

---

# **django-storages Documentation**

***Release 1.6.2***

**David Larlet, et. al.**

**Jun 22, 2017**



---

## Contents

---

<b>1</b>	<b>Amazon S3</b>	<b>3</b>
<b>2</b>	<b>Apache Libcloud</b>	<b>9</b>
<b>3</b>	<b>Azure Storage</b>	<b>13</b>
<b>4</b>	<b>DropBox</b>	<b>15</b>
<b>5</b>	<b>FTP</b>	<b>17</b>
<b>6</b>	<b>Google Cloud Storage</b>	<b>19</b>
<b>7</b>	<b>SFTP</b>	<b>23</b>
<b>8</b>	<b>Installation</b>	<b>25</b>
<b>9</b>	<b>Contributing</b>	<b>27</b>
<b>10</b>	<b>Issues</b>	<b>29</b>
<b>11</b>	<b>Indices and tables</b>	<b>31</b>



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



### Usage

There are two backends for interacting with Amazon's S3, one based on boto3 and an older one based on boto. It is highly recommended that all new projects (at least) use the boto3 backend since it has many bug fixes and performance improvements over boto and is the future; boto is lightly maintained if at all. The boto based backed will continue to be maintained for the foreseeable future.

For historical completeness an extreme legacy backend was removed in version 1.2

If using the boto backend it is recommended that you configure it to also use [AWS Signature Version 4](#). This can be done by adding `S3_USE_SIGV4 = True` to your settings and setting the `AWS_S3_HOST` configuration option.

### Settings

To use boto3 set:

```
DEFAULT_FILE_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'
```

To use the boto version of the backend set:

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

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.s3boto3.S3Boto3Storage'
```

Available are numerous settings. It should be especially noted the following:

**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\_DEFAULT\_ACL (optional)** If set to `private` changes uploaded file's Access Control List from the default permission `public-read` to give owner full control and remove read access from everyone else.

**AWS\_AUTO\_CREATE\_BUCKET (optional)** If set to `True` the bucket specified in `AWS_STORAGE_BUCKET_NAME` is automatically created.

**AWS\_HEADERS (optional - boto only, for boto3 see AWS\_S3\_OBJECT\_PARAMETERS)** 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',
}
```

**AWS\_S3\_OBJECT\_PARAMETERS (optional - boto3 only)** Use this to set object parameters on your object (such as `CacheControl`):

```
AWS_S3_OBJECT_PARAMETERS = {
    'CacheControl': 'max-age=86400',
}
```

**AWS\_QUERYSTRING\_AUTH (optional; default is `True`)** Setting `AWS_QUERYSTRING_AUTH` to `False` removes `query parameter authentication` from generated URLs. This can be useful if your S3 buckets are public.

**AWS\_QUERYSTRING\_EXPIRE (optional; default is 3600 seconds)** The number of seconds that a generated URL is valid for.

**AWS\_S3\_ENCRYPTION (optional; default is `False`)** Enable server-side file encryption while at rest, by setting `encrypt_key` parameter to `True`. More info available here: <http://boto.cloudhackers.com/en/latest/ref/s3.html>

**AWS\_S3\_FILE\_OVERWRITE (optional: default is `True`)** By default files with the same name will overwrite each other. Set this to `False` to have extra characters appended.

`AWS_S3_HOST` (optional - boto only, default is `s3.amazonaws.com`)

To ensure you use [AWS Signature Version 4](#) it is recommended to set this to the host of your bucket. See the [mapping of region to endpoint names](#) to figure out the appropriate endpoint for your bucket. Also be sure to add `S3_USE_SIGV4 = True` to `settings.py`

**AWS\_LOCATION (optional: default is `''`)** A path prefix that will be prepended to all uploads

**AWS\_IS\_GZIPPED (optional: default is `False`)** Whether or not to enable gzipping of content types specified by `GZIP_CONTENT_TYPES`

**GZIP\_CONTENT\_TYPES (optional: default is `text/css, text/javascript, application/javascript, application/javascript`)** When `AWS_IS_GZIPPED` is set to `True` the content types which will be gzipped

**AWS\_S3\_REGION\_NAME (optional: default is `None`)** Name of the AWS S3 region to use (eg. eu-west-1)

**AWS\_S3\_USE\_SSL (optional: default is `True`)** Whether or not to use SSL when connecting to S3.

**AWS\_S3\_ENDPOINT\_URL (optional: default is `None`)** Custom S3 URL to use when connecting to S3, including scheme. Overrides `AWS_S3_REGION_NAME` and `AWS_S3_USE_SSL`.

**AWS\_S3\_CALLING\_FORMAT (optional: default is `SubdomainCallingFormat()`)** Defines the S3 calling format to use to connect to the static bucket.

`AWS_S3_SIGNATURE_VERSION` (optional - boto3 only: default is `s3v4`)

All AWS regions support the v4 version of the signing protocol. To use the legacy v2 set this to `'s3'`. Some non-Amazon S3 implementations might require this change.



## CloudFront

If you're using S3 as a CDN (via CloudFront), you'll probably want this storage to serve those files using that:

```
AWS_S3_CUSTOM_DOMAIN = 'cdn.mydomain.com'
```

**NOTE:** Django's `STATIC_URL` must end in a slash and the `AWS_S3_CUSTOM_DOMAIN` must not. It is best to set this variable independently of `STATIC_URL`.

Keep in mind you'll have to configure CloudFront to use the proper bucket as an origin manually for this to work.

If you need to use multiple storages that are served via CloudFront, pass the `custom_domain` parameter to their constructors.

## 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
```

## 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).

Use pip to install apache-libcloud from PyPI:

```
pip install apache-libcloud
```

**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.

## Settings

### 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')
```

## 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': ...
    },
}
```

## 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.

## Certificate 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_STANDARD_HOST: No CA Certificates were found in CA_CERTS_PATH.
```

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.

Before you start configuration, you will need to install the Azure SDK for Python.

Install the package:

```
pip install azure
```

Add to your requirements file:

```
pip freeze > requirements.txt
```

### Settings

To use *AzureStorage* set:

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

The following settings are available:

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/](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

---

### DropBox

---

A custom storage system for Django using Dropbox Storage backend.

Before you start configuration, you will need to install [Dropbox SDK for Python](#).

Install the package:

```
pip install dropbox
```

### Settings

To use DropBoxStorage set:

```
DEFAULT_FILE_STORAGE = 'storages.backends.dropbox.DropBoxStorage'
```

**DROPBOX\_OAUTH2\_TOKEN** Your DropBox token, if you haven't follow this [guide step](#).

**DROPBOX\_ROOT\_PATH** Allow to jail your storage to a defined directory.



**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.

## Settings

**LOCATION** URL of the server that hold the files. Example 'ftp://<user>:<pass>@<host>:<port>'

**BASE\_URL** URL that serves the files stored at this location. Defaults to the value of your MEDIA\_URL setting.



---

## Google Cloud Storage

---

### Usage

This backend provides support for Google Cloud Storage using the library provided by Google.

It's possible to access Google Cloud Storage in S3 compatibility mode using other libraries in `django-storages`, but this is the only library offering native support.

By default this library will use the credentials associated with the current instance for authentication. To override this, see the settings below.

### Settings

To use `gcloud` set:

```
DEFAULT_FILE_STORAGE = 'storages.backends.gcloud.GoogleCloudStorage'
```

`GS_BUCKET_NAME`

Your Google Storage bucket name, as a string.

`GS_PROJECT_ID` (optional)

Your Google Cloud project ID. If unset, falls back to the default inferred from the environment.

`GS_CREDENTIALS` (optional)

The OAuth 2 credentials to use for the connection. If unset, falls back to the default inferred from the environment.

`GS_AUTO_CREATE_BUCKET` (optional, default is `False`)

If `True`, attempt to create the bucket if it does not exist.

`GS_AUTO_CREATE_ACL` (optional, default is `projectPrivate`)

ACL used when creating a new bucket, from the [list of predefined ACLs](#). (A “JSON API” ACL is preferred but an “XML API/gsutil” ACL will be translated.)

Note that the ACL you select must still give the service account running the gcloud backend to have OWNER permission on the bucket. If you're using the default service account, this means you're restricted to the `projectPrivate` ACL.

`GS_FILE_CHARSET` (optional)

Allows overriding the character set used in filenames.

`GS_FILE_OVERWRITE` (optional: default is `True`)

By default files with the same name will overwrite each other. Set this to `False` to have extra characters appended.

`GS_MAX_MEMORY_SIZE` (optional)

The maximum amount of memory a returned file can take up before being rolled over into a temporary file on disk. Default is 0: Do not roll over.

## Fields

Once you're done, `default_storage` will be Google Cloud Storage:

```
>>> from django.core.files.storage import default_storage
>>> print default_storage.__class__
<class 'storages.backends.gcloud.GoogleCloudStorage'>
```

This way, if you define a new `FileField`, it will use the Google Cloud 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
<storages.backends.gcloud.GoogleCloudStorage object at ...>
```

## 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
```



## 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()
```

## Settings

**SFTP\_STORAGE\_HOST** The hostname where you want the files to be saved.

**SFTP\_STORAGE\_ROOT** The root directory on the remote host into which files should be placed. Should work the same way that `STATIC_ROOT` works for local files. Must include a trailing slash.

**SFTP\_STORAGE\_PARAMS (optional)** A dictionary containing connection parameters to be passed as keyword arguments to `paramiko.SSHClient().connect()` (do not include hostname here). See [paramiko SSH-Client.connect\(\) documentation](#) for details

**SFTP\_STORAGE\_INTERACTIVE (optional)** A boolean indicating whether to prompt for a password if the connection cannot be made using keys, and there is not already a password in `SFTP_STORAGE_PARAMS`. You can set this to `True` to enable interactive login when running `manage.py collectstatic`, for example.

**Warning:** DO NOT set `SFTP_STORAGE_INTERACTIVE` to `True` if you are using this storage for files being uploaded to your site by users, because you'll have no way to enter the password when they submit the form..

**SFTP\_STORAGE\_FILE\_MODE (optional)** A bitmask for setting permissions on newly-created files. See [Python `os.chmod` documentation](#) for acceptable values.

**SFTP\_STORAGE\_DIR\_MODE (optional)** A bitmask for setting permissions on newly-created directories. See [Python `os.chmod` documentation](#) for acceptable values.

---

**Note:** Hint: if you start the mode number with a 0 you can express it in octal just like you would when doing “`chmod 775 myfile`” from bash.

---

**SFTP\_STORAGE\_UID (optional)** UID of the account that should be set as owner of the files on the remote host. You may have to be root to set this.

**SFTP\_STORAGE\_GID (optional)** GID of the group that should be set on the files on the remote host. You have to be a member of the group to set this.

**SFTP\_KNOWN\_HOST\_FILE (optional)** Absolute path of know host file, if it isn't set "`~/ .ssh/known_hosts`" will be used.

## CHAPTER 8

---

### 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 9

---

### Contributing

---

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





## CHAPTER 10

---

### Issues

---

Use the GitHub [issue tracker](#) for django-storages to submit bugs, issues, and feature requests.



# CHAPTER 11

---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`