

Image Manipulation With jQuery and PHP GD

APRIL 5, 2011

One of the numerous advantages brought about by the explosion of jQuery and other JavaScript libraries is the ease with which you can create interactive tools for your site. When combined with server-side technologies such as PHP, this puts a serious amount of power at your finger tips.

In this article, I'll be looking at how to combine JavaScript/jQuery with PHP and, particularly, PHP's GD library to create an image manipulation tool to upload an image, then crop it and finally save the revised version to the server. Sure, there are plugins out there that you can use to do this; but this article aims to show you what's behind the process. You can download the source files for reference.

We've all seen this sort of Web application before — Facebook, Flickr, t-shirt-printing sites. The advantages are obvious; by including a functionality like this, you alleviate the need to edit pictures manually from your visitors, which has obvious drawbacks. They may not have access to or have the necessary skills to use Photoshop, and in any case why would you want to make the experience of your visitors more difficult?

Before You Start

For this article, you would ideally have had at least some experience working with PHP. Not necessarily GD — I'll run you through that part, and GD is very friendly anyway. You should also be at least intermediate level in JavaScript, though if you're a fast learning beginner, you should be fine as well.

A quick word about the technologies you'll need to work through this article. You'll need a PHP test server running the GD library, either on your hosting or, if working locally, through something like XAMPP. GD has come bundled with PHP

as standard for some time, but you can confirm this by running the `phpinfo()` function and verifying that it's available on your server. Client-side-wise you'll need a text editor, some pictures and a copy of jQuery.

Setting Up The Files

And off we go, then. Set up a working folder and create four files in it: *index.php*, *js.js*, *image_manipulation.php* and *css.css*. *index.php* is the actual webpage, *js.js* and *css.css* should be obvious, while *image_manipulation.php* will store the code that handles the uploaded image and then, later, saves the manipulated version.

In *index.php*, first let's add a line of PHP to start a PHP session and call in our *image_manipulation.php* file:

After that, add in the DOCTYPE and skeleton-structure of the page (header, body areas etc) and call in jQuery and the CSS sheet via script and link tags respectively.

Add a directory to your folder, called *imgs*, which will receive the uploaded files. If you're working on a remote server, ensure you set the permissions on the directory such that the script will be able to save image files in it.

First, let's set up and apply some basic styling to the upload facility.

The Upload Functionality

Now to some basic HTML. Let's add a heading and a simple form to our page that will allow the user to upload an image and assign that image a name:

```
<h1>Image uploader and manipulator</h1>
```

Image on your PC to
upload

Please note that we specify
enctype='multipart/form-data'

which is necessary whenever your form contains file upload fields.

As you can see, the form is pretty basic. It contains 3 fields: an upload field for the image itself, a text field, so the user can give it a name and a submit button. The submit button has a name so it can act as an identifier for our PHP handler script which will know that the form was submitted.

Let's add a smattering of CSS to our stylesheet:

```
/* -----
|  UPLOAD FORM
----- */

#imgForm { border: solid 4px #ddd; background: #eee; padding: 10px; margin: 10px 0; }
#imgForm label { float: left; width: 200px; font-weight: bold; }
#imgForm input { float: left; }
#imgForm input[type="submit"] { clear: both; }
#img_upload { width: 400px; }
#img_name { width: 200px; }
```

Now we have the basic page set up and styled. Next we need to nip into *image_manipulation.php* and prepare it to receive the submitted form. Which leads nicely on to validation...

Validating The Form

Open up *image_manipulation.php*. Since we made a point above of including it into our HTML page, we can rest assured that when it's called into action, it will be present in the environment.

Let's set up a condition, so the PHP knows what task it is being asked to do. Remember we named our submit button *upload_form_submitted*? PHP can now check its existence, since the script knows that it should start handling the form.

This is important because, as I said above, the PHP script has two jobs to do: to handle the uploaded form and to save the manipulated image later on. It therefore needs a technique such as this to know which role it should be doing at

any given time.

```
/* -----
| UPLOAD FORM - validate form and handle submission
----- */

if (isset($_POST['upload_form_submitted'])) {
    //code to validate and handle upload form submission here
}
```

So if the form was submitted, the condition resolves to `true` and whatever code we put inside, it will execute. That code will be validation code. Knowing that the form was submitted, there are now five possible obstacles to successfully saving the file: 1) the upload field was left blank; 2) the file name field was left blank; 3) both these fields were filled in, but the file being uploaded isn't a valid image file; 4) an image with the desired name already exists; 5) everything is fine, but for some reason, the server fails to save the image, perhaps due to file permission issues. Let's look at the code behind picking up each of these scenarios, should any occur, then we'll put it all together to build our validation script.

Combined into a single validation script, the whole code looks as follows.

```
/* -----
| UPLOAD FORM - validate form and handle submission
----- */

if (isset($_POST['upload_form_submitted'])) {

    //error scenario 1
    if (!isset($_FILES['img_upload']) || empty($_FILES['img_upload'])) {
        $error = "Error: You didn't upload a file";
    }

    //error scenario 2
} else if (!isset($_POST['img_name']) || empty($_POST['img_name'])) {
```

```

        $error = "Error: You didn't specify a file name";
    } else {

        $allowedExtensions = array('jpg', 'jpeg', 'gif', 'png',
        preg_match('/\.(implode($allowedExtensions, '|')).')
        $newPath = 'imgs/'.$_POST['img_name'].'.'.$fileExt[0]

        //error scenario 3
        if (file_exists($newPath)) {
            $error = "Error: A file with that name already exists";

        //error scenario 4
        } else if (!in_array(substr($fileExt[0], 1), $allowedExtensions)) {
            $error = 'Error: Invalid file format - please use a valid format';

        //error scenario 5
        } else if (!copy($_FILES['img_upload']['tmp_name'], $newPath)) {
            $error = 'Error: Could not save file to server';

        //...all OK!
        } else {
            $_SESSION['newPath'] = $newPath;
            $_SESSION['fileExt'] = $fileExt;
        }
    }
}

```

There are a couple of things to note here.

\$error & \$_SESSION['newPath']

Firstly, note that I'm using a variable, \$error, to log whether we hit any of the hurdles. If no error occurs and the image is saved, we set a session variable,

`$_SESSION['new_path']`, to store the path to the saved image. This will be helpful in the next step where we need to display the image and, therefore, need to know its SRC.

I'm using a session variable rather than a simple variable, so when the time comes for our PHP script to crop the image, we don't have to pass it a variable informing the script which image to use — the script will already know the context, because it will remember this session variable. Whilst this article doesn't concern itself deeply with security, this is a simple precaution. Doing this means that the user can affect only the image he uploaded, rather than, potentially, someone else's previously-saved image — the user is locked into manipulating only the image referenced in `$error` and has no ability to enforce the PHP script to affect another image.

The `$_FILES` superglobal

Note that even though the form was sent via POST, we access the file upload not via the `$_POST` superglobal (i.e. variables in PHP which are available in all scopes throughout a script), but via the special `$_FILES` superglobal. PHP automatically assigns file fields to that, provided the form was sent with the required `enctype='multipart/form-data'` attribute. Unlike the `$_POST` and `$_GET` superglobals, the `$_FILES` superglobal goes a little “deeper” and is actually a multi-dimensional array. Through this, you can access not only the file itself but also a variety of meta data related to it. You'll see how we can use this information shortly. We use this meta data in the third stage of validation above, namely checking that the file was a valid image file. Let's look at this code in a little more detail.

Confirming the upload is an image


It's sensible that we don't allow the user to proceed if the uploaded file is not an image. So we need to look out for this. First, we create an array of allowed file extensions:

```
$allowedExtensions = array('jpg', 'jpeg', 'gif', 'png');
```

We then check whether or not the extension of the uploaded file is in that array. To do this, we of course need to extract the file extension. Surprisingly, the superglobal doesn't provide this directly, so instead we'll extract it with a regular expression.

Regular expressions are typically considered one of the hardest parts of programming to master. This is definitely true, yet they are often extremely valuable. If you want to read up more on regular expressions, take a look at Smashing Magazine's articles [Crucial Concepts Behind Advanced Regular Expressions](#) or the excellent [Regular-Expressions.info](#). The concept is essentially matching patterns within strings. We know that our extension is the final part of the final name, preceded by a dot, so that forms the basis of our pattern:

```
preg_match('/\.(implode($allowedExtensions, '|')).'$/', $_FILES['im
```



`preg_match()` is the preferred function in PHP to match via REGEXP. It takes three arguments: the pattern, the string to look in, and an array to save matches to. So if a match is found — and of course it should be — our file extension will live in `$fileExt[0]`, i.e. the first and only key of the array of matches.

Patterns are expressed as strings, and inside forward slashes (usually, but not always), so please ignore these parts. Our actual pattern starts with the dot. It has a backslash before it as it needs escaping, because otherwise it would be read as a special character (unescaped dots denote wildcard characters in the regular expression syntax). This is no different to having to escape quotes when using them inside strings, e.g.

```
"...and then he said \"hello, there\"";
```

The next part says: match any ONE of our allowed extensions. Since these live in our array, we convert them into a string via the `implode()` function, separated

by the pipe character. Finally, the dollar character forces the expression to match the end of the string — required in our case, since a file extension is always at the end of a filename. So by the time the PHP engine has evaluated this pattern, it looks as though we had specified this (which is much more readable):

```
'/\.(jpg|jpeg|gif|png)$/'
```

Saving the file

All uploaded files are assigned a temporary home by the server until such time as the session expires or they are moved. So saving the file means moving the file from its temporary location to a permanent home. This is done via the `copy()` function, which needs to know two rather obvious things: what's the path to the temporary file, and what's the path to where we want to put it.

The answer to the first question is read from the `tmp_name` part of the `$_FILES` superglobal. The answer to the second is the full path, including new filename, to where you want it to live. So it is formed of the name of the directory we set up to store images (*/imgs*), plus the new file name (i.e. the value entered into the `img_name` field) and the extension. Let's assign it to its own variable, `$newPath` and then save the file:

```
$newPath = 'imgs/'. $_POST['img_name'] . '.' . $fileExt;  
...  
copy($_FILES['img_upload']['tmp_name'], $newPath);
```

Reporting Back and Moving On

What happens next depends entirely on whether an error occurred, and we can find it out by looking up whether `$error` is set. If it is, we need to communicate this error back to the user. If it's not set, it's time to move on and show the image and let the user manipulate it. Add the following above your form:


```
".$error."
```

```
"; ?>
```

If there's an error, we'd want to show the form again. But the form is currently set to show regardless of the situation. This needs to change, so that it shows only if no image has been uploaded yet, i.e. if the form hasn't been submitted yet, or if it has but there was an error. We can check whether an uploaded image has been saved by interrogating the `$_SESSION['newPath']` variable. Wrap your form HTML in the following two lines of code:

```
"; ?>
```

Now the form appears only if an uploaded image isn't registered — i.e. `$_SESSION['newPath']` isn't set — or if `new=true` is found in the URL. (This latter part provides us with a means of letting the user start over with a new image upload should they wish so; we'll add a link for this in a moment). Otherwise, the uploaded image displays (we know where it lives because we saved its path in `$_SESSION['newPath']`).

This is a good time to take stock of where we are, so try it out. Upload an image, and verify that that it displays. Assuming it does, it's time for our JavaScript to provide some interactivity for image manipulation.

Adding Interactivity

First, let's extend the line we just added so that we a) give the image an ID to reference it later on; b) call the JavaScript itself (along with jQuery); and c) we provide a “start again” link, so the user can start over with a new upload (if necessary). Here is the code snippet:

 " />

start over with new image

```
<script src="http://www.google.com/jsapi"></script>
<script>google.load("jquery", "1.5");</script>
<script src="js.js"></script>
```

Note that I defined an ID for the image, not a class, because it's a unique element, and not one of the many (this sounds obvious, but many people fail to observe this distinction when assigning IDs and classes). Note also, in the image's SRC, I'm appending a random string. This is done to force the browser not to cache the image once we've cropped it (since the SRC doesn't change).

Open *js.js* and let's add the obligatory document ready handler (DRH), required any time you're using freestanding jQuery (i.e. not inside a custom function) to reference or manipulate the DOM. Put the following JavaScript inside this DRH:

```
$(function() {
    // all our JS code will go here
});
```

We're providing the functionality to a user to crop the image, and it of course means allowing him to drag a box area on the image, denoting the part he wishes to keep. Therefore, the first step is to listen for a `mousedown` event on the image, the first of three events involved in a drag action (mouse down, mouse move and then, when the box is drawn, mouse up).

```
var dragInProgress = false;

$("#uploaded_image").mousedown(function(evt) {
    dragInProgress = true;
});
```

And in similar fashion, let's listen to the final mouseup event.

```
$(window).mouseup(function() {  
    dragInProgress = false;  
});
```

Note that our `mouseup` event runs on `window`, not the image itself, since it's possible that the user could release the mouse button anywhere on the page, not necessarily on the image.

Note also that the `mousedown` event handler is prepped to receive the event object. This object holds data about the event, and jQuery always passes it to your event handler, whether or not it's set up to receive it. That object will be crucial later on in ascertaining where the mouse was when the event fired. The `mouseup` event doesn't need this, because all we care about is that the drag action is over and it doesn't really matter where the mouse is.

We're tracking whether or not the mouse button is currently depressed in a variable, `.dragInProgress`. Why? Because, in a drag action, the middle event of the three (see above) only applies if the first happened. That is, in a drag action, you move the mouse *whilst* the mouse is down. If it's not, our `mousemove` event handler should exit. And here it is:

```
$("#uploaded_image").mousemove(function(evt) {  
    if (!dragInProgress) return;  
});
```

So now our three event handlers are set up. As you can see, the `mousemove` event handler exits if it discovers that the mouse button is not currently down, as we decided above it should be.

Now let's extend these event handlers.

This is a good time to explain how our JavaScript will be simulating the drag

action being done by the user. **The trick is to create a DIV on mousedown, and position it at the mouse cursor.** Then, as the mouse moves, i.e. the user is drawing his box, that element should resize consistently to mimic that.

Let's add, position and style our DIV. Before we add it, though, let's remove any previous such DIV, i.e. from a previous drag attempt. This ensures there's only ever one drag box, not several. Also, we want to log the mouse coordinates at the time of mouse down, as we'll need to reference these later when it comes to drawing and resizing our DIV. Extend the mousedown event handler to become:

```
$("#uploaded_image").mousedown(function(evt) {
    dragInProgress = true;
    $("#drag_box").remove();
    $(""
    ").appendTo("body").attr("id", "drag_box").css({left: evt.clientX, top:
    mouseDown_left = evt.clientX;
    mouseDown_top = evt.clientY;
    });
```

Notice that we don't prefix the three variables there with the 'var'.

Notice that we obtain the coordinates of where the event took place —

Let's style the DIV by adding the following CSS to your stylesheet.

```
#drag_box { position: absolute; border: solid 1px #333; background: #fff; opa
```



Now, if you upload an image and then click it, the DIV will be inserted.

So far, so good. Our drag box functionality is almost complete. Now we

The key to this part is working out what properties should be updated

Sounds pretty obvious. However, it's not as simple as it sounds. Imag

To get around this, we need to compare the mousedown coordinates with

▲ left: the lower of the two clientX coordinates

- left: the lower of the two clientX coordinates
- width: the difference between the two clientX coordinates
- top: the lower of the two clientY coordinates
- height: the difference between the two clientY coordinates

So let's extend the mousemove event handler to become:

```
$("#uploaded_image").mousemove(function(evt) {  
    if (!dragInProgress) return;  
    var newLeft = mouseDown_left
```

Notice that, to establish the new width and height, we didn't have to do any

```
result = 50 - 20; //30  
result = Math.abs(20 - 50); //30 (-30 made positive)
```

Try your application out now. You should have full drag box functionality.

Saving the Cropped Image

And so to the last part, saving the modified image. The plan here is simple:

Grabbing the drag box data

It makes sense to grab the drag box's coordinates and dimensions in our mouse

```
var db = $("#drag_box");  
var db_data = {left: db.offset().left, top: db.offset().top, width: db.width(), height:
```



However there's a problem, and it has to do with the drag box's coordinates.

```
var db = $("#drag_box");  
var img_pos = $('#uploaded_image').offset();  
var db_data = {  
    left: db.offset().left - img_pos.left,  
    top: db.offset().top - img_pos.top,  
    width: db.width(),  
    height: db.height()  
};
```

What's happening there? We're first referencing the drag box in a local short

Note that we get the left and top coordinates via jQuery's offset() method. T

For more information on the distinction between all these methods, see my pre
readability.com/articles/awdsfyqi?legac...

Let's now throw out a confirm dialogue box to check that the user wishes to p

```
if (confirm("Crop the image using this drag box?")) {
    location.href = "index.php?crop_attempt=true&crop_l="+db_data.left+"&crop_t="+
db_data.top+"&crop_w="+db_data.width+"&crop_h="+db_data.height;
} else {
    db.remove();
}
```

So if the user clicks 'OK' on the dialogue box that pops up, we redirect to t

PHP: saving the modified file

Remember we said that our *image_manipulation.php* had two tasks – one to first

```
/* -----
| CROP saved image
----- */
```

```
if (isset($_GET["crop_attempt"])) {
    //cropping code here
}
```

So just like before, we condition-off the code area and make sure a flag is p



- destination, the destination image handle
- source, the source image handle
- destination X, the left position to paste TO on the destination im
- destination Y, the top position " " " "
- source X, the left position to grab FROM on the source image handl
- source Y, the top position " " " "
- source W, the width (counting from source X) of the portion to be
- source H, the height (counting from source Y) " " " "

Fortunately, we already have the data necessary to pass to the

Let's create our first handle. As I said, we'll import the uploaded i

```
switch($_SESSION["fileExt"]) {
    case "jpg": case "jpeg":
```

```
    $source_img = imagecreatefromjpeg($_SESSION["basePath"]);
```

```

        var source_img = imagecreatefromjpeg($_SESSION["newPath"]);
        break;
    case "gif":
        var source_img = imagecreatefromgif($_SESSION["newPath"]);
        break;
    case "png":
        var source_img = imagecreatefrompng($_SESSION["newPath"]);
        break;
}

```

As you can see, the file type of the image determines which function

```

switch($_SESSION["fileExt"]) {
    case "jpg": case "jpeg":
        $source_img = imagecreatefromjpeg($_SESSION["newPath"]);
        $dest_img = imagecreatetruecolor($_GET["crop_w"], $_GET["crop_h"]);
        break;
    case "gif":
        $source_img = imagecreatefromgif($_SESSION["newPath"]);
        $dest_img = imagecreate($_GET["crop_w"], $_GET["crop_h"]);
        break;
    case "png":
        $source_img = imagecreatefrompng($_SESSION["newPath"]);
        $dest_img = imagecreate($_GET["crop_w"], $_GET["crop_h"]);
        break;
}

```



You'll notice that the difference between opening a blank image and opening an image from a file is that the image is created with a transparent background.

So now we have our two canvases, it's time to do the copying. The following code will copy the image from the source to the destination.

```

imagecopy(
    $dest_img,
    $source_img,
    0,
    0,
    $_GET["crop_l"],
    $_GET["crop_t"],
    $_GET["crop_w"],
    $_GET["crop_h"]
);

```

```
        $_GET["crop_h"]  
    );
```

The final part is to save the cropped image. For this tutorial, we'll

Saving the image is easy. We just call a particular function based on

```
switch($_SESSION["fileExt"]) {  
    case "jpg": case "jpeg":  
        imagejpeg($dest_img, $_SESSION["newPath"]); break;  
    case "gif":  
        imagegif($dest_img, $_SESSION["newPath"]); break;  
    case "png":  
        imagepng($dest_img, $_SESSION["newPath"]); break;  
}
```

Lastly, we want to redirect to the index page. You might wonder why we

```
header("Location: index.php"); //bye bye arguments
```

Final Touches

So that's it. We now have a fully-working facility to first upload the

There's plenty of ways you could extend this simple application. Expl

With this in mind, you might make the saved file's path more complex,

The possibilities are endless, but hopefully this tutorial has given

(vf)

Tweet

