# File Uploads

When Django handles a file upload, the file data ends up placed in request.FILES

# Basic file uploads ---

Consider a form containing a FileField:

**Listing 6: 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:

Listing 7: 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 proper multiple file field support at some point in the future.

Listing 8: 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:

Listing 9: 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.

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.

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

...

------------------------------------- END ------------------------------------------------