File Uploads

When Django handles a file upload, the file data ends up placed in `request.FILES` (for more on the `request` object see the documentation for request and response objects). This document explains how files are stored on disk and in memory, and how to customize the default behavior.

Warning: There are security risks if you are accepting uploaded content from untrusted users! See the security guide's topic on User-uploaded content for mitigation details.

Basic file uploads

Consider a form containing a `FileField`:

`forms.py`

```
from django import forms


class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file = forms.FileField()
```

A view handling this form will receive the file data in `request.FILES`, which is a dictionary containing a key for each `FileField` (or `ImageField`, or other `FileField` subclass) in the form. So the data from the above form would be accessible as `request.FILES['file']`.

Note that `request.FILES` will only contain data if the request method was `POST`, at least one file field was actually posted, and the `<form>` that posted the request has the attribute `enctype="multipart/form-data"`. Otherwise, `request.FILES` will be empty.

Most of the time, you'll pass the file data from `request` into the form as described in Binding uploaded files to a form. This would look something like:

`views.py`

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm

# Imaginary function to handle an uploaded file.
from somewhere import handle_uploaded_file


def upload_file(request):
    if request.method == "POST":
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
```

```
            handle_uploaded_file(request.FILES["file"])
            return HttpResponseRedirect("/success/url/")
    else:
        form = UploadFileForm()
    return render(request, "upload.html", {"form": form})
```

Notice that we have to pass `request.FILES` into the form's constructor; this is how file data gets bound into a form.

Here's a common way you might handle an uploaded file:

```
def handle_uploaded_file(f):
    with open("some/file/name.txt", "wb+") as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

Looping over `UploadedFile.chunks()` instead of using `read()` ensures that large files don't overwhelm your system's memory.

There are a few other methods and attributes available on `UploadedFile` objects; see `UploadedFile` for a complete reference.

Handling uploaded files with a model

If you're saving a file on a `Model` with a `FileField`, using a `ModelForm` makes this process much easier. The file object will be saved to the location specified by the `upload_to` argument of the corresponding `FileField` when calling `form.save()`:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import ModelFormWithFileField


def upload_file(request):
    if request.method == "POST":
        form = ModelFormWithFileField(request.POST, request.FILES)
        if form.is_valid():
            # file is saved
            form.save()
            return HttpResponseRedirect("/success/url/")
    else:
        form = ModelFormWithFileField()
    return render(request, "upload.html", {"form": form})
```

If you are constructing an object manually, you can assign the file object from `request.FILES` to the file field in the model:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm
from .models import ModelWithFileField


def upload_file(request):
```

```
    if request.method == "POST":
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            instance = ModelWithFileField(file_field=request.FILES["file"])
            instance.save()
            return HttpResponseRedirect("/success/url/")
    else:
        form = UploadFileForm()
    return render(request, "upload.html", {"form": form})
```

If you are constructing an object manually outside of a request, you can assign a `File` like object to the `FileField`:

```
from django.core.management.base import BaseCommand
from django.core.files.base import ContentFile


class MyCommand(BaseCommand):
    def handle(self, *args, **options):
        content_file = ContentFile(b"Hello world!", name="hello-world.txt")
        instance = ModelWithFileField(file_field=content_file)
        instance.save()
```

Uploading multiple files

If you want to upload multiple files using one form field, create a subclass of the field's widget and set its `allow_multiple_selected` class attribute to `True`.

In order for such files to be all validated by your form (and have the value of the field include them all), you will also have to subclass `FileField`. See below for an example.

> **Multiple file field:** Django is likely to have a proper multiple file field support at some point in the future.

`forms.py`

```
from django import forms


class MultipleFileInput(forms.ClearableFileInput):
    allow_multiple_selected = True


class MultipleFileField(forms.FileField):
    def __init__(self, *args, **kwargs):
        kwargs.setdefault("widget", MultipleFileInput())
        super().__init__(*args, **kwargs)

    def clean(self, data, initial=None):
        single_file_clean = super().clean
        if isinstance(data, (list, tuple)):
            result = [single_file_clean(d, initial) for d in data]
```

```
        else:
            result = [single_file_clean(data, initial)]
        return result


class FileFieldForm(forms.Form):
    file_field = MultipleFileField()
```

Then override the `form_valid()` method of your `FormView` subclass to handle multiple file uploads:

views.py

```
from django.views.generic.edit import FormView
from .forms import FileFieldForm


class FileFieldFormView(FormView):
    form_class = FileFieldForm
    template_name = "upload.html"  # Replace with your template.
    success_url = "..."  # Replace with your URL or reverse().

    def form_valid(self, form):
        files = form.cleaned_data["file_field"]
        for f in files:
            ...  # Do something with each file.
        return super().form_valid(form)
```

Warning: This will allow you to handle multiple files at the form level only. Be aware that you cannot use it to put multiple files on a single model instance (in a single field), for example, even if the custom widget is used with a form field related to a model `FileField`.

Upload Handlers

When a user uploads a file, Django passes off the file data to an *upload handler* – a small class that handles file data as it gets uploaded. Upload handlers are initially defined in the `FILE_UPLOAD_HANDLERS` setting, which defaults to:

```
[
    "django.core.files.uploadhandler.MemoryFileUploadHandler",
    "django.core.files.uploadhandler.TemporaryFileUploadHandler",
]
```

Together `MemoryFileUploadHandler` and `TemporaryFileUploadHandler` provide Django's default file upload behavior of reading small files into memory and large ones onto disk.

You can write custom handlers that customize how Django handles files. You could, for example, use custom handlers to enforce user-level quotas, compress data on the fly, render progress bars, and even send data to another storage location directly without storing it locally. See Writing custom upload handlers for details on how you can customize or completely replace upload behavior.

Where uploaded data is stored

Before you save uploaded files, the data needs to be stored somewhere.

By default, if an uploaded file is smaller than 2.5 megabytes, Django will hold the entire contents of the upload in memory. This means that saving the file involves only a read from memory and a write to disk and thus is very fast.

However, if an uploaded file is too large, Django will write the uploaded file to a temporary file stored in your system's temporary directory. On a Unix-like platform this means you can expect Django to generate a file called something like `/tmp/tmpzfp6I6.upload`. If an upload is large enough, you can watch this file grow in size as Django streams the data onto disk.

These specifics – 2.5 megabytes; `/tmp`; etc. – are "reasonable defaults" which can be customized as described in the next section.

Changing upload handler behavior

There are a few settings which control Django's file upload behavior. See File Upload Settings for details.

Modifying upload handlers on the fly

Sometimes particular views require different upload behavior. In these cases, you can override upload handlers on a per-request basis by modifying `request.upload_handlers`. By default, this list will contain the upload handlers given by `FILE_UPLOAD_HANDLERS`, but you can modify the list as you would any other list.

For instance, suppose you've written a `ProgressBarUploadHandler` that provides feedback on upload progress to some sort of AJAX widget. You'd add this handler to your upload handlers like this:

```
request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
```

You'd probably want to use `list.insert()` in this case (instead of `append()`) because a progress bar handler would need to run *before* any other handlers. Remember, the upload handlers are processed in order.

If you want to replace the upload handlers completely, you can assign a new list:

```
request.upload_handlers = [ProgressBarUploadHandler(request)]
```

Note: You can only modify upload handlers *before* accessing `request.POST` or `request.FILES` – it doesn't make sense to change upload handlers after upload handling has already started. If you try to modify `request.upload_handlers` after reading from `request.POST` or `request.FILES` Django will throw an error.

Thus, you should always modify uploading handlers as early in your view as possible.

Also, `request.POST` is accessed by `CsrfViewMiddleware` which is enabled by default. This means you will need to use `csrf_exempt()` on your view to allow you to change the upload handlers. You will then need to use `csrf_protect()` on

the function that actually processes the request. Note that this means that the handlers may start receiving the file upload before the CSRF checks have been done. Example code:

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect


@csrf_exempt
def upload_file_view(request):
    request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
    return _upload_file_view(request)


@csrf_protect
def _upload_file_view(request):
    # Process request
    ...
```

If you are using a class-based view, you will need to use `csrf_exempt()` on its `dispatch()` method and `csrf_protect()` on the method that actually processes the request. Example code:

```
from django.utils.decorators import method_decorator
from django.views import View
from django.views.decorators.csrf import csrf_exempt, csrf_protect


@method_decorator(csrf_exempt, name="dispatch")
class UploadFileView(View):
    def setup(self, request, *args, **kwargs):
        request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
        super().setup(request, *args, **kwargs)

    @method_decorator(csrf_protect)
    def post(self, request, *args, **kwargs):
        # Process request
        ...
```