WORKING WITH IMAGES

This chapter shows you how to create and manipulate images such as GIFs and JPEGs. You won't be able to write scripts for heavy-duty image manipulation or processing such as online character recognition (your webserver limits the amount of computation that one script can do), but there are many small image operations that are very handy when developing a website.

Creating a CAPTCHA (Security) Image

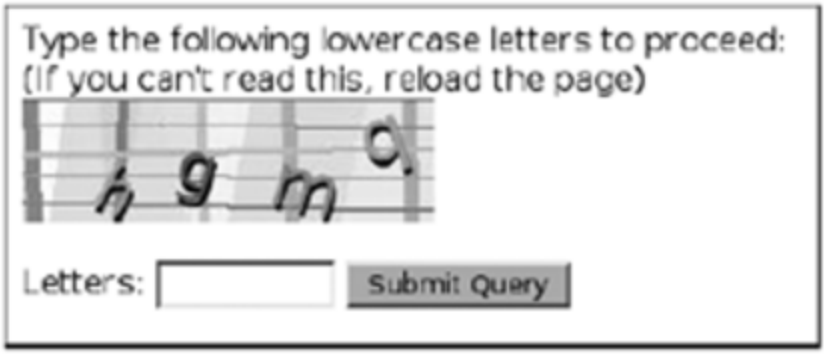
Because many examples in this book use cURL to connect to and interact with websites, you know how easy it is to automate interactions with websites. You also know that if you have a site feature that you want no one but a real human to work with, you need to protect it. Otherwise, you may (at the very least) find your site completely overrun by spam comments. One way around this problem is to dynamically draw an image containing text that a user must type in order to post a comment (or perform some other human-only action, such as voting or rating). If you get the correct text from the user, the user is probably human.

This sort of test is named CAPTCHA, for Completely Automated Public Turing Test to Tell Computers and Humans Apart. It has two drawbacks. First, if you're blind, you can't read the text from the CAPTCHA image. Visually impaired people can use the Internet with the help of voice software to read the pages aloud, but this software can't interpret images without the descriptive `alt` attribute. A CAPTCHA locks out these users.

Furthermore, a CAPTCHA doesn't always work. Spammers are always looking for ways around your roadblocks. One of the more creative ways is linking the CAPTCHA image to a spammer-owned porn site with a label of INPUT THIS PHRASE AND SEE NAKED PEOPLE! Then some lonely guy inputs the password, which is then passed back to your site.

With these two shortcomings in mind, you should use CAPTCHAs sparingly and weigh them against alternatives such as the Akismet scanner. The CAPTCHA presented here is simple (see Figure 1 for a screenshot), but it is effective. To use it, your PHP must have the GD graphics library with Freetype support. To see if you have GD support, use `phpinfo()`. The output contains a section on GD with Freetype information if everything is okay.

Figure 1. A CAPTCHA interface



You also need one or more fonts in the same directory as the script. It's best to go with fonts that are at least halfway readable.

There are two pieces to this system:

1. The main part is a script that creates CAPTCHA images. These images have irregular backgrounds, with strange swaths of color and lines running both horizontally and vertically. Then the script picks a passphrase, stores it in the `$_SESSION['tt_pass']` variable, and draws each letter in the passphrase differently. Finally, it sends the image output.
2. The second piece is a script to compare a CAPTCHA form's input with the `$_SESSION['tt_pass']` variable. The script here is a simple demonstration, but because the image-generator script does most of the work, you won't need to do much to make it fit your own needs.

Let's start with the image generator. The first step is to set several configuration variables for the length of the passphrase, the image size, and the place to look for fonts:

```
<?php
/* captcha_image.php */

/* Passphrase length */
$pass_length = 4;

/* Image dimensions */
$width = 200;
$height = 60;

/* TTF font path */
$font_path = dirname(__FILE__);
```

Now let's create the passphrase. Unlike most passwords, this doesn't need to be particularly challenging: There's no need to annoy your users more than you have to.

```
session_start();

/* Create a passphrase. */
$passwd = "";
$i = 0;
while ($i < $pass_length) {
    $passwd .= chr(rand(97, 122));
    $i++;
}
```

We'll put the passphrase in a session variable. Therefore, it never appears in any pages in any form of encoding. This avoids the need to come up with an encoding that someone will just break.

```
/* Store passphrase. */
$_SESSION["tt_pass"] = $passwd;
```

Before drawing anything, we want to make sure that we have some fonts to use for the image. This code creates a list of fonts in the current directory. If you'd like to make this part faster, you can set the `$font` array to a set of TrueType font filenames:

```
/* Get a list of available fonts. */
$fonts = array();

if ($handle = opendir($font_path)) {
    while (false !== ($file = readdir($handle))) {
        /* Look for TTF fonts. */
        if (substr(strtolower($file), -4, 4) == '.ttf') {
            $fonts[] = $font_path . '/' . $file;
        }
    }
}

if (count($fonts) < 1)  {
    die("No fonts found!");
}
```

The client has to know what kind of image you're sending. This script creates a JPEG image, so it sends that MIME type. In addition, we don't want the browser to cache the image, so we'll send several cache-invalidating headers:

```
/* Image header */
header("Content-Type: image/jpeg");
/* Invalidate cache. */
header("Expires: Mon, 01 Jul 1998 05:00:00 GMT");
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
```

```
header("Cache-Control: no-store, no-cache, must-revalidate");
header("Cache-Control: post-check=0, pre-check=0", false);
header("Pragma: no-cache");
```

With all of the preparatory work complete, we can now start creating the image. GD functions begin with an `image` prefix. To initialize a color image canvas, use `imagecreatetruecolor()`.

```
/* Create image. */
$img = imagecreatetruecolor($width, $height);
```

The first thing we'll draw is a rectangle to cover the entire background. To draw anything, you need to allocate a color in the image with `imagecreatetruecolor()`. The second, third, and fourth parameters to this function are the red, green, and blue values of the color, respectively. The value is an integer between 0 and 255, with 0 being the darkest and 255 being the lightest. The following statement creates a random pastel color:

Code View:
```
/* Fill background with a random shade of pastel. */
$bg = imagecolorallocate($img, rand(210,255), rand(210,255), rand(210,255));
```

Now let's draw that rectangle with the `imagefilledrectangle()` function. The rectangle is parallel to the image axes, so you can define it by two points: (0, 0) and (`$width, $height`):

```
imagefilledrectangle($img, 0, 0, $width, $height, $bg);
```

To convolute the background some more, we'll draw several vertically oriented polygons across the canvas. The following loop creates the polygons with the `imagefilledpolygon()` function. This is different than drawing a rectangle because a polygon can have three or more sides in any orientation, so you need to provide an array of points as a parameter (here, `$poly_points`) instead of two sets of coordinates.

Try not to get sucked into the details of this loop. It is not a terribly important part of this script.

Code View:
```
/* Make the background jaggedy by creating different-colored
   four-cornered polygons to cover the background area. */

/* Create swaths between 10 and 30 pixels wide across the image. */
$right = rand(10, 30);
$left = 0;
while ($left < $width) {
    $poly_points = array(
        $left, 0,                                   /* Upper left */
        $right, 0,                                  /* Upper right */
```

```
        rand($right-25, $right+25), $height,      /* Lower right */
        rand($left-15, $left+15), $height);       /* Lower left */

    /* Create the polygon, using four points from the array. */
    $c = imagecolorallocate($img, rand(210,255), rand(210,255),
rand(210,255));
    imagefilledpolygon($img, $poly_points, 4, $c);

    /* Advance to the right edge. */
    $random_amount = rand(10, 30);
    $left += $random_amount;
    $right += $random_amount;
}
```

To further thwart those who might use OCR software to try to break this system, let's draw some random horizontal and vertical lines across the screen. To mix it up even more, we'll define a different color range for the lines for each image. By choosing random lower and upper bounds for the colors, the lines may or may not vary in intensity and will vary in consistency from image to image.

```
/* Choose base value range for vertical and horizontal lines. */
$c_min = rand(120, 185);
$c_max = rand(195, 280);
```

To draw the vertical lines, start at the left side and pick a thickness and a slight offset to tilt the line. Then pick a color (using the preceding bounds), draw the line as a polygon, and advance to the right by yet another random amount:

```
/* Draw random vertical lines across the width. */
$left = 0;
while ($left < $width) {
    $right = $left + rand(3, 7);
    $offset = rand(-3, 3);                        /* Offset for angle */

    $line_points = array(
        $left, 0,                                 /* Upper left */
        $right, 0,                                /* Upper right */
        $right + $offset, $height,                /* Lower right */
        $left + $offset, $height);                /* Lower left */

    $pc = imagecolorallocate($img, rand($c_min, $c_max),
                                   rand($c_min, $c_max),
                                   rand($c_min, $c_max));
    imagefilledpolygon($img, $line_points, 4, $pc);

    /* Advance to the right. */
    $left += rand(20, 60);
}
```

Use the same procedure to create the horizontal lines:

```
/* Create random horizontal lines across the height. */
$top = 0;
while ($top < $height) {
    $bottom = $top + rand(1, 4);
    $offset = rand(-6, 6);                  /* Offset for angle */

    $line_points = array(
        0, $top,                            /* Upper left */
        0, $bottom,                         /* Lower left */
        $width, $bottom + $offset,          /* Lower right */
        $width, $top + $offset);            /* Upper right */
    $pc = imagecolorallocate($img, rand($c_min, $c_max),
                                   rand($c_min, $c_max),
                                   rand($c_min, $c_max));
    imagefilledpolygon($img, $line_points, 4, $pc);
    $top += rand(8, 15);
}
```

We're finally ready for the passphrase characters. Before drawing, determine roughly how far apart you want the letters to appear, and then set a variable to the leftmost character position:

```
/* Determine character spacing. */
$spacing = $width / (strlen($passwd)+2);

/* Initial x coordinate */
$x = $spacing;
```

Now, iterate through the characters and pick a slightly different offset, angle, size, and font each time:

```
/* Draw each character. */
for ($i = 0; $i < strlen($passwd); $i++) {
    $letter = $passwd[$i];
    $size = rand($height/3, $height/2);
    $rotation = rand(-30, 30);
    /* Random y position with room for character descenders */
    $y = rand($height * .90, $height - $size - 4);

    /* Pick a random font. */
    $font = $fonts[array_rand($fonts)];
```

The letters will be very difficult to read without some sort of outlining. You can create a shadow color by dividing the original color values by 3:

```
        /* Pick a color for the letter. */
        $r = rand(100, 255); $g = rand(100, 255); $b = rand(100, 255);

        /* Create letter and shadow colors. */
```

```
        $color = imagecolorallocate($img, $r, $g, $b);
        $shadow = imagecolorallocate($img, $r/3, $g/3, $b/3);
```

Use `imagettftext()` to draw the shadow and then the letter, and advance to the next character:

Code View:

```
    /* Draw shadow, then letter. */
    imagettftext($img, $size, $rotation, $x, $y, $shadow, $font, $letter);
    imagettftext($img, $size, $rotation, $x-1, $y-3, $color, $font, $letter);

    /* Advance across the canvas. */
    $x += rand($spacing, $spacing * 1.5);
}
```

Once this loop is complete, you're ready to send the image to the client. Use `imagejpeg()` to do so, and then deallocate its memory with `imagedestroy()`:

```
imagejpeg($img);          /* Send image output. */
imagedestroy($img);       /* Free image memory. */

?>
```

The other piece of the CAPTCHA system is a small piece of code to display the form, embed an image with the preceding script, and verify that the form input is the passphrase the image generator created. Because the image script does all of the hard work, you'll be able to figure out the following without comment:

```
<?php

session_start();

/* Look for a submitted password. */
if ($_REQUEST["tt_pass"]) {
    if ($_REQUEST["tt_pass"] == $_SESSION["tt_pass"]) {
        echo "passphrase correct.";
    } else {
        echo "passphrase incorrect.";
    }
    exit(0);
}

/* By default, send the form. */

print '<form action="' . $_SERVER['PHP_SELF'] . '" method="post">';
?>

Type the following lowercase letters to proceed:<br />
(If you can't read this, reload the page)<br />
```

```
<img src="captcha_image.php"><br /><br />
Letters: <input name="tt_pass" type="text" size="10" maxlength="10">
<input type="submit">
</form>
```

Obviously, you'll need to incorporate this script into your code more tightly, but one of the best aspects of this system is that it's relatively self contained. There are several things you could do to tighten it up a little, such as deleting the stored passphrase once it is verified (so it can be used only once). But if you find yourself needing much more, you might have a larger problem that you need to solve with a login system.

Creating Thumbnail Images

When you allow users to upload images to your site, you often need a small preview image (called a thumbnail) for inclusion on larger pages and image gallery browsing. This section shows you a function called `mkthumb()` that creates thumbnails with the help of GD. To use the function, you need the following:

- A PHP-writable directory for writing the thumbnails
- The GD library

The `mkthumb()` function takes two parameters: the name of the input image file and the desired name of the thumbnail. You need to independently verify that the image file exists and has a .jpg, .gif, or .png extension. With all that out of the way, start by defining default values for the maximum thumbnail width and height (for this function, these dimensions must be equal):

```
<?php
function mkthumb($filename, $thumbname) {
    /* Generate an image thumbnail. */
    $thumb_width = 125;
    $thumb_height = $thumb_width;
```

Now load the image into memory based on the file extension. The GD `imagecreatefrom`*`format()`* functions return an image resource handle. One thing you'll notice right away is that there is no error checking here, primarily because `mkthumb()` assumes that you've taken care of the prerequisites. If you need error checking, check `$src_image` afterward to see if it exists.

```
    if (preg_match('/\.gif$/i', $filename)) {
        $src_image = imagecreatefromgif($filename);
    } else if (preg_match('/\.png$/i', $filename)) {
        $src_image = imagecreatefrompng($filename);
    } else {
        /* Assume JPEG by default. */
        $src_image = imagecreatefromjpeg($filename);
    }
```

Now extract the pixel width and height from the image with the `imagesx()` and `imagesy()` functions:

```
$width = imagesx($src_image);
$height = imagesy($src_image);
```

You resize the image only when the original image is larger than the maximum thumbnail dimensions, so check to see if this is the case:

```
if (($height > $thumb_height) || ($width > $thumb_width)) {
```

When resizing an image, you need to know the final image dimensions in pixels. You can't use the maximum thumbnail height and width, because the original image is probably not square. If you try to shoehorn a non-square rectangle into a square, you will end up with a distorted image. To solve this problem, find the longest side of the original image, and then create a scaling ratio based on the ratio between length of the longest side and the length of the thumbnail side:

```
/* Create a thumbnail. */
if ($width > $height) {
    $ratio = $thumb_width / $width;
} else {
    $ratio = $thumb_height / $height;
}
```

Warning: The reason the maximum thumbnail dimensions are square in this function is that the preceding logic errs for general rectangles. If you try to put a 200 x 400 image into a 100 x 400 thumbnail profile, *`$ratio`* is 1, and you end up with a 200 x 400 thumbnail (see the following code). That size is larger than the desired maximum thumbnail size. Fixing this problem is not difficult, but it is left as an exercise to the reader.

Use the ratio to find the exact thumbnail dimensions in pixels, and create a new image resource handle for the resized image:

```
$new_width = round($width * $ratio);
$new_height = round($height * $ratio);
$dest_image = ImageCreateTrueColor($new_width, $new_height);
```

Now you can perform the resizing with `imagecopyresampled()`:

```
imagecopyresampled($dest_image, $src_image, 0, 0, 0, 0,
                   $new_width, $new_height, $width, $height);
```

You can then free the memory for the original image:

```
        imagedestroy($src_image);
```

At this point, `$dest_image` is an image resource containing the resized image, ready for output. But what if we didn't need to resize? The code falls through to the following `else` clause, which says that it's fine to assign `$dest_image` to the original image handle:

```
        } else {
            /* Image is already small enough; just output. */
            $dest_image = $src_image;
        }
```

Because we now know that `$dest_image` is ready for output, we can write it to a file and clean up:

```
    imagejpeg($dest_image, $thumbname);
    imagedestroy($dest_image);
}

?>
```

Here's a very simple example script that uses the `mkthumb()` function:

```
<?php
include("mkthumb.inc");
$upfile = $_FILES["upfile"]["tmp_name"];
$fn = $_FILES["upfile"]["name"];
$thumb_filename = "thumbs/$fn";

if ($upfile) {
    mkthumb($upfile, $thumb_filename);
    print "<img src=\"$thumb_filename\" />";
} else { ?>
    <form action="thumb.php" enctype="multipart/form-data" method="post">
    Upload an image:<br />
    <input name="upfile" type="file" /><br />
    <input name="Submit" type="submit" value="Upload Image" />
    <?
}
?>
```

Obviously, this script makes a lot of assumptions, and you'd never want to use it for general users.

There are several optimizations that you can do for `mkthumb()` in order to suit your particular needs. For example, JPEG is a lossy format. If you make a thumbnail of a cartoon-style GIF or PNG, the resulting thumbnail will be of poor quality. To fix this problem, you may elect to write in the PNG format with the `imagepng()` function instead of `imagejpeg()`. You could also try to

get tricky and resize into the image's original format. However, this may backfire on you, because GIF images have a limited color map, and when you resize an image, the color map tends to change.

Overall, though, beware of trying to be too tricky with PHP image manipulation. PHP is typically used on demand, so you have little time for fancy operations. In addition to the user getting impatient, the webserver won't let a script run for very long.

Excerpt from: Wicked Cool PHP