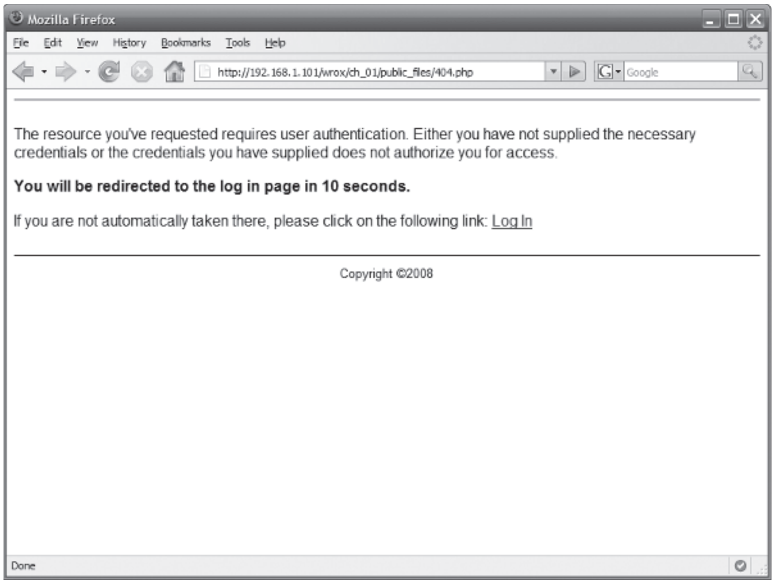
```
<?php
// include shared code
include '../lib/common.php';

// start or join the session
session_start();
header('Cache-control: private');

// issue 401 error if the user has not been authenticated
if (!isset($_SESSION['access']) || $_SESSION['access'] != TRUE)
{
    header('HTTP/1.0 401 Authorization Error');
```

(continued)

(continued)

```
        ob_start();
    ?>
    <script type="text/javascript">
    window.seconds = 10;
    window.onload = function()
    {
        if (window.seconds != 0)
        {
            document.getElementById('secondsDisplay').innerHTML = '' +
                window.seconds + ' second' + ((window.seconds > 1) ? 's' : '');
            window.seconds--;
            setTimeout(window.onload, 1000);
        }
        else
        {
            window.location = 'login.php';
        }
    }
    </script>
    <?php
        $GLOBALS['TEMPLATE']['extra_head'] = ob_get_contents();
        ob_clean();

    ?>
    <p>The resource you've requested requires user authentication. Either you have
    not supplied the necessary credentials or the credentials you have supplied
    do not authorize you for access.</p>

    <p><strong>You will be redirected to the login page in
    <span id="secondsDisplay">10 seconds</span>.</strong></p>

    <p>If you are not automatically taken there, please click on the following
    link: <a href="login.php">Log In</a></p>
    <?php
        $GLOBALS['TEMPLATE']['content'] = ob_get_clean();

        include '../templates/template-page.php';
        exit();
    }
    ?>
```

The 401 response is shown in Figure 1-5. The primary responsibility of the script is to send an authorization error to the browser and redirect the user back to the login form (the response code for an HTTP authorization error is `401`). Because `session_start()` is called and `$_SESSION['access']` is checked, the error is only sent if the user hasn't been authenticated. To protect any page you only need to include this file at the top of the document. If the user has logged in then he or she will see the intended content.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.101/wrox/ch_01/public_files/404.php
Google

The resource you've requested requires user authentication. Either you have not supplied the necessary credentials or the credentials you have supplied does not authorize you for access.

You will be redirected to the log in page in 10 seconds.

If you are not automatically taken there, please click on the following link: Log In

Copyright ©2008

Done

Figure 1-5

There are different ways to perform a client-side redirect for a user. Here I've mixed a little bit of JavaScript code in with the script's output to count down 10 seconds (1,000 microseconds) — enough time for the user to see he or she is being denied access — and actively updates the time remaining until it performs the redirect by setting the `window.location` property. Another way to redirect the client is by outputting an HTML meta element:

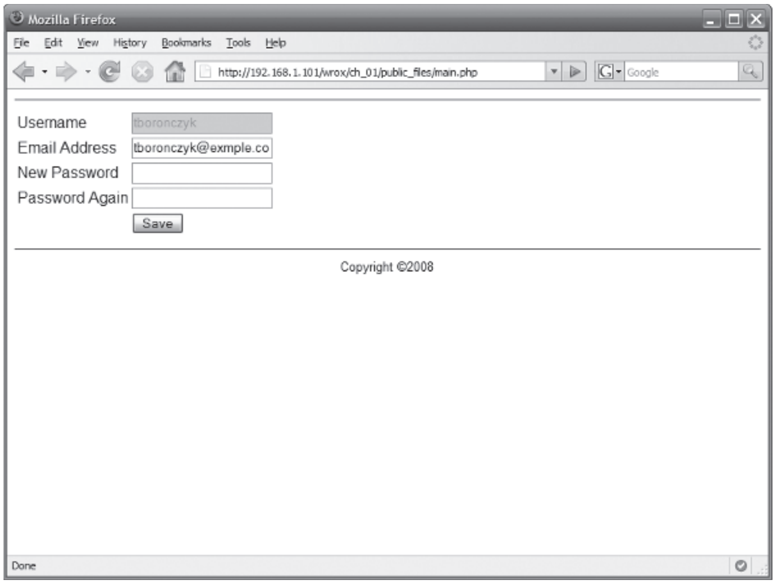
```
<meta http-equiv="refresh"
 content="10;URL=http://www.example.com/login.php" />
```

Regardless of the method you choose to employ, you should always provide a link in case the browser doesn't redirect the user properly.

Changing Information

People may want to change their names, passwords, and e-mail addresses and it makes sense to allow this in your applications. I've already shown you an example of changing a user record earlier when I first discussed the `User` class. It's the same process here — simply set the object's properties to new values and call the `save()` method.

I've saved this code as `main.php` for the simple fact that `login.php` redirects the user to `main.php` after a successful login. In your own implementation, you may want to name it something like `editmember.php` and have `main.php` offer some interesting content instead. Either way, the form is shown in Figure 1-6.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.101/wrox/ch_01/public_files/main.php
Google
Done

Username	tboronczyk
Email Address	tboronczyk@exmple.co
New Password	
Password Again	
	Save

Copyright ©2008

Figure 1-6

```
<?php
// include shared code
include '../lib/common.php';
include '../lib/db.php';
include '../lib/functions.php';
include '../lib/User.php';

// 401 file referenced since user should be logged in to view this page
include '401.php';

// generate user information form
$user = User::getById($_SESSION['userId']);

ob_start();
?>
<form action="<?php echo htmlspecialchars($_SERVER['PHP_SELF']); ?>"
 method="post">
 <table>
  <tr>
   <td><label>Username</label></td>
   <td><input type="text" name="username" disabled="disabled"
    readonly="readonly"value="<?php echo $user->username; ?>"/></td>
  </tr><tr>
   <td><label for="email">Email Address</label></td>
   <td><input type="text" name="email" id="email"
    value="<?php echo (isset($_POST['email']))? htmlspecialchars(
$_POST['email']) : $user->emailAddr; ?>"/></td>
  </tr><tr>
```

```
     <td><label for="password">New Password</label></td>
     <td><input type="password" name="password1" id="password1"/></td>
    </tr><tr>
     <td><label for="password2">Password Again</label></td>
     <td><input type="password" name="password2" id="password2"/></td>
    </tr><tr>
    <td> </td>
     <td><input type="submit" value="Save"/></td>
     <td><input type="hidden" name="submitted" value="1"/></td>
    </tr><tr>
  </table>
</form>
<?php
$form = ob_get_clean();

// show the form if this is the first time the page is viewed
if (!isset($_POST['submitted']))
{
    $GLOBALS['TEMPLATE']['content'] = $form;
}
// otherwise process incoming data
else
{
    // validate password
    $password1 = (isset($_POST['password1']) && $_POST['password1']) ?
        sha1($_POST['password1']) : $user->password;
    $password2 = (isset($_POST['password2']) && $_POST['password2']) ?
        sha1($_POST['password2']) : $user->password;
    $password = ($password1 == $password2) ? $password1 : '';

    // update the record if the input validates
    if (User::validateEmailAddr($_POST['email']) && $password)
    {
        $user->emailAddr = $_POST['email'];
        $user->password = $password;
        $user->save();

        $GLOBALS['TEMPLATE']['content'] = '<p><strong>Information ' .
            'in your record has been updated.</strong></p>';
    }
    // there was invalid data
    else
    {
        $GLOBALS['TEMPLATE']['content'] .= '<p><strong>You provided some ' .
            'invalid data.</strong></p>';
        $GLOBALS['TEMPLATE']['content'] .= $form;
    }
}

// display the page
include '../templates/template-page.php';
?>
```

You may want to modify the code to verify the user's password before processing any changes to his or her user record. It's also common to set the account inactive and reverify the e-mail address if the user updates it.

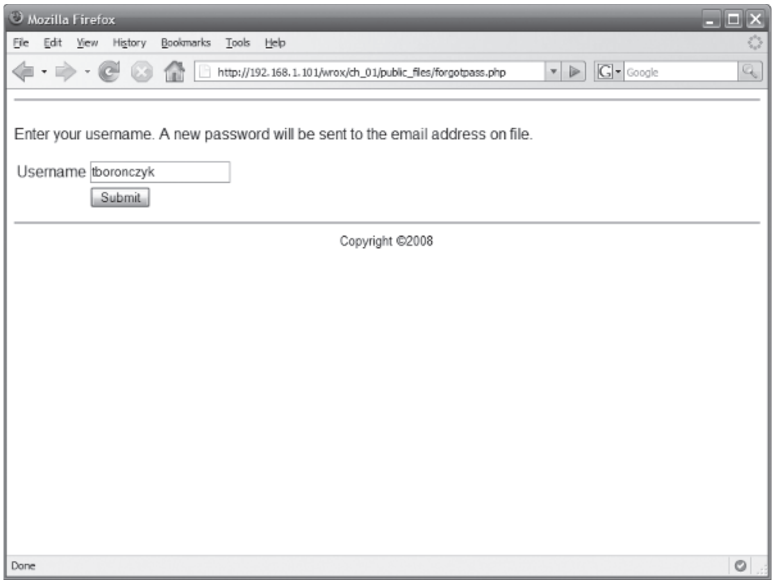
Forgotten Passwords

Sometimes users will forget their passwords and not be able to log in. Since the actual password is never stored, there's no way to retrieve it for them. Instead, a new password must be generated and sent to the user's e-mail address on file. Code to accomplish this can be saved as `forgotpass.php`:

```
<?php
// include shared code
include '../lib/common.php';
include '../lib/db.php';
include '../lib/functions.php';
include '../lib/User.php';

// construct password request form HTML
ob_start();
?>
<form action="<?php echo htmlspecialchars($_SEVER['PHP_SELF']); ?>"
 method="post">
<p>Enter your username. A new password will be sent to the email address on
 file.</p>
<table>
<tr>
 <td><label for="username">Username</label></td>
 <td><input type="text" name="username" id="username"
  value="<?php if (isset($_POST['username']))
  echo htmlspecialchars($_POST['username']); ?>"/></td>
</tr><tr>
 <td> </td>
 <td><input type="submit" value="Submit"/></td>
 <td><input type="hidden" name="submitted" value="1"/></td>
</tr><tr>
</table>
</form>
<?php
$form = ob_get_clean();

// show the form if this is the first time the page is viewed
if (!isset($_POST['submitted']))
{
    $GLOBALS['TEMPLATE']['content'] = $form;
}
// otherwise process incoming data
```

```
else
{
    // validate username
    if (User::validateUsername($_POST['username']))
    {
        $user = User::getByUsername($_POST['username']);
        if (!$user->userId)
        {
            $GLOBALS['TEMPLATE']['content'] = '<p><strong>Sorry, that ' .
                'account does not exist.</strong></p> <p>Please try a ' .
                'different username.</p>';
            $GLOBALS['TEMPLATE']['content'] .= $form;
        }
        else
        {
            // generate new password
            $password = random_text(8);

            // send the new password to the email address on record
            $message = 'Your new password is: ' . $password;
            mail($user->emailAddr, 'New password', $message);

            $GLOBALS['TEMPLATE']['content'] = '<p><strong>A new ' .
                'password has been emailed to you.</strong></p>';

            // store the new password
            $user->password = $password;
            $user->save();
        }
    }
    // there was invalid data
    else
    {
        $GLOBALS['TEMPLATE']['content'] .= '<p><strong>You did not ' .
            'provide a valid username.</strong></p> <p>Please try ' .
            'again.</p>';
        $GLOBALS['TEMPLATE']['content'] .= $form;
    }
}

// display the page
include '../templates/template-page.php';
?>
```

Figure 1-6 shows the page viewed in a web browser.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.101/wrox/ch_01/public_files/forgotpass.php
Google

Enter your username. A new password will be sent to the email address on file.

Username tboronczyk

Submit

Copyright ©2008

Done

Figure 1-7

Summary

Voilà! You now have a basic user-registration framework to extend anyway you like. Perhaps you want to collect more information from your users, such as their cell phone numbers for SMS messaging, mailing addresses, or even internet messenger screen names.

You've also learned how to establish a well-organized directory structure for your applications and seen some of the benefits of writing reusable code. The directory layout and many of the support files will be used throughout the rest of this book.

In the next chapter you'll build upon what you've written so far and program a basic community bulletin board, more commonly known as a *forum*.