django-storages Documentation

Release 1.3.1

David Larlet, et. al.

October 17, 2016

Contents

django-storages is a collection of custom storage backends for Django.

CHAPTER 1

Amazon S3

1.1 Usage

There is one backend for interacting with S3 based on the boto library. A legacy backend backed on the Amazon S3 Python library was removed in version 1.2.

1.1.1 Settings

To use s3boto set:

```
DEFAULT_FILE_STORAGE = 'storages.backends.s3boto.S3BotoStorage'
```

`AWS_ACCESS_KEY_ID`

Your Amazon Web Services access key, as a string.

`AWS_SECRET_ACCESS_KEY`

Your Amazon Web Services secret access key, as a string.

`AWS_STORAGE_BUCKET_NAME`

Your Amazon Web Services storage bucket name, as a string.

`AWS_AUTO_CREATE_BUCKET` (optional)

If set to `True` the bucket specified in `AWS_STORAGE_BUCKET_NAME` is automatically created.

`AWS_HEADERS` (optional)

If you'd like to set headers sent with each file of the storage:

```
# see http://developer.yahoo.com/performance/rules.html#expires
AWS_HEADERS = {
    'Expires': 'Thu, 15 Apr 2010 20:00:00 GMT',
    'Cache-Control': 'max-age=86400',
}
```

To allow `django-admin.py` collectstatic to automatically put your static files in your bucket set the following in your settings.py:

```
STATICFILES_STORAGE = 'storages.backends.s3boto.S3BotoStorage'
```

1.1.2 Fields

Once you're done, default_storage will be the S3 storage:

```
>>> from django.core.files.storage import default_storage
>>> print default_storage.__class__
<class 'S3Storage.S3Storage'>
```

The above doesn't seem to be true for django 1.3+ instead look at:

```
>>> from django.core.files.storage import default_storage
>>> print default_storage.connection
S3Connection:s3.amazonaws.com
```

This way, if you define a new FileField, it will use the S3 storage:

```
>>> from django.db import models
>>> class Resume(models.Model):
...     pdf = models.FileField(upload_to='pdfs')
...     photos = models.ImageField(upload_to='photos')
...
>>> resume = Resume()
>>> print resume.pdf.storage
<S3Storage.S3Storage object at ...>
```

1.2 Tests

Initialization:

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

1.2.1 Storage

Standard file access options are available, and work as expected:

```
>>> default_storage.exists('storage_test')
False
>>> file = default_storage.open('storage_test', 'w')
>>> file.write('storage contents')
>>> file.close()

>>> default_storage.exists('storage_test')
True
>>> file = default_storage.open('storage_test', 'r')
>>> file.read()
'storage contents'
>>> file.close()

>>> default_storage.delete('storage_test')
>>> default_storage.exists('storage_test')
False
```

1.2.2 Model

An object without a file has limited functionality:

```
>>> obj1 = MyStorage()
>>> obj1.normal
<FieldFile: None>
>>> obj1.normal.size
Traceback (most recent call last):
...
ValueError: The 'normal' attribute has no file associated with it.
```

Saving a file enables full functionality:

```
>>> obj1.normal.save('django_test.txt', ContentFile('content'))
>>> obj1.normal
<FieldFile: tests/django_test.txt>
>>> obj1.normal.size
7
>>> obj1.normal.read()
'content'
```

Files can be read in a little at a time, if necessary:

```
>>> obj1.normal.open()
>>> obj1.normal.read(3)
'con'
>>> obj1.normal.read()
'tent'
>>> '-'.join(obj1.normal.chunks(chunk_size=2))
'co-nt-en-t'
```

Save another file with the same name:

```
>>> obj2 = MyStorage()
>>> obj2.normal.save('django_test.txt', ContentFile('more content'))
>>> obj2.normal
<FieldFile: tests/django_test_.txt>
>>> obj2.normal.size
12
```

Push the objects into the cache to make sure they pickle properly:

```
>>> cache.set('obj1', obj1)
>>> cache.set('obj2', obj2)
>>> cache.get('obj2').normal
<FieldFile: tests/django_test_.txt>
```

Deleting an object deletes the file it uses, if there are no other objects still using that file:

```
>>> obj2.delete()
>>> obj2.normal.save('django_test.txt', ContentFile('more content'))
>>> obj2.normal
<FieldFile: tests/django_test_.txt>
```

Default values allow an object to access a single file:

```
>>> obj3 = MyStorage.objects.create()
>>> obj3.default
<FieldFile: tests/default.txt>
```

```
>>> obj3.default.read()
'default content'
```

But it shouldn't be deleted, even if there are no more objects using it:

```
>>> obj3.delete()
>>> obj3 = MyStorage()
>>> obj3.default.read()
'default content'
```

Verify the fix for #5655, making sure the directory is only determined once:

```
>>> obj4 = MyStorage()
>>> obj4.random.save('random_file', ContentFile('random content'))
>>> obj4.random
<FieldFile: .../random_file>
```

Clean up the temporary files:

```
>>> obj1.normal.delete()
>>> obj2.normal.delete()
>>> obj3.default.delete()
>>> obj4.random.delete()
```

CHAPTER 2

Apache Libcloud

Apache Libcloud is an API wrapper around a range of cloud storage providers. It aims to provide a consistent API for dealing with cloud storage (and, more broadly, the many other services provided by cloud providers, such as device provisioning, load balancer configuration, and DNS configuration).

As of v0.10.1, Libcloud supports the following cloud storage providers:

- Amazon S3
- Google Cloud Storage
- Nimbus.io
- Ninefold Cloud Storage
- Rackspace CloudFiles

Libcloud can also be configured with relatively little effort to support any provider using EMC Atmos storage, or the OpenStack API.

2.1 Settings

2.1.1 LIBCLOUD_PROVIDERS

This setting is required to configure connections to cloud storage providers. Each entry corresponds to a single 'bucket' of storage. You can have multiple buckets for a single service provider (e.g., multiple S3 buckets), and you can define buckets at multiple providers. For example, the following configuration defines 3 providers: two buckets (`bucket-1` and `bucket-2`) on a US-based Amazon S3 store, and a third bucket (`bucket-3`) on Google:

```
LIBCLOUD_PROVIDERS = {
    'amazon_1': {
        'type': 'libcloud.storage.types.Provider.S3_US_STANDARD_HOST',
        'user': '<your username here>',
        'key': '<your key here>',
        'bucket': 'bucket-1',
    },
    'amazon_2': {
        'type': 'libcloud.storage.types.Provider.S3_US_STANDARD_HOST',
        'user': '<your username here>',
        'key': '<your key here>',
        'bucket': 'bucket-2',
    },
    'google': {
```

```
        'type': 'libcloud.storage.types.Provider.GOOGLE_STORAGE',
        'user': '<Your Google APIv1 username>',
        'key': '<Your Google APIv1 Key>',
        'bucket': 'bucket-3',
    },
}
```

The values for the `type`, `user` and `key` arguments will vary depending on your storage provider:

Amazon S3:

type: `libcloud.storage.types.Provider.S3_US_STANDARD_HOST`,

user: Your AWS access key ID

key: Your AWS secret access key

If you want to use a availability zone other than the US default, you can use one of `S3_US_WEST_HOST`, `S3_US_WEST_OREGON_HOST`, `S3_EU_WEST_HOST`, `S3_AP_SOUTHEAST_HOST`, or `S3_AP_NORTHEAST_HOST` instead of `S3_US_STANDARD_HOST`.

Google Cloud Storage:

type: `libcloud.storage.types.Provider.GOOGLE_STORAGE`,

user: Your Google APIv1 username (20 characters)

key: Your Google APIv1 key

Nimbus.io:

type: `libcloud.storage.types.Provider.NIMBUS`,

user: Your Nimbus.io user ID

key: Your Nimbus.io access key

Ninefold Cloud Storage:

type: `libcloud.storage.types.Provider.NINEFOLD`,

user: Your Atmos Access Token

key: Your Atmos Shared Secret

Rackspace Cloudfiles:

type: `libcloud.storage.types.Provider.CLOUDFIULES_US` or `libcloud.storage.types.Provider.CLOUDFIULES_UK`,

user: Your Rackspace user ID

key: Your Rackspace access key

You can specify any bucket name you want; however, the bucket must exist before you can start using it. If you need to create the bucket, you can use the storage API. For example, to create `bucket-1` from our previous example:

```
>>> from storages.backends.apache_libcloud import LibCloudStorage
>>> store = LibCloudStorage('amazon_1')
>>> store.driver.create_container('bucket-1')
```

2.1.2 DEFAULT_LIBCLOUD_PROVIDER

Once you have defined your Libcloud providers, you have the option of setting one provider as the default provider of Libcloud storage. This is done setting `DEFAULT_LIBCLOUD_PROVIDER` to the key in `LIBCLOUD_PROVIDER` that you want to use as the default provider. For example, if you want the `amazon-1` provider to be the default provider, use:

```
DEFAULT_LIBCLOUD_PROVIDER = 'amazon-1'
```

If `DEFAULT_LIBCLOUD_PROVIDER` isn't set, the Libcloud backend will assume that the default storage backend is named `default`. Therefore, you can avoid settings DEFAULT_LIBCLOUD_PROVIDER by simply naming one of your Libcloud providers `default`:

```
LIBCLOUD_PROVIDERS = {
    'default': {
        'type': ...
    },
}
```

2.1.3 DEFAULT_FILE_STORAGE

If you want your Libcloud storage to be the default Django file store, you can set:

```
DEFAULT_FILE_STORAGE = 'storages.backends.apache_libcloud.LibCloudStorage'
```

Your default Libcloud provider will be used as the file store.

2.2 Certifcate authorities

Libcloud uses HTTPS connections, and in order to validate that these HTTPS connections are correctly signed, root CA certificates must be present. On some platforms (most notably, OS X and Windows), the required certificates may not be available by default. To test

```
>>> from storages.backends.apache_libcloud import LibCloudStorage
>>> store = LibCloudStorage('amazon_1')
Traceback (most recent call last):
...
ImproperlyConfigured: Unable to create libcloud driver type libcloud.storage.types.Provider.S3_US_STA
```

If you get this error, you need to install a certificate authority. Download a certificate authority file, and then put the following two lines into your settings.py:

```
import libcloud.security
libcloud.security.CA_CERTS_PATH.append("/path/to/your/cacerts.pem")
```

CHAPTER 3

Azure Storage

A custom storage system for Django using Windows Azure Storage backend.

3.1 Settings

`DEFAULT_FILE_STORAGE`

This setting sets the path to the Azure storage class:

```
DEFAULT_FILE_STORAGE = 'storages.backends.azure_storage.AzureStorage'
```

`AZURE_ACCOUNT_NAME`

This setting is the Windows Azure Storage Account name, which in many cases is also the first part of the url for instance: http://azure_account_name.blob.core.windows.net/ would mean:

```
AZURE_ACCOUNT_NAME = "azure_account_name"
```

`AZURE_ACCOUNT_KEY`

This is the private key that gives your Django app access to your Windows Azure Account.

`AZURE_CONTAINER`

This is where the files uploaded through your Django app will be uploaded. The container must be already created as the storage system will not attempt to create it.

CHAPTER 4

CouchDB

A custom storage system for Django with CouchDB backend.

CHAPTER 5

Database

Class DatabaseStorage can be used with either FileField or ImageField. It can be used to map filenames to database blobs: so you have to use it with a special additional table created manually. The table should contain a pk-column for filenames (better to use the same type that FileField uses: nvarchar(100)), blob field (image type for example) and size field (bigint). You can't just create blob column in the same table, where you defined FileField, since there is no way to find required row in the save() method. Also size field is required to obtain better perfomance (see size() method).

So you can use it with different FileFields and even with different "upload_to" variables used. Thus it implements a kind of root filesystem, where you can define dirs using "upload_to" with FileField and store any files in these dirs.

It uses either settings.DB_FILES_URL or constructor param 'base_url' (see __init__()) to create urls to files. Base url should be mapped to view that provides access to files. To store files in the same table, where FileField is defined you have to define your own field and provide extra argument (e.g. pk) to save().

Raw sql is used for all operations. In constructor or in DB_FILES of settings.py () you should specify a dictionary with db_table, fname_column, blob_column, size_column and 'base_url'. For example I just put to the settings.py the following line:

```
DB_FILES = {
    'db_table': 'FILES',
    'fname_column':  'FILE_NAME',
    'blob_column': 'BLOB',
    'size_column': 'SIZE',
    'base_url': 'http://localhost/dbfiles/'
}
```

And use it with ImageField as following:

```
player_photo = models.ImageField(upload_to="player_photos", storage=DatabaseStorage() )
```

DatabaseStorage class uses your settings.py file to perform custom connection to your database.

The reason to use custom connection: http://code.djangoproject.com/ticket/5135 Connection string looks like:

```
cnxn = pyodbc.connect('DRIVER={SQL Server};SERVER=localhost;DATABASE=testdb;UID=me;PWD=pass')
```

It's based on pyodbc module, so can be used with any database supported by pyodbc. I've tested it with MS Sql Express 2005.

Note: It returns special path, which should be mapped to special view, which returns requested file:

```
def image_view(request, filename):
    import os
    from django.http import HttpResponse
    from django.conf import settings
    from django.utils._os import safe_join
```

```
    from filestorage import DatabaseStorage
    from django.core.exceptions import ObjectDoesNotExist

    storage = DatabaseStorage()

    try:
        image_file = storage.open(filename, 'rb')
        file_content = image_file.read()
    except:
        filename = 'no_image.gif'
        path = safe_join(os.path.abspath(settings.MEDIA_ROOT), filename)
        if not os.path.exists(path):
            raise ObjectDoesNotExist
        no_image = open(path, 'rb')
        file_content = no_image.read()

    response = HttpResponse(file_content, mimetype="image/jpeg")
    response['Content-Disposition'] = 'inline; filename=%s'%filename
    return response
```

Note: If filename exist, blob will be overwritten, to change this remove get_available_name(self, name), so Storage.get_available_name(self, name) will be used to generate new filename.

CHAPTER 6

FTP

Warning: This FTP storage is not prepared to work with large files, because it uses memory for temporary data storage. It also does not close FTP connection automatically (but open it lazy and try to reestablish when disconnected).

This implementation was done preliminary for upload files in admin to remote FTP location and read them back on site by HTTP. It was tested mostly in this configuration, so read/write using FTPStorageFile class may break.

CHAPTER 7

Image

A custom FileSystemStorage made for normalizing extensions. It lets PIL look at the file to determine the format and append an always lower-case extension based on the results.

CHAPTER 8

MogileFS

This storage allows you to use MogileFS, it comes from this blog post.

The MogileFS storage backend is fairly simple: it uses URLs (or, rather, parts of URLs) as keys into the mogile database. When the user requests a file stored by mogile (say, an avatar), the URL gets passed to a view which, using a client to the mogile tracker, retrieves the "correct" path (the path that points to the actual file data). The view will then either return the path(s) to perlbal to reproxy, or, if you're not using perlbal to reproxy (which you should), it serves the data of the file directly from django.

- `MOGILEFS_DOMAIN`: The mogile domain that files should read from/written to, e.g "production"
- `MOGILEFS_TRACKERS`: A list of trackers to connect to, e.g. ["foo.sample.com:7001", "bar.sample.com:7001"]
- `MOGILEFS_MEDIA_URL` (optional): The prefix for URLs that point to mogile files. This is used in a similar way to `MEDIA_URL`, e.g. "/mogilefs/"
- `SERVE_WITH_PERLBAL`: Boolean that, when True, will pass the paths back in the response in the `X-REPROXY-URL` header. If False, django will serve all mogile media files itself (bad idea for production, but useful if you're testing on a setup that doesn't have perlbal running)
- `DEFAULT_FILE_STORAGE`: This is the class that's used for the backend. You'll want to set this to `project.app.storages.MogileFSStorage` (or wherever you've installed the backend)

8.1 Getting files into mogile

The great thing about file backends is that we just need to specify the backend in the model file and everything is taken care for us - all the default save() methods work correctly.

For Fluther, we have two main media types we use mogile for: avatars and thumbnails. Mogile defines "classes" that dictate how each type of file is replicated - so you can make sure you have 3 copies of the original avatar but only 1 of the thumbnail.

In order for classes to behave nicely with the backend framework, we've had to do a little tomfoolery. (This is something that may change in future versions of the filestorage framework).

Here's what the models.py file looks like for the avatars:

```
from django.core.filestorage import storage

# TODO: Find a better way to deal with classes. Maybe a generator?
class AvatarStorage(storage.__class__):
    mogile_class = 'avatar'
```

```
class ThumbnailStorage(storage.__class__):
    mogile_class = 'thumb'

class Avatar(models.Model):
    user = models.ForeignKey(User, null=True, blank=True)
    image = models.ImageField(storage=AvatarStorage())
    thumb = models.ImageField(storage=ThumbnailStorage())
```

Each of the custom storage classes defines a class attribute which gets passed to the mogile backend behind the scenes. If you don't want to worry about mogile classes, don't need to define a custom storage engine or specify it in the field - the default should work just fine.

8.2 Serving files from mogile

Now, all we need to do is plug in the view that serves up mogile data.

Here's what we use:

```
urlpatterns += patterns('',
    (r'^%s(?P<key>.*)' % settings.MOGILEFS_MEDIA_URL[1:],
        'MogileFSStorage.serve_mogilefs_file')
)
```

Any url beginning with the value of `MOGILEFS_MEDIA_URL` will get passed to our view. Since `MOGILEFS_MEDIA_URL` requires a leading slash (like `MEDIA_URL`), we strip that off and pass the rest of the url over to the view.

That's it! Happy mogiling!

CHAPTER 9

Overwrite

This is a simple implementation overwrite of the FileSystemStorage. It removes the addition of an '_' to the filename if the file already exists in the storage system. I needed a model in the admin area to act exactly like a file system (overwriting the file if it already exists).

CHAPTER 10

SFTP

Take a look at the top of the backend's file for the documentation.

CHAPTER 11

Symlink or copy

Stores symlinks to files instead of actual files whenever possible

When a file that's being saved is currently stored in the symlink_within directory, then symlink the file. Otherwise, copy the file.

CHAPTER 12

Installation

Use pip to install from PyPI:

```
pip install django-storages
```

Add `storages` to your settings.py file:

```
INSTALLED_APPS = (
    ...
    'storages',
    ...
)
```

Each storage backend has its own unique settings you will need to add to your settings.py file. Read the documentation for your storage engine(s) of choice to determine what you need to add.

CHAPTER 13

Contributing

To contribute to django-storages create a fork on GitHub. Clone your fork, make some changes, and submit a pull request.

CHAPTER 14

Issues

Use the GitHub issue tracker for django-storages to submit bugs, issues, and feature requests.

CHAPTER 15

Indices and tables

- genindex
- modindex
- search