django-queued-storage Documentation

Release 0.8.1.dev66+gd30d70d

Jannis Leidel

Aug 20, 2017

Contents

This storage backend enables having a local and a remote storage backend. It will save any file locally and queue a task to transfer it somewhere else using Celery.

If the file is accessed before it's transferred, the local copy is returned.

CHAPTER 1

Installation

```
pip install django-queued-storage
```

CHAPTER 2

Configuration

- Follow the configuration instructions for django-celery
- Set up a caching backend
- Add `'queued_storage'` to your `INSTALLED_APPS` setting

CHAPTER 3

Usage

The *QueuedStorage* can be used as a drop-in replacement wherever using `django.core.files.storage.Storage` might otherwise be required.

This example is using django-storages for the remote backend:

```
from django.db import models
from queued_storage.backends import QueuedStorage
from storages.backends.s3boto import S3BotoStorage

queued_s3storage = QueuedStorage(
    'django.core.files.storage.FileSystemStorage',
    'storages.backends.s3boto.S3BotoStorage')

class MyModel(models.Model):
    image = ImageField(storage=queued_s3storage)
```

CHAPTER 4

Settings

queued_storage.conf.settings.**QUEUED_STORAGE_CACHE_PREFIX**

Default `'queued_storage'`

The cache key prefix to use when caching the storage backends.

queued_storage.conf.settings.**QUEUED_STORAGE_RETRIES**

Default `5`

How many retries should be attempted before aborting.

queued_storage.conf.settings.**QUEUED_STORAGE_RETRY_DELAY**

Default `60`

The delay between retries in seconds.

CHAPTER 5

Reference

For further details see the reference documentation:

Backends

class queued_storage.backends.**QueuedStorage**(*local=None*, *remote=None*, *local_options=None*, *remote_options=None*, *cache_prefix=None*, *delayed=None*, *task=None*)

Base class for queued storages. You can use this to specify your own backends.

Parameters

- **local** (*str*) – local storage class to transfer from
- **local_options** (*dict*) – options of the local storage class
- **remote** (*str*) – remote storage class to transfer to
- **remote_options** (*dict*) – options of the remote storage class
- **cache_prefix** (*str*) – prefix to use in the cache key
- **delayed** (*bool*) – whether the transfer task should be executed automatically
- **task** (*str*) – Celery task to use for the transfer

local_options = None

The options of the local storage class, defined as a dictionary.

local = None

The local storage class to use. A dotted path (e.g. `'django.core.files.storage.FileSystemStorage'`).

remote_options = None

The options of the remote storage class, defined as a dictionary.

remote = None
The remote storage class to use. A dotted path (e.g. `'django.core.files.storage.FileSystemStorage'`).

task = 'queued_storage.tasks.Transfer'
The Celery task class to use to transfer files from the local to the remote storage. A dotted path (e.g. `'queued_storage.tasks.Transfer'`).

delayed = False
If set to `True` the backend will *not* transfer files to the remote location automatically, but instead requires manual intervention by the user with the *transfer()* method.

cache_prefix = 'queued_storage'
The cache key prefix to use when saving the which storage backend to use, local or remote (default see *QUEUED_STORAGE_CACHE_PREFIX*)

get_storage(*name*)
Returns the storage backend instance responsible for the file with the given name (either local or remote). This method is used in most of the storage API methods.

Parameters **name** (*str*) – file name

Return type Storage

get_cache_key(*name*)
Returns the cache key for the given file name.

Parameters **name** (*str*) – file name

Return type str

using_local(*name*)
Determines for the file with the given name whether the local storage is current used.

Parameters **name** (*str*) – file name

Return type bool

using_remote(*name*)
Determines for the file with the given name whether the remote storage is current used.

Parameters **name** (*str*) – file name

Return type bool

open(*name*, *mode='rb'*)
Retrieves the specified file from storage.

Parameters

- **name** (*str*) – file name
- **mode** (*str*) – mode to open the file with

Return type File

save(*name*, *content*, *max_length=None*)
Saves the given content with the given name using the local storage. If the *delayed* attribute is `True` this will automatically call the *transfer()* method queuing the transfer from local to remote storage.

Parameters

- **name** (*str*) – file name
- **content** (File) – content of the file specified by name

Return type str

transfer(*name*, *cache_key=None*)
Transfers the file with the given name to the remote storage backend by queuing the task.

Parameters

- **name** (*str*) – file name
- **cache_key** (*str*) – the cache key to set after a successful task run

Return type task result

get_valid_name(*name*)
Returns a filename, based on the provided filename, that's suitable for use in the current storage system.

Parameters **name** (*str*) – file name

Return type str

get_available_name(*name*)
Returns a filename that's free on both the local and remote storage systems, and available for new content to be written to.

Parameters **name** (*str*) – file name

Return type str

path(*name*)
Returns a local filesystem path where the file can be retrieved using Python's built-in open() function. Storage systems that can't be accessed using open() should *not* implement this method.

Parameters **name** (*str*) – file name

Return type str

delete(*name*)
Deletes the specified file from the storage system.

Parameters **name** (*str*) – file name

exists(*name*)
Returns `True` if a file referened by the given name already exists in the storage system, or False if the name is available for a new file.

Parameters **name** (*str*) – file name

Return type bool

listdir(*name*)
Lists the contents of the specified path, returning a 2-tuple of lists; the first item being directories, the second item being files.

Parameters **name** (*str*) – file name

Return type tuple

size(*name*)
Returns the total size, in bytes, of the file specified by name.

Parameters **name** (*str*) – file name

Return type int

url(*name*)
Returns an absolute URL where the file's contents can be accessed directly by a Web browser.

Parameters **name** (*str*) – file name

Return type str

accessed_time(*name*)
Returns the last accessed time (as datetime object) of the file specified by name.

Parameters **name** (*str*) – file name

Return type datetime

created_time(*name*)
Returns the creation time (as datetime object) of the file specified by name.

Parameters **name** (*str*) – file name

Return type datetime

modified_time(*name*)
Returns the last modified time (as datetime object) of the file specified by name.

Parameters **name** (*str*) – file name

Return type datetime

get_accessed_time(*name*)
Django +1.10 Returns the last accessed time (as datetime object) of the file specified by name.

Parameters **name** (*str*) – file name

Return type datetime

get_created_time(*name*)
Django +1.10 Returns the creation time (as datetime object) of the file specified by name.

Parameters **name** (*str*) – file name

Return type datetime

get_modified_time(*name*)
Django +1.10 Returns the last modified time (as datetime object) of the file specified by name.

Parameters **name** (*str*) – file name

Return type datetime

generate_filename(*filename*)

class queued_storage.backends.**QueuedFileSystemStorage**(*local='django.core.files.storage.FileSystemStorage'*, **args*, ***kwargs*)
A *QueuedStorage* subclass which conveniently uses FileSystemStorage as the local storage.

class queued_storage.backends.**QueuedS3BotoStorage**(*remote='storages.backends.s3boto.S3BotoStorage'*, **args*, ***kwargs*)
A custom *QueuedFileSystemStorage* subclass which uses the S3BotoStorage storage of the django-storages app as the remote storage.

class queued_storage.backends.**QueuedCouchDBStorage**(*remote='storages.backends.couchdb.CouchDBStorage'*, **args*, ***kwargs*)
A custom *QueuedFileSystemStorage* subclass which uses the CouchDBStorage storage of the django-storages app as the remote storage.

class queued_storage.backends.**QueuedDatabaseStorage**(*remote='storages.backends.database.DatabaseStorage'*, **args*, ***kwargs*)
A custom *QueuedFileSystemStorage* subclass which uses the DatabaseStorage storage of the django-storages app as the remote storage.

class queued_storage.backends.**QueuedFTPStorage**(*remote='storages.backends.ftp.FTPStorage'*, **args*, ***kwargs*)

A custom *QueuedFileSystemStorage* subclass which uses the FTPStorage storage of the django-storages app as the remote storage.

class queued_storage.backends.**QueuedMogileFSStorage**(*remote='storages.backends.mogile.MogileFSStorage'*, **args*, ***kwargs*)

A custom *QueuedFileSystemStorage* subclass which uses the MogileFSStorage storage of the django-storages app as the remote storage.

class queued_storage.backends.**QueuedGridFSStorage**(*remote='storages.backends.mongodb.GridFSStorage'*, **args*, ***kwargs*)

A custom *QueuedFileSystemStorage* subclass which uses the GridFSStorage storage of the django-storages app as the remote storage.

class queued_storage.backends.**QueuedCloudFilesStorage**(*remote='storages.backends.mosso.CloudFilesStorage'*, **args*, ***kwargs*)

A custom *QueuedFileSystemStorage* subclass which uses the CloudFilesStorage storage of the django-storages app as the remote storage.

class queued_storage.backends.**QueuedSFTPStorage**(*remote='storages.backends.sftpstorage.SFTPStorage'*, **args*, ***kwargs*)

A custom *QueuedFileSystemStorage* subclass which uses the SFTPStorage storage of the django-storages app as the remote storage.

Fields

class queued_storage.fields.**QueuedFileField**(*verbose_name=None*, *name=None*, *upload_to=''*, *storage=None*, ***kwargs*)

Field to be used together with *QueuedStorage* instances or instances of subclasses.

Tiny wrapper around FileField, which provides a convenient method to transfer files, using the *transfer()* method, e.g.:

```
from queued_storage.backends import QueuedS3BotoStorage
from queued_storage.fields import QueuedFileField

class MyModel(models.Model):
    image = QueuedFileField(storage=QueuedS3BotoStorage(delayed=True))

my_obj = MyModel(image=File(open('image.png')))
# Save locally:
my_obj.save()
# Transfer to remote location:
my_obj.image.transfer()
```

attr_class

alias of *QueuedFieldFile*

class queued_storage.fields.**QueuedFieldFile**(*instance*, *field*, *name*)

A custom FieldFile which has an additional method to transfer the file to the remote storage using the backend's transfer method.

transfer()

Transfers the file using the storage backend.

Tasks

class queued_storage.tasks.**Transfer**

The default task. Transfers a file to a remote location. The actual transfer is implemented in the remote backend.

To use a different task, pass it into the backend:

```
from queued_storage.backends import QueuedS3BotoStorage

s3_delete_storage = QueuedS3BotoStorage(
    task='queued_storage.tasks.TransferAndDelete')

# later, in model definition:
image = models.ImageField(storage=s3_delete_storage)
```

The result should be `True` if the transfer was successful, or `False` if unsuccessful. In the latter case the task will be retried.

You can subclass the *Transfer* class to customize the behaviour, to do something like this:

```
from queued_storage.tasks import Transfer

class TransferAndNotify(Transfer):
    def transfer(self, *args, **kwargs):
        result = super(TransferAndNotify, self).transfer(*args, **kwargs)
        if result:
            # call the (imaginary) notify function with the result
            notify(result)
        return result
```

max_retries = 5

The number of retries if unsuccessful (default: see *QUEUED_STORAGE_RETRIES*)

default_retry_delay = 60

The delay between each retry in seconds (default: see *QUEUED_STORAGE_RETRY_DELAY*)

run(*name*, *cache_key*, *local_path*, *remote_path*, *local_options*, *remote_options*, ***kwargs*)

The main work horse of the transfer task. Calls the transfer method with the local and remote storage backends as given with the parameters.

Parameters

- **name** (*str*) – name of the file to transfer
- **local_path** (*str*) – local storage class to transfer from
- **local_options** (*dict*) – options of the local storage class
- **remote_path** (*str*) – remote storage class to transfer to
- **remote_options** (*dict*) – options of the remote storage class
- **cache_key** (*str*) – cache key to set after a successful transfer

Return type task result

transfer(*name*, *local*, *remote*, ***kwargs*)

Transfers the file with the given name from the local to the remote storage backend.

Parameters

- **name** – The name of the file to transfer

- **local** – The local storage backend instance
- **remote** – The remote storage backend instance

Returns *True* when the transfer succeeded, *False* if not. Retries the task when returning *False*

Return type bool

acks_late = False

delivery_mode = 2

exchange_type = u'direct'

ignore_result = False

name = u'queued_storage.tasks.Transfer'

rate_limit = None

reject_on_worker_lost = None

request_stack = <celery.utils.threads._LocalStack object>

serializer = u'json'

store_errors_even_if_ignored = False

track_started = False

typing = True

class queued_storage.tasks.**TransferAndDelete**

A *Transfer* subclass which deletes the file with the given name using the local storage if the transfer was successful.

transfer(*name*, *local*, *remote*, ***kwargs*)

name = u'queued_storage.tasks.TransferAndDelete'

rate_limit = None

reject_on_worker_lost = None

Signals

django-queued-storage ships with a signal fired after a file was transfered by the Transfer task. It provides the name of the file, the local and the remote storage backend instances as arguments to connected signal callbacks.

Imagine you'd want to post-process the file that has been transfered from the local to the remote storage, e.g. add it to a log model to always know what exactly happened. All you'd have to do is to connect a callback to the file_transferred signal:

```
from django.dispatch import receiver
from django.utils.timezone import now

from queued_storage.signals import file_transferred

from mysite.transferlog.models import TransferLogEntry


@receiver(file_transferred)
def log_file_transferred(sender, name, local, remote, **kwargs):
    remote_url = remote.url(name)
```

```
    TransferLogEntry.objects.create(name=name, remote_url=remote_url, transfer_
↪date=now())

# Alternatively, you can also use the signal's connect method to connect:
file_transferred.connect(log_file_transferred)
```

Note that this signal does **NOT** have access to the calling Model or even the FileField instance that it relates to, only the name of the file. As a result, this signal is somewhat limited and may only be of use if you have a very specific usage of django-queued-storage.

Changelog

v0.8 (2015-12-14)

- Added Django 1.9 support

v0.7.2 (2015-12-02)

- Documentation config fixes.

v0.7.1 (2015-12-02)

- Fix dependency on django-appconf.
- Minor code cleanup.

v0.7 (2015-12-02)

- Dropping Django 1.6 support
- Dropping Python 2.6 and 3.2 support
- Switched testing to use tox and py.test
- Added Python 3 support
- Switched to using setuptools_scm
- Transfered to Jazzband: https://github.com/jazzband/django-queued-storage
- Tests can be found at: http://travis-ci.org/jazzband/django-queued-storage

v0.6 (2012-05-24)

- Added `file_transferred` signal that is called right after a file has been transfered from the local to the remote storage.
- Switched to using django-discover-runner and Travis for testing: http://travis-ci.org/jezdez/django-queued-storage

v0.5 (2012-03-19)

- Fixed retrying in case of errors.
- Dropped Python 2.5 support as Celery has dropped it, too.
- Use django-jenkins.

v0.4 (2011-11-03)

- Revised storage parameters to fix an incompatibility with Celery regarding task parameter passing and pickling.

  It's now *required* to pass the dotted Python import path of the local and remote storage backend, as well as a dictionary of options for instantiation of those classes (if needed). Passing storage instances to the *QueuedStorage* class is now considered an error. For example:

  Old:

```
from django.core.files.storage import FileSystemStorage
from mysite.storage import MyCustomStorageBackend

my_storage = QueuedStorage(
    FileSystemStorage(location='/path/to/files'),
    MyCustomStorageBackend(spam='eggs'))
```

  New:

```
my_storage = QueuedStorage(
  local='django.core.files.storage.FileSystemStorage',
  local_options={'location': '/path/to/files'},
  remote='mysite.storage.MyCustomStorageBackend',
  remote_options={'spam': 'eggs'})
```

  Warning: This change is backwards-incompatible if you used the *QueuedStorage* API.

v0.3 (2011-09-19)

- Initial release.

CHAPTER 6

Issues

For any bug reports and feature requests, please use the Github issue tracker.

Python Module Index

q

Index

Q

R

S

T

U