Django Database File Storage Documentation

Release 0.5.5

Victor Oliveira da Silva

May 31, 2020

Contents

Django Database File Storage (PyPI) (GitHub) is a custom file storage system for Django. Use it to save files in your database instead of your file system.

CHAPTER 1

Installing

You can pip-install django-db-file-storage in your environment by typing the following code on your shell:

```
pip install django-db-file-storage
```

1.1 Settings

On your project's settings, add `'db_file_storage'` to your INSTALLED_APPS list.

CHAPTER 2

How to use (for FileFields on models)

When used for models' FileFields, django-db-file-storage uses a specific model to hold each FileField.

2.1 Settings

On your project's settings, set DEFAULT_FILE_STORAGE like this:

```
DEFAULT_FILE_STORAGE = 'db_file_storage.storage.DatabaseFileStorage'
```

2.2 URLs

Add the following URL pattern to your project's main urlpatterns (/urls.py):

```
url(r'^files/', include('db_file_storage.urls')),
```

2.3 Models

For each FileField you want to save, you will need a separated model to hold the file in the database. I will refer to this extra model as the **FileModel**. The FileModel must have exactly these fields:

- either a `TextField()` or a `BinaryField()` - will hold the encoded contents of the file
- a `CharField(max_length=255)` - will hold the file's name
- a `CharField(max_length=50)` - will hold the file's MIME type

For example (in a models.py file, inside an app called 'console'):

```
from django.db import models

class ConsolePicture(models.Model):
    bytes = models.TextField()
    filename = models.CharField(max_length=255)
    mimetype = models.CharField(max_length=50)
```

And the class which will have the FileField:

```
class Console(models.Model):
    name = models.CharField(max_length=100)
    picture = models.ImageField(upload_to='console.ConsolePicture/bytes/filename/
↪mimetype', blank=True, null=True)
```

In this example, the FileField is actually an ImageField. Its `upload_to` argument must be a string in the following format:

1. the FileModel's app's name
2. a dot (.)
3. the FileModel's name
4. a forward slash (/)
5. the name of the FileModel's field which will hold the encoded contents of the files
6. a forward slash
7. the name of the FileModel's field which will hold the name of the files
8. a forward slash
9. the name of the FileModel's field which will hold the MIME type of the files

Let's check it again:

```
#   1   2       3       4  5  6    7   8    9
'console.ConsolePicture/bytes/filename/mimetype'
```

Don't forget to create the necessary tables in your database, if you haven't yet.

If you want stale files to be deleted when editing and deleting instances, override the `save` and `delete` methods of your model, calling `db_file_storage.model_utils.delete_file_if_needed` and `db_file_storage.model_utils.delete_file` inside them, respectively:

```
from db_file_storage.model_utils import delete_file, delete_file_if_needed
from django.db import models

class Console(models.Model):
    name = models.CharField(max_length=100, unique=True)
    picture = models.ImageField(upload_to='console.ConsolePicture/bytes/filename/
↪mimetype', blank=True, null=True)

    def save(self, *args, **kwargs):
        delete_file_if_needed(self, 'picture')
        super(Console, self).save(*args, **kwargs)

    def delete(self, *args, **kwargs):
        super(Console, self).delete(*args, **kwargs)
        delete_file(self, 'picture')
```

Pay extra attention here to when the methods should be called. `delete_file_if_needed` should be called **before** the `save` method of the super class, and `delete_file` should be called **after** the `delete` method of the super class.

2.4 Form widget

At this point, your project already must be saving files in the database when you use Django's ModelForms.

However, due to Django Database File Storage's internal logic, Django's default widget for file inputs won't show the proper filename when downloading uploaded files. The download itself works perfectly, it's just the widget that doesn't show the correct name in its download link.

Django Database File Storage comes with a custom widget to solve this problem: DBClearableFileInput. You just need to use it when defining your form class:

```
from console.models import Console
from db_file_storage.form_widgets import DBClearableFileInput
from django import forms

class ConsoleForm(forms.ModelForm):
    class Meta:
        model = Console
        exclude = []
        widgets = {
            'picture': DBClearableFileInput
        }
```

2.4.1 Admin Form widget

In order to solve the same problem in the Django Admin interface, Django Database File Storage comes with another custom widget: DBAdminClearableFileInput. You just need to use it when defining your form class, and then use such form when defining your ModelAdmin class inside your `admin.py` file:

```
from console.models import Console
from db_file_storage.form_widgets import DBAdminClearableFileInput
from django import forms
from django.contrib import admin

class ConsoleForm(forms.ModelForm):
    class Meta:
        model = Console
        exclude = []
        widgets = {
            'picture': DBAdminClearableFileInput
        }

class ConsoleAdmin(admin.ModelAdmin):
    form = ConsoleForm
```

2.5 Downloading (and viewing) the files

Django Database File Storage comes with views that you can use to download the files or to just view them (for images, for example). They are accessed through the named url patterns **db_file_storage.download_file** and

db_file_storage.get_file.

Both views must be passed a GET parameter named `name`, and the value of this parameter must be the value of the filefield of the instance. The template-snippet example below must make it clearer; `console` is an instance of the `Console` model defined above:

```
<!-- The url used to VIEW the file: -->
<img src="{% url 'db_file_storage.get_file' %}?name={{ console.picture }}" />

<br/>

<!-- The url used to DOWNLOAD the file: -->
<a href='{% url "db_file_storage.download_file" %}?name={{ console.picture }}'>
    <i>Click here to download the picture</i>
</a>
```

2.5.1 Customizing HTTP headers

If you need to set extra HTTP headers, you can create your own URL patterns using Django Database File Storage's views and customize the response headers with the keyword argument `extra_headers`.

First, define the new URL patterns in your app's urls.py:

```
from db_file_storage import views as db_file_storage_views

urlpatterns = [
  url(
       r'^console-picture-view/',
       db_file_storage_views.get_file,
       {
           'add_attachment_headers': False,
           'extra_headers': {'Content-Language': 'en'}
       },
       name='console.view_picture'
   ),
   url(
       r'^console-picture-download/',
       db_file_storage_views.get_file,
       {
           'add_attachment_headers': True,  # Shows a "File Download" box in the␣
↪browser
           'extra_headers': {'Content-Language': 'en'}
       },
       name='console.download_picture'
   ),
]
```

Then, use the new URLs in the templates:

```
<!-- The url used to VIEW the file: -->
 <img src="{% url 'console.view_picture' %}?name={{ console.picture }}" />

 <br/>

 <!-- The url used to DOWNLOAD the file: -->
 <a href='{% url "console.download_picture" %}?name={{ console.picture }}'>
```

(continues on next page)

(continued from previous page)

```
    <i>Click here to download the picture</i>
</a>
```

In the demo project there is a working example with custom HTTP headers in the responses. It's the cover download link in the books list.

CHAPTER 3

How to use (for Form Wizards)

When used this way, django-db-file-storage uses a fixed model to store all the saved files. Just set *db_file_storage.storage.FixedModelDatabaseFileStorage* as the wizard's *file_storage*, passing all the attributes that you would define if you were using a model's FileField:

```
from db_file_storage.storage import FixedModelDatabaseFileStorage
from formtools.wizard.views import SessionWizardView

class ExampleFormWizard(SessionWizardView):
    file_storage = FixedModelDatabaseFileStorage(
        model_class_path='form_wizard_example.FormWizardFile',
        content_field='bytes',
        filename_field='filename',
        mimetype_field='mimetype'
    )
    (...)
```

All the parameters shown above are required for the *FixedModelDatabaseFileStorage* initialization. The model that will hold the files must be defined as well (in *form_wizard_example/models.py*, in this case):

```
class FormWizardFile(models.Model):
    bytes = models.TextField()
    filename = models.CharField(max_length=255)
    mimetype = models.CharField(max_length=50)
```

In the demo project there is a working example with a Form Wizard.