Managing files

This document describes Django's file access APIs for files such as those uploaded by a user. The lower level APIs are general enough that you could use them for other purposes. If you want to handle "static files" (JS, CSS, etc.), see How to manage static files (e.g. images, JavaScript, CSS).

By default, Django stores files locally, using the `MEDIA_ROOT` and `MEDIA_URL` settings. The examples below assume that you're using these defaults.

However, Django provides ways to write custom file storage systems that allow you to completely customize where and how Django stores files. The second half of this document describes how these storage systems work.

Using files in models

When you use a `FileField` or `ImageField`, Django provides a set of APIs you can use to deal with that file.

Consider the following model, using an `ImageField` to store a photo:

```
from django.db import models


class Car(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField(max_digits=5, decimal_places=2)
    photo = models.ImageField(upload_to="cars")
    specs = models.FileField(upload_to="specs")
```

Any `Car` instance will have a `photo` attribute that you can use to get at the details of the attached photo:

```
>>> car = Car.objects.get(name="57 Chevy")
>>> car.photo
<ImageFieldFile: cars/chevy.jpg>
>>> car.photo.name
'cars/chevy.jpg'
>>> car.photo.path
'/media/cars/chevy.jpg'
>>> car.photo.url
'https://media.example.com/cars/chevy.jpg'
```

This object – `car.photo` in the example – is a `File` object, which means it has all the methods and attributes described below.

Note: The file is saved as part of saving the model in the database, so the actual file name used on disk cannot be relied on until after the model has been saved.

For example, you can change the file name by setting the file's `name` to a path relative to the file storage's location (`MEDIA_ROOT` if you are using the default `FileSystemStorage`):

```
>>> import os
>>> from django.conf import settings
>>> initial_path = car.photo.path
>>> car.photo.name = "cars/chevy_ii.jpg"
>>> new_path = settings.MEDIA_ROOT + car.photo.name
>>> # Move the file on the filesystem
>>> os.rename(initial_path, new_path)
>>> car.save()
>>> car.photo.path
'/media/cars/chevy_ii.jpg'
>>> car.photo.path == new_path
True
```

To save an existing file on disk to a `FileField`:

```
>>> from pathlib import Path
>>> from django.core.files import File
>>> path = Path("/some/external/specs.pdf")
>>> car = Car.objects.get(name="57 Chevy")
>>> with path.open(mode="rb") as f:
...     car.specs = File(f, name=path.name)
...     car.save()
...
```

Note: While `ImageField` non-image data attributes, such as `height`, `width`, and `size` are available on the instance, the underlying image data cannot be used without reopening the image. For example:

```
>>> from PIL import Image
>>> car = Car.objects.get(name="57 Chevy")
>>> car.photo.width
191
>>> car.photo.height
287
>>> image = Image.open(car.photo)
# Raises ValueError: seek of closed file.
>>> car.photo.open()
<ImageFieldFile: cars/chevy.jpg>
>>> image = Image.open(car.photo)
>>> image
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=191x287 at 0x7F99A94E9048>
```

The `File` object

Internally, Django uses a `django.core.files.File` instance any time it needs to represent a file.

Most of the time you'll use a `File` that Django's given you (i.e. a file attached to a model as above, or perhaps an uploaded file).

If you need to construct a `File` yourself, the easiest way is to create one using a Python built-in `file` object:

```
>>> from django.core.files import File

# Create a Python file object using open()
>>> f = open("/path/to/hello.world", "w")
>>> myfile = File(f)
```

Now you can use any of the documented attributes and methods of the `File` class.

Be aware that files created in this way are not automatically closed. The following approach may be used to close files automatically:

```
>>> from django.core.files import File

# Create a Python file object using open() and the with statement
>>> with open("/path/to/hello.world", "w") as f:
...     myfile = File(f)
...     myfile.write("Hello World")
...
>>> myfile.closed
True
>>> f.closed
True
```

Closing files is especially important when accessing file fields in a loop over a large number of objects. If files are not manually closed after accessing them, the risk of running out of file descriptors may arise. This may lead to the following error:

```
OSError: [Errno 24] Too many open files
```

File storage

Behind the scenes, Django delegates decisions about how and where to store files to a file storage system. This is the object that actually understands things like file systems, opening and reading files, etc.

Django's default file storage is `'django.core.files.storage.FileSystemStorage'`. If you don't explicitly provide a storage system in the `default` key of the `STORAGES` setting, this is the one that will be used.

See below for details of the built-in default file storage system, and see How to write a custom storage class for information on writing your own file storage system.

Storage objects

Though most of the time you'll want to use a `File` object (which delegates to the proper storage for that file), you can use file storage systems directly. You can create an instance of some custom file storage class, or – often more useful – you can use the global default storage system:

```
>>> from django.core.files.base import ContentFile
>>> from django.core.files.storage import default_storage

>>> path = default_storage.save("path/to/file", ContentFile(b"new content"))
>>> path
'path/to/file'

>>> default_storage.size(path)
11
>>> default_storage.open(path).read()
b'new content'

>>> default_storage.delete(path)
>>> default_storage.exists(path)
False
```

See File storage API for the file storage API.

The built-in filesystem storage class

Django ships with a `django.core.files.storage.FileSystemStorage` class which implements basic local filesystem file storage.

For example, the following code will store uploaded files under `/media/photos` regardless of what your `MEDIA_ROOT` setting is:

```
from django.core.files.storage import FileSystemStorage
from django.db import models

fs = FileSystemStorage(location="/media/photos")


class Car(models.Model):
    ...
    photo = models.ImageField(storage=fs)
```

Custom storage systems work the same way: you can pass them in as the `storage` argument to a `FileField`.

Using a callable

You can use a callable as the `storage` parameter for `FileField` or `ImageField`. This allows you to modify the used storage at runtime, selecting different storages for different environments, for example.

Your callable will be evaluated when your models classes are loaded, and must return an instance of `Storage`.

For example:

```
from django.conf import settings
from django.db import models
from .storages import MyLocalStorage, MyRemoteStorage


def select_storage():
    return MyLocalStorage() if settings.DEBUG else MyRemoteStorage()


class MyModel(models.Model):
    my_file = models.FileField(storage=select_storage)
```

In order to set a storage defined in the `STORAGES` setting you can use `storages`:

```
from django.core.files.storage import storages


def select_storage():
    return storages["mystorage"]


class MyModel(models.Model):
    upload = models.FileField(storage=select_storage)
```