Flask-FS Documentation

Release 0.5.0

Axel Haustant

Mar 12, 2018

Contents

Flask-FS provide a simple and flexible file storage interface for Flask. It is inspired by Django file storage.

CHAPTER 1

Documentation

This part of the documentation will show you how to get started in using Flask-FS with Flask.

1.1 Installation

Install Flask-FS with `pip`:

```
pip install flask-fs
```

Each backend has its own dependencies:

```
$ pip install flask-fs[s3]  # For Amazon S3 backend support
$ pip install flask-fs[swift]  # For OpenStack swift backend support
$ pip install flask-fs[gridfs]  # For GridFS backend support
$ pip install flask-fs[all]  # To include all dependencies for all backends
```

The development version can be downloaded from GitHub.

```
git clone https://github.com/noirbizarre/flask-fs.git
cd flask-fs
pip install -e .[dev]
```

Flask-FS requires Python version 2.6, 2.7, 3.3, 3.4 or 3.5. It's also working with PyPy and PyPy3.

1.2 Quick Start

1.2.1 Initialization

Flask-FS need to be initialized with an application:

```
from flask import Flask
import flask_fs as fs

app = Flask(__name__)
fs.init_app(app)
```

1.2.2 Storages declaration

You need to declare some storages before being able to read or write files.

```
import flask_fs as fs

images = fs.Storage('images')
uploads = fs.Storage('uploads')
```

You can limit the allowed file types.

```
import flask_fs as fs

images = fs.Storage('images', fs.IMAGES)
custom = fs.Storage('custom', ('bat', 'sh'))
```

You can also specify allowed extensions by exclusion:

```
import flask_fs as fs

WITHOUT_SCRIPTS = fs.AllExcept(fs.SCRIPTS + fs.EXECUTABLES)
store = fs.Storage('store', WITHOUT_SCRIPTS)
```

By default files in storage are not overwritables. You can allow overwriting with the *overwrite* parameter in *Storage* class.

```
import flask_fs as fs

store = fs.Storage('store', overwrite=True)
```

1.2.3 Storages operations

Storages provides an abstraction layer for common operations. All filenames are root relative to the storage.

```
store = fs.Storage('store')

# Writing
store.write('my.file', 'content')

# Reading
content = store.read('my.file')

# Working with file object
with store.open('my.file', 'wb') as f:
    # do something

# Testing file presence
if store.exists('my.file'):
```

(continues on next page)

(continued from previous page)

```
    # do something

if 'my.file' in store:
    # do something

# Deleting file
store.delete('my.file')
```

See *Storage* class definition.

1.3 Configuration

Flask-FS expose both global and by storage settings.

1.3.1 Global configuration

FS_SERVE

default: `DEBUG`

A boolean whether or not Flask-FS should serve files

FS_ROOT

default: `{app.instance_path}/fs`

The global local storage root. Each storage will have its own root as a subdirectory unless not local or overridden by configuration.

FS_PREFIX

default: `None`

An optionnal URL path prefix for storages (ex: `'/fs'`).

FS_URL

default: `None`

An optionnal URL on which the *FS_ROOT* is visible (ex: `'https://static.mydomain.com/'`).

FS_BACKEND

default: `'local'`

The default backend used for storages. Can be one of `local`, `s3`, `gridfs` or `swift`

FS_IMAGES_OPTIMIZE

default: `False`

Whether or not image should be compressedd/optimized by default.

1.3.2 Storages configuration

Each storage configuration can be overriden from the application configuration. The configuration is loaded in the following order:

- `FS_{BACKEND_NAME}_{KEY}` (backend specific configuration)
- `{STORAGE_NAME}_FS_{KEY}` (specific configuration)
- `FS_{KEY}` (global configuration)
- default value

Given a storage declared like this:

```
import flask_fs as fs

avatars = fs.Storage('avatars', fs.IMAGES)
```

You can override its root with the following configuration:

```
AVATARS_FS_ROOT = '/somewhere/on/the/filesystem'
```

Or you can set a base URL to all storages for a given backend:

```
FS_S3_URL = 'https://s3.somewhere.com/'
FS_S3_REGION = 'us-east-1'
```

1.4 Backends

1.4.1 Local backend (`local`)

A local file system storage. This is the default storage backend.

Expect the following settings:

- `ROOT`: The file system root

1.4.2 S3 backend (`s3`)

An Amazon S3 Backend (compatible with any S3-like API)

Expect the following settings:

- `ENDPOINT`: The S3 API endpoint
- `REGION`: The region to work on.
- `ACCESS_KEY`: The AWS credential access key
- `SECRET_KEY`: The AWS credential secret key

1.4.3 GridFS backend (`gridfs`)

A Mongo GridFS backend

Expect the following settings:

- `MONGO_URL`: The Mongo access URL
- `MONGO_DB`: The database to store the file in.

1.4.4 Swift backend (`swift`)

An OpenStack Swift backend

Expect the following settings:

- `AUTHURL`: The Swift Auth URL
- `USER`: The Swift user in
- `KEY`: The user API Key

1.4.5 Custom backends

Flask-FS allows you to defined your own backend by extending the *`BaseBackend`* class.

You need to register your backend using setuptools entrypoints in your `setup.py`:

```
entry_points={
    'fs.backend': [
        'custom = my.custom.package:CustomBackend',
    ]
},
```

1.4.6 Sample configuration

Given these storages:

```
import flask_fs as fs

files = fs.Storage('files')
avatars = fs.Storage('avatars', fs.IMAGES)
images = fs.Storage('images', fs.IMAGES)
```

Here an example configuration with local files storages and s3 images storage:

```
# Shared S3 configuration
FS_S3_ENDPOINT = 'https://s3-eu-west-2.amazonaws.com'
FS_S3_REGION = 'eu-west-2'
FS_S3_ACCESS_KEY = 'ABCDEFGHIJKLMNOQRSTU'
FS_S3_SECRET_KEY = 'abcdefghiklmnoqrstuvwxyz1234567890abcdef'
FS_S3_URL = 'https://s3.somewhere.com/'

# storage specific configuration
AVATARS_FS_BACKEND = 's3'
IMAGES_FS_BACKEND = 's3'
```

(continues on next page)

(continued from previous page)

```
FILES_FS_URL = 'https://images.somewhere.com/'
FILES_FS_URL = 'https://files.somewhere.com/'
```

In this configuration, storages will have the following configuration:

- `files`: `local` storage served on `https://files.somewhere.com/`
- `avatars`: `s3` storage served on `https://s3.somewhere.com/avatars/`
- `images`: `s3` storage served on `https://images.somewhere.com/`

1.5 Mongoengine support

Flask-FS provides a thin mongoengine integration as `field classes`.

Both *`FileField`* and *`ImageField`* provides a common interface:

```
images = fs.Storage('images', fs.IMAGES,
                    upload_to=lambda o: 'prefix',
                    basename=lambda o: 'basename')

class MyDoc(Document):
    file = FileField(fs=files)


doc = MyDoc()

# Test file presence
print(bool(doc.file))  # False
# Get filename
print(doc.file.filename)  # None
# Get file URL
print(doc.file.url)  # None
# Print file URL
print(str(doc.file))  # ''

doc.file.save(io.Bytes(b'xxx'), 'test.file')

print(bool(doc.file))  # True
print(doc.file.filename)  # 'test.file'
print(doc.file.url)  # 'http://myserver.com/files/prefix/test.file'
print(str(doc.file))  # 'http://myserver.com/files/prefix/test.file'

# Override Werkzeug Filestorage filename with basename
f = FileStorage(io.Bytes(b'xxx'), 'test.file')
doc.file.save(f)
print(doc.file.filename)  # 'basename.file'
```

The *`ImageField`* provides some extra features.

On declaration:

- an optionnal *max_size* attribute allows to limit image size
- an optionnal *thumbnails* list of thumbnail sizes to be generated
- an optionnal *optimize* booleanoverriding the `FS_IMAGES_OPTIMIZE` setting by field.

On instance:

- the *original* property gives the unmodified image filename
- the *best_url(size)* method match a thumbnail URL given a size
- the *thumbnail(size)* method get a thumbnail filename given a registered size
- the *save* method accept an optionnal *bbox* kwarg for to crop the thumbnails
- the *rerender* method allows to force a new image rendering (taking in account new parameters)
- the instance is callable as shortcut for *best_url()*

```
images = fs.Storage('images', fs.IMAGES)
files = fs.Storage('files', fs.ALL)

class MyDoc(Document):
    image = ImageField(fs=images,
                       max_size=150,
                       thumbnails=[100, 32])


doc = MyDoc()

with open(some_image, 'rb') as f:
    doc.file.save(f, 'test.png')

print(doc.image.filename)  # 'test.png'
print(doc.image.original)  # 'test-original.png'
print(doc.image.thumbnail(100))  # 'test-100.png'
print(doc.image.thumbnail(32))  # 'test-32.png'

# Guess best image url for a given size
assert doc.image.best_url().endswith(doc.image.filename)
assert doc.image.best_url(200).endswith(doc.image.filename)
assert doc.image.best_url(150).endswith(doc.image.filename)
assert doc.image.best_url(100).endswith(doc.image.thumbnail(100))
assert doc.image.best_url(90).endswith(doc.image.thumbnail(100))
assert doc.image.best_url(30).endswith(doc.image.thumbnail(32))

# Call as shortcut for best_url()
assert doc.image().endswith(doc.image.filename)
assert doc.image(200).endswith(doc.image.filename)
assert doc.image(150).endswith(doc.image.filename)
assert doc.image(100).endswith(doc.image.thumbnail(100))

# Save an optionnal bbox for thumbnails cropping
bbox = (10, 10, 100, 100)
with open(some_image, 'rb') as f:
    doc.file.save(f, 'test.png', bbox=bbox)
```

1.6 API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

1.6.1 API

Core

flask_fs.**by_name**(*name*)
Get a storage by its name

flask_fs.**init_app**(*app*, **storages*)
Initialize Storages configuration Register blueprint if necessary.

Parameters

- **app** – The *~flask.Flask* instance to get the configuration from.
- **storages** – A *Storage* instance list to register and configure.

class flask_fs.**Storage**(*name=u'files', extensions=[u'txt', u'rtf', u'odf', u'ods', u'gnumeric', u'abw', u'doc', u'docx', u'xls', u'xlsx', u'jpg', u'jpe', u'jpeg', u'png', u'gif', u'svg', u'bmp', u'csv', u'ini', u'json', u'plist', u'xml', u'yaml', u'yml'], upload_to=None, overwrite=False*)
This represents a single set of files. Each Storage is independent of the others. This can be reused across multiple application instances, as all configuration is stored on the application object itself and found with *flask.current_app*.

Parameters

- **name** (*str*) – The name of this storage. It defaults to `files`, but you can pick any alphanumeric name you want.
- **extensions** (*tuple*) – The extensions to allow uploading in this storage. The easiest way to do this is to add together the extension presets (for example, `TEXT + DOCUMENTS + IMAGES`). It can be overridden by the configuration with the *{NAME}_FS_ALLOW* and *{NAME}_FS__DENY* configuration parameters. The default is *DEFAULTS*.
- **upload_to** (*str|callable*) – If given, this should be a callable. If you call it with the app, it should return the default upload destination path for that app.
- **overwrite** (*bool*) – Whether or not to allow overwriting

base_url
The public URL for this storage

configure(*app*)
Load configuration from application configuration.

For each storage, the configuration is loaded with the following pattern:

```
FS_{BACKEND_NAME}_{KEY} then
{STORAGE_NAME}_FS_{KEY}
```

If no configuration is set for a given key, global config is taken as default.

delete(*filename*)
Delete a file.

Parameters **filename** (*str*) – The storage root-relative filename

exists(*filename*)
Verify whether a file exists or not.

extension_allowed(*ext*)
This determines whether a specific extension is allowed. It is called by *file_allowed*, so if you override that but still want to check extensions, call back into this.

Parameters **ext** (*str*) – The extension to check, without the dot.

file_allowed(*storage*, *basename*)
This tells whether a file is allowed.

It should return *True* if the given `FileStorage` object can be saved with the given basename, and *False* if it can't. The default implementation just checks the extension, so you can override this if you want.

Parameters

- **storage** – The *werkzeug.FileStorage* to check.
- **basename** – The basename it will be saved under.

has_url
Whether this storage has a public URL or not

list_files()
Returns a filename generator to iterate through all the file in the storage bucket

metadata(*filename*)
Get some metadata for a given file.

Can vary from a backend to another but some are always present: - *filename*: the base filename (without the path/prefix) - *url*: the file public URL - *checksum*: a checksum expressed in the form *algo:hash* - 'mime': the mime type - *modified*: the last modification date

open(*filename*, *mode=u'r'*, ***kwargs*)
Open the file and return a file-like object.

Parameters

- **filename** (*str*) – The storage root-relative filename
- **mode** (*str*) – The open mode (`(r|w)b?`)

Raises ***FileNotFound*** – If trying to read a file that does not exists

path(*filename*)
This returns the absolute path of a file uploaded to this set. It doesn't actually check whether said file exists.

Parameters

- **filename** – The filename to return the path for.
- **folder** – The subfolder within the upload set previously used to save to.

Raises ***OperationNotSupported*** – when the backenddoesn't support direct file access

read(*filename*)
Read a file content.

Parameters **filename** (*string*) – The storage root-relative filename

Raises ***FileNotFound*** – If the file does not exists

resolve_conflict(*target_folder*, *basename*)
If a file with the selected name already exists in the target folder, this method is called to resolve the conflict. It should return a new basename for the file.

The default implementation splits the name and extension and adds a suffix to the name consisting of an underscore and a number, and tries that until it finds one that doesn't exist.

Parameters

- **target_folder** (*str*) – The absolute path to the target.

- **basename** (*str*) – The file's original basename.

save (*file_or_wfs*, *filename=None*, *prefix=None*, *overwrite=None*)
Saves a *file* or a `FileStorage` into this storage.

If the upload is not allowed, an `UploadNotAllowed` error will be raised. Otherwise, the file will be saved and its name (including the folder) will be returned.

Parameters

- **file_or_wfs** – a file or `werkzeug.FileStorage` file to save.
- **filename** (*string*) – The expected filename in the storage. Optionnal with a `FileStorage` but allow to override clietn value
- **prefix** (*string*) – a path or a callable returning a path to be prepended to the filename.
- **overwrite** (*bool*) – if specified, override the storage default value.

Raises ***UnauthorizedFileType*** – If the file type is not allowed

serve (*filename*)
Serve a file given its filename

url (*filename*, *external=False*)
This function gets the URL a file uploaded to this set would be accessed at. It doesn't check whether said file exists.

Parameters

- **filename** (*string*) – The filename to return the URL for.
- **external** (*bool*) – If True, returns an absolute URL

write (*filename*, *content*, *overwrite=False*)
Write content to a file.

Parameters

- **filename** (*str*) – The storage root-relative filename
- **content** – The content to write in the file
- **overwrite** (*bool*) – Whether to wllow overwrite or not

Raises ***FileExists*** – If the file exists and *overwrite* is *False*

File types

flask_fs.**TEXT = [u'txt']**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

flask_fs.**DOCUMENTS = [u'rtf', u'odf', u'ods', u'gnumeric', u'abw', u'doc', u'docx', u'xls',**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

flask_fs.**IMAGES = [u'jpg', u'jpe', u'jpeg', u'png', u'gif', u'svg', u'bmp']**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

flask_fs.**AUDIO = [u'wav', u'mp3', u'aac', u'ogg', u'oga', u'flac']**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

flask_fs.**DATA = [u'csv', u'ini', u'json', u'plist', u'xml', u'yaml', u'yml']**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

flask_fs.**SCRIPTS = [u'js', u'php', u'pl', u'py', u'rb', u'sh', u'bat']**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

flask_fs.**ARCHIVES = [u'gz', u'bz2', u'zip', u'tar', u'tgz', u'txz', u'7z']**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

flask_fs.**EXECUTABLES = [u'so', u'exe', u'dll']**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

flask_fs.**DEFAULTS = [u'txt', u'rtf', u'odf', u'ods', u'gnumeric', u'abw', u'doc', u'docx',**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

flask_fs.**ALL = <flask_fs.files.All object>**
This "contains" all items. You can use it to allow all extensions to be uploaded.

class flask_fs.**All**
This type can be used to allow all extensions. There is a predefined instance named *ALL*.

class flask_fs.**AllExcept**(*items*)
This can be used to allow all file types except certain ones.

For example, to exclude .exe and .iso files, pass:

```
AllExcept(('exe', 'iso'))
```

to the *Storage* constructor as *extensions* parameter.

You can use any container, for example:

```
AllExcept(SCRIPTS + EXECUTABLES)
```

This module handle image operations (thumbnailing, resizing…)

flask_fs.images.**make_thumbnail**(*file*, *size*, *bbox=None*)
Generate a thumbnail for a given image file.

Parameters

- **file** (*file*) – The source image file to thumbnail
- **size** (*int*) – The thumbnail size in pixels (Thumbnails are squares)
- **bbox** (*tuple*) – An optionnal Bounding box definition for the thumbnail

Backends

class flask_fs.backends.**BaseBackend**(*name*, *config*)
Abstract class to implement backend.

as_binary(*content*, *encoding=u'utf8'*)
Perform content encoding for binary write

delete(*filename*)
Delete a file given its filename in the storage

exists(*filename*)
Test wether a file exists or not given its filename in the storage

open(*filename*, **args*, ***kwargs*)
Open a file given its filename relative to the storage root

read(*filename*)
Read a file content given its filename in the storage

save (*file_or_wfs*, *filename*, *overwrite=False*)
Save a file-like object or a *werkzeug.FileStorage* with the specified filename.

Parameters

- **storage** – The file or the storage to be saved.
- **filename** – The destination in the storage.
- **overwrite** – if *False*, raise an exception if file exists in storage

Raises ***FileExists*** – when file exists and overwrite is *False*

serve (*filename*)
Serve a file given its filename

write (*filename*, *content*)
Write content into a file given its filename in the storage

class flask_fs.backends.local.**LocalBackend** (*name*, *config*)
A local file system storage

Expect the following settings:

- *root*: The file system root

metadata (*filename*)
Fetch all availabe metadata

path (*filename*)
Return the full path for a given filename in the storage

serve (*filename*)
Serve files for storages with direct file access

class flask_fs.backends.s3.**S3Backend** (*name*, *config*)
An Amazon S3 Backend (compatible with any S3-like API)

Expect the following settings:

- *endpoint*: The S3 API endpoint
- *region*: The region to work on.
- *access_key*: The AWS credential access key
- *secret_key*: The AWS credential secret key

metadata (*filename*)
Fetch all availabe metadata

class flask_fs.backends.swift.**SwiftBackend** (*name*, *config*)
An OpenStack Swift backend

Expect the following settings:

- *authurl*: The Swift Auth URL
- *user*: The Swift user in
- *key*: The user API Key

class flask_fs.backends.gridfs.**GridFsBackend** (*name*, *config*)
A Mongo GridFS backend

Expect the following settings:

- *mongo_url*: The Mongo access URL

- *mongo_db*: The database to store the file in.

Mongo

class flask_fs.mongo.**FileField**(*fs=None*, *upload_to=None*, *basename=None*, **args*, ***kwargs*)
Store reference to files in a given storage.

proxy_class
alias of *FileReference*

to_mongo(*value*)
Convert a Python type to a MongoDB-compatible type.

to_python(*value*)
Convert a MongoDB-compatible type to a Python type.

class flask_fs.mongo.**FileReference**(*fs=None*, *filename=None*, *upload_to=None*, *basename=None*, *instance=None*, *name=None*)
Implements the FileField interface

save(*wfs*, *filename=None*)
Save a Werkzeug FileStorage object

class flask_fs.mongo.**ImageField**(*max_size=None*, *thumbnails=None*, *optimize=None*, **args*, ***kwargs*)
Store reference to images in a given Storage.

Allow to automatically generate thumbnails or resized image. Original image always stay untouched.

proxy_class
alias of *ImageReference*

class flask_fs.mongo.**ImageReference**(*original=None*, *max_size=None*, *thumbnail_sizes=None*, *thumbnails=None*, *bbox=None*, *optimize=None*, ***kwargs*)
Implements the ImageField interface

best_url(*size=None*, *external=False*)
Provide the best thumbnail for downscaling.

If there is no match, provide the bigger if exists or the original

rerender()
Rerender all derived images from the original. If optmization settings or expected sizes changed, they will be used for the new rendering.

save(*file_or_wfs*, *filename=None*, *bbox=None*, *overwrite=None*)
Save a Werkzeug FileStorage object

thumbnail(*size*)
Get the thumbnail filename for a given size

Errors

These are all errors used accross this extensions.

exception flask_fs.errors.**FSError**
Base class for all Flask-FS Exceptions

exception flask_fs.errors.**FileExists**
Raised when trying to overwrite an existing file

exception flask_fs.errors.**FileNotFound**
Raised when trying to access a non existant file

exception flask_fs.errors.**UnauthorizedFileType**
This exception is raised when trying to upload an unauthorized file type.

exception flask_fs.errors.**UploadNotAllowed**
Raised when trying to upload into storage where upload is not allowed.

exception flask_fs.errors.**OperationNotSupported**
Raised when trying to perform an operation not supported by the current backend

Internals

These are internal classes or helpers. Most of the time you shouldn't have to deal directly with them.

class flask_fs.storage.**Config**
Wrap the configuration for a single `Storage`.

Basically, it's an ObjectDict

1.7 Additional Notes

1.7.1 Contributing

Flask-FS is open-source and very open to contributions.

Submitting issues

Issues are contributions in a way so don't hesitate to submit reports on the official bugtracker.

Provide as much informations as possible to specify the issues:

- the flask-fs version used
- a stacktrace
- installed applications list
- a code sample to reproduce the issue
- ...

Submitting patches (bugfix, features, ...)

If you want to contribute some code:

1. fork the official Flask-FS repository
2. create a branch with an explicit name (like `my-new-feature` or `issue-XX`)
3. do your work in it
4. rebase it on the master branch from the official repository (cleanup your history by performing an interactive rebase)
5. submit your pull-request

There are some rules to follow:

- your contribution should be documented (if needed)
- your contribution should be tested and the test suite should pass successfully
- your code should be mostly PEP8 compatible with a 120 characters line length
- your contribution should support both Python 2 and 3 (use `tox` to test)

You need to install some dependencies to develop on Flask-FS:

```
$ pip install -e .[dev]
```

An Invoke `tasks.py` is provided to simplify the common tasks:

```
$ inv -l
Available tasks:

  all      Run tests, reports and packaging
  clean    Cleanup all build artifacts
  cover    Run tests suite with coverage
  dist     Package for distribution
  doc      Build the documentation
  qa       Run a quality report
  start    Start the middlewares (docker)
  stop     Stop the middlewares (docker)
  test     Run tests suite
  tox      Run tests against Python versions
```

You can launch invoke without any parameters, it will:

- start `docker` middlewares containers (ensure docker and docker-compose are installed)
- execute tox to run tests on all supported Python version
- build the documentation
- execute flake8 quality report
- build a distributable wheel

Or you can execute any task on demand. By exemple, to only run tests in the current Python environment and a quality report:

```
$ inv test qa
```

1.7.2 Changelog

0.5.0 (2018-03-12)

- Added *metadata* method to *Storage* to retrieve file metadata
- Force *boto3* >= *1.4.5* because of API change (lifecycle)
- Drop Python 3.3 support
- Create parent directories when opening a local file in write mode

0.4.1 (2017-06-24)

- Fix broken packaging for Python 2.7

0.4.0 (2017-06-24)

- Added backend level configuration `FS_{BACKEND_NAME}_{KEY}`
- Improved backend documentation
- Use setuptools entry points to register backends.
- Added *NONE* extensions specification
- Added *list_files* to *Storage* to list the current bucket files
- Image optimization preserve file type as much as possible
- Ensure images are not overwritted before rerendering

0.3.0 (2017-03-05)

- Switch to pytest
- `ImageField` optimization/compression. Resized images are now compressed. Default image can also be optimized on upload with `FS_IMAGES_OPTIMIZE = True` or by specifying *optimize=True* as field parameter.
- `ImageField` has now the ability to rerender images with the `rerender()` method.

0.2.1 (2017-01-17)

- Expose Python 3 compatibility

0.2.0 (2016-10-11)

- Proper github publication
- Initial S3, GridFS and Swift backend implementations
- Python 3 fixes

0.1 (2015-04-07)

- Initial release

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

R

S

T

U

W