Uploading Files

Ah yes, the good old problem of file uploads. The basic idea of file uploads is actually quite simple. It basically works like this:

1. A `<form>` tag is marked with `enctype=multipart/form-data` and an `<input type=file>` is placed in that form.
2. The application accesses the file from the **`files`** dictionary on the request object.
3. use the **`save()`** method of the file to save the file permanently somewhere on the filesystem.

A Gentle Introduction

Let's start with a very basic application that uploads a file to a specific upload folder and displays a file to the user. Let's look at the bootstrapping code for our application:

```
import os
from flask import Flask, flash, request, redirect, url_for
from werkzeug.utils import secure_filename

UPLOAD_FOLDER = '/path/to/the/uploads'
ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'}

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
```

So first we need a couple of imports. Most should be straightforward, the **`werkzeug.secure_filename()`** is explained a little bit later. The `UPLOAD_FOLDER` is where we will store the uploaded files and the `ALLOWED_EXTENSIONS` is the set of allowed file extensions.

Why do we limit the extensions that are allowed? You probably don't want your users to be able to upload everything there if the server is directly sending out the data to the client. That way you can make sure that users are not able to upload HTML files that would cause XSS problems (see Cross-Site Scripting (XSS)). Also make sure to disallow `.php` files if the server executes them, but who has PHP installed on their server, right? :)

Next the functions that check if an extension is valid and that uploads the file and redirects the user to the URL for the uploaded file:

```
def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

```
@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # check if the post request has the file part
        if 'file' not in request.files:
            flash('No file part')
            return redirect(request.url)
        file = request.files['file']
        # if user does not select file, browser also
        # submit an empty part without filename
        if file.filename == '':
            flash('No selected file')
            return redirect(request.url)
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filenam
            return redirect(url_for('uploaded_file',
                                    filename=filename))
    return '''
    <!doctype html>
    <title>Upload new File</title>
    <h1>Upload new File</h1>
    <form method=post enctype=multipart/form-data>
      <input type=file name=file>
      <input type=submit value=Upload>
    </form>
    '''
```

So what does that **`secure_filename()`** function actually do? Now the problem is that there is that principle called “never trust user input”. This is also true for the filename of an uploaded file. All submitted form data can be forged, and filenames can be dangerous. For the moment just remember: always use that function to secure a filename before storing it directly on the filesystem.

Information for the Pros:

So you’re interested in what that **`secure_filename()`** function does and what the problem is if you’re not using it? So just imagine someone would send the following information as *filename* to your application:

```
filename = "../../../../home/username/.bashrc"
```

Assuming the number of `../` is correct and you would join this with the `UPLOAD_FOLDER` the user might have the ability to modify a file on the server’s filesystem he or she should

not modify. This does require some knowledge about how the application looks like, but trust me, hackers are patient :)

Now let's look how that function works:

```
>>> secure_filename('../../../../home/username/.bashrc')
'home_username_.bashrc'
```

Now one last thing is missing: the serving of the uploaded files. In the **upload_file()** we redirect the user to `url_for('uploaded_file', filename=filename)`, that is, `/uploads/filename`. So we write the **uploaded_file()** function to return the file of that name. As of Flask 0.5 we can use a function that does that for us:

```
from flask import send_from_directory

@app.route('/uploads/<filename>')
def uploaded_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                               filename)
```

Alternatively you can register *uploaded_file* as *build_only* rule and use the **SharedDataMiddleware**. This also works with older versions of Flask:

```
from werkzeug.middleware.shared_data import SharedDataMiddleware
app.add_url_rule('/uploads/<filename>', 'uploaded_file',
                 build_only=True)
app.wsgi_app = SharedDataMiddleware(app.wsgi_app, {
    '/uploads':  app.config['UPLOAD_FOLDER']
})
```

If you now run the application everything should work as expected.

Improving Uploads

▶ *Changelog*

So how exactly does Flask handle uploads? Well it will store them in the webserver's memory if the files are reasonable small otherwise in a temporary location (as returned by **tempfile.gettempdir()**). But how do you specify the maximum file size after which an upload is aborted? By default Flask will happily accept file uploads to an unlimited amount of memory, but you can limit that by setting the `MAX_CONTENT_LENGTH` config key:

```
from flask import Flask, Request

app = Flask(__name__)
app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024
```

The code above will limit the maximum allowed payload to 16 megabytes. If a larger file is transmitted, Flask will raise a **`RequestEntityTooLarge`** exception.

Connection Reset Issue:

When using the local development server, you may get a connection reset error instead of a 413 response. You will get the correct status response when running the app with a production WSGI server.

This feature was added in Flask 0.6 but can be achieved in older versions as well by subclassing the request object. For more information on that consult the Werkzeug documentation on file handling.

Upload Progress Bars

A while ago many developers had the idea to read the incoming file in small chunks and store the upload progress in the database to be able to poll the progress with JavaScript from the client. Long story short: the client asks the server every 5 seconds how much it has transmitted already. Do you realize the irony? The client is asking for something it should already know.

An Easier Solution

Now there are better solutions that work faster and are more reliable. There are JavaScript libraries like jQuery that have form plugins to ease the construction of progress bar.

Because the common pattern for file uploads exists almost unchanged in all applications dealing with uploads, there is also a Flask extension called Flask-Uploads that implements a full fledged upload mechanism with white and blacklisting of extensions and more.