Flask-Store

Release 0.0.4.3

April 06, 2016

Contents

`Flask-Store` is a Flask Extension designed to provide easy file upload handling in the same vien as Django-Storages, allowing developers to user custom storage backends or one of the provided storage backends.

Warning: This Flask Extenstion is under heavy development. It is likely API's will change over time but will be versioned so you can always stick to a version that works for you.

CHAPTER 1

Example Usage

```python
from flask import Flask, request
from flask.ext.store import Store

app = Flask(__name__)
app.config['STORE_DOMAIN'] = 'http://127.0.0.1:5000'
app.config['STORE_PATH'] = '/some/path/to/somewhere'
store = Store(app)

@app.route('/upload', methods=['POST', ])
def upload():
    provider = store.Provider(request.files.get('afile'))
    provider.save()

    return provider.absolute_url

if __name__ == "__main__":
    app.run()
```

CHAPTER 2

Included Providers

- Local File System
- AWS Simple Storage Service (S3)

CHAPTER 3

Usage Documentation

3.1 Installation

Simply grab it from PyPI:

```
pip install Flask-Store
```

3.2 Quick Start

Getting up and running with Flask-Store is pretty easy. By default Flask-Store will use local file system storage to store your files. All you need to do is to tell it where you want your uploaded files to live.

3.2.1 Step 1: Integration

First lets initialise the Flask-Store extension with our Flask application object.

```
from flask import Flask
from flask.ext.store import Store

app = Flask(__name__)
store = Store(app)

if __name__ == "__main__":
    app.run()
```

That is all there is to it. If you use an application factory then you can use *flask_store.Store.init_app()* method instead:

```
from flask import Flask
from flask.ext.store import Store

store = Store()

def create_app():
    app = Flask(__name__)
    store.init_app(app)

if __name__ == "__main__":
    app.run()
```

3.2.2 Step 2: Configuration

So all we need to do now is tell Flask-Store where to save files once they have been uploaded. For asolute url generation we also need to tell Flask-Store about the domain where the files can accessed.

To do this we just need to set a configuration variable called `STORE_PATH` and `STORE_DOMAIN`.

For brevity we will not show the application factory way because its pretty much identical.

```
from flask import Flask
from flask.ext.store import Store

app = Flask(__name__)
app.config['STORE_DOMAIN'] = 'http://127.0.0.1:5000'
app.config['STORE_PATH'] = '/some/path/to/somewhere'
store = Store(app)

if __name__ == "__main__":
    app.run()
```

Now when Flask-Store saves a file it will be located here: `/some/path/to/somewhere`.

3.2.3 Step 3: Add a route

Now we just need to save a file. We just need a route which gets a file from the request object and send it to our Flask-Store Provider (by default local Storage) to save it.

Note: It is important to note the Flask-Store makes no attempt to validate your file size, extensions or what not, it just does one thing and that is save files somewhere. So if you need validation you should use something like `WTForms` to validate incoming data from the user.

```
from flask import Flask, request
from flask.ext.store import Store

app = Flask(__name__)
app.config['STORE_DOMAIN'] = 'http://127.0.0.1:5000'
app.config['STORE_PATH'] = '/some/path/to/somewhere'
store = Store(app)

@app.route('/upload', methods=['POST', ])
def upload():
    provider = store.Provider(request.files.get('afile'))
    provider.save()

    return provider.absolute_url

if __name__ == "__main__":
    app.run()
```

Now if we were to `curl` a file to our upload route we should get a url back which tells how we can access it.

```
curl -i -F afile=@localfile.jpg http://127.0.0.1:5000/upload
```

We should get back something like:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 44
Server: Werkzeug/0.9.6 Python/2.7.5
Date: Thu, 17 Jul 2014 11:32:02 GMT

http://127.0.0.1:5000/uploads/localfile.jpg%
```

Now if you went to `http://127.0.0.1:5000/uploads/localfile.jpg` in your browser you should see the image you uploaded. That is because Flask-Store automatically registers a route for serving files.

Note: By the way, if you don't like the url you can change it by setting `STORE_URL_PREFIX` in your application configuration.

3.2.4 Step 4: There is no Step 4

Have a beer (or alcoholic beverage (or not) of your choice), that was exhausting.

3.3 SQLAlchemy

If you use SQLAlchemy to store data in a database you can take advantage of the bundled *`flask_store.sqla.FlaskStoreType`* which will take a lot of the cruft away for you.

Note: In the following examples we are assuming you have your application setup using the application factory pattern. We will also not show the application factory method.

3.3.1 Model

As normal you would use Flask-SQLAlchemy to define your model but you would use the *`flask_store.sqla.FlaskStoreType`* type when defining the field type for the field you want to store the file path too.

```
from flask_store.sqla import FlaskStoreType
from yourapp.ext import db

class MyModel(db.Model):
    field = db.Column(FlaskStoreType(128, location='/some/where'))
```

This will act as a standard unicode string field. You do not need to pass a `max_length` integer as we have here as this will default to `256`.

The `location` keyword argument we have passed as an optional **relative** path to where your file should be saved too from the `STORE_PATH` defined in your Flask Application Configuration as described in the Quick Start guide.

3.3.2 Saving

When wanting to save the file you just need to set the attribute to be the instance of the request file uploaded, this will save the file to the location.

```
from yourapp import create_app
from yourapp.ext import db
from yourapp.models import MyModel

app = create_app()

@route('/foo')
def foo():
    foo = MyModel()
    foo.field = request.files.get('foo')

    db.session.add(foo)
    db.session.commit()

    return foo.absolute_url
```

3.3.3 Accessing

When accessing an object the relative path stored in the database will be automatically converted to a store provider instance. This will give you access to the object:

```
from yourapp import create_app
from yourapp.ext import db
from yourapp.models import MyModel

app = create_app()

@route('/bar')
def foo():
    foo = MyModel.query.get(1)

    return foo.absolute_url
```

3.4 Local Store

Note: This document assumes you have already read the Quick Start guide.

As we discussed in the Quick Start guide Flask-Store uses the `flask_store.providers.local.LocalProvidder` class as the default provider and here we will discuss some of the more advanced concepts of this store provider.

3.4.1 Enable

This is the default provider but if you wish to be explicit (+1) then simply set the following in your application configuration:

```
STORE_PROVIDER='flask_store.providers.local.LocalProvider'
```

3.4.2 Configuration

The following configuration variables are available for you to customise.

Name	Example Value
`STORE_PATH`	`/somewhere/on/disk`
This tells Flask-Store where to save uploaded files too. For this provider it must be an absolute path to a location on disk you have permission to write too. If the directory does not exist the provider will attempt to create the directory	
`STORE_URL_PREFIX`	`/uploads`
Used to generate the URL for the uploaded file. The `LocalStore` will automatically register a route with your Flask application so the file can be accessed. Do not place domains in the path prefix.	

3.5 S3 Store

Note: This document assumes you have already read the Quick Start guide.

The S3 Store allows you to forward your uploaded files up to an AWS Simple Storage Service (S3) bucket. This takes the problem of storing large numbers of files away from you onto Amazon.

Note: Amazon's `boto` is required. Boto is not included as a install requirement for Flask-Store as not everyone will want to use the S3 provider. To install just run:

```
pip install boto
```

3.5.1 Enable

To use this provider simply set the following in your application configuration:

```
STORE_PROVIDER='flask_store.providers.s3.S3Provider'
```

3.5.2 Configuration

The following configuration variables are availible to you.

Name	Example Value
`STORE_PATH`	`/some/place/in/bucket`
For the `S3Provider` is basically your key name prefix rather than an actual location. So for the example value above the key for a file might be: `/some/place/in/bucket/foo.jpg`	
`STORE_DOMAIN`	`https://bucket.s3.amazonaws.com`
Your S3 bucket domain, this is used to generate an absolute url.	
`STORE_S3_REGION`	`us-east-1`
The region in which your bucket lives	
`STORE_S3_BUCKET`	`your.bucket.name`
The name of the S3 bucket to upload files too	
`STORE_S3_ACCESS_KEY`	`ABCDEFG12345`
Your AWS access key which has permission to upload files to the `STORE_S3_BUCKET`.	
`STORE_S3_SECRET_KEY`	`ABCDEFG12345`
Your AWS access secret key	
`STORE_S3_ACL` | `public-read`	
ACL to set uploaded files, defaults to `private`, see S3_ACL	

3.6 S3 Gevent Store

Note: This document assumes you have already read the Quick Start guide.

The *`flask_store.providers.s3.S3GeventProvider`* allows you to run the upload to S3 process in a Gevent Greenlet process. This allows your webserver to send a response back to the client whilst the upload to S3 happends in the background.

Obviously this means that when the request has finished the upload may not have finished and the key not exist in the bucket. You will need to build your application around this.

Note: The `gevent` package is required. Gevent is not included as a install requirement for Flask-Store as not everyone will want to use the S3 Gevent provider. To install just run:

```
pip install gevent
```

3.6.1 Enable

To use this provider simply set the following in your application configuration:

```
STORE_PROVIDER='flask_store.providers.s3.S3GeventProvider'
```

3.6.2 Configuration

Note: This is a sub class of *`flask_store.providers.s3.S3Provider`* and therefore all the same confiuration options apply.

CHAPTER 4

Reference

4.1 API Reference

4.1.1 flask_store

Adds simple file handling for different providers to your application. Provides the following providers out of the box:

- Local file storeage
- Amazon Simple File Storage (requires `boto` to be installed)

class `flask_store.`**`Store`**(*app=None*)

Flask-Store integration into Flask applications. Flask-Store can be integrated in two different ways depending on how you have setup your Flask application.

You can bind to a specific flask application:

```
app = Flask(__name__)
store = Store(app)
```

Or if you use an application factory you can use *`flask_store.Store.init_app()`*:

```
store = Store()
def create_app():
    app = Flask(__name__)
    store.init_app(app)
    return app
```

`check_config`(*app*)

Checks the required application configuration variables are set in the flask application.

Parameters **`app`** (*`flask.app.Flask`*) – Flask application instance

Raises `NotConfiguredError` – In the event a required config parameter is required by the Store.

`init_app`(*app*)

Sets up application default confugration options and sets a `Provider` property which can be used to access the default provider class which handles the saving of files.

Parameters **`app`** (*`flask.app.Flask`*) – Flask application instance

`provider`(*app*)

Fetches the provider class as defined by the application configuration.

Parameters **`app`** (*`flask.app.Flask`*) – Flask application instance

Raises `ImportError` – If the class or module cannot be imported

Returns The provider class

Return type class

register_route(*app*)
Registers a default route for serving uploaded assets via Flask-Store, this is based on the absolute and relative paths defined in the app configuration.

Parameters **app** (*flask.app.Flask*) – Flask application instance

set_provider_defaults(*app*)
If the provider has a `app_defaults` static method then this simply calls that method. This will set sensible application configuration options for the provider.

Parameters **app** (*flask.app.Flask*) – Flask application instance

class `flask_store.`**StoreState**(*store*, *app*)
Stores the state of Flask-Store from application init.

`flask_store.`**store_provider**()
Returns the default provider class as defined in the application configuration.

Returns The provider class

Return type class

4.1.2 flask_store.exceptions

Custom Flask-Store exception classes.

exception `flask_store.exceptions.`**NotConfiguredError**
Raise this exception in the event the flask application has not been configured properly.

4.1.3 flask_store.sqlalchemy

Custom SQLAlchemy types for handling Flask-Store instances in SQLAlchemy.

class `flask_store.sqla.`**FlaskStoreType**(*max_length=256*, *location=None*, **args*, ***kwargs*)
A SQL Alchemy custom type which will save a file using the Flask Application Configured Store Provider and saves the relative path the the database.

Also creates a fresh provider instance when accessing the data attribute from an instance.

Example

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_store import Store
from flask_store.sqla import FlaskStoreType

app = Flask(__name__)

db = SQLAlchemy(app)
store = Store(app)

class MyModel(db.Model):
    field = db.Column(FlaskStoreType(location='/some/place'))
```

`impl` = **Unicode(length=256)**
Implements a standard unicode type

`process_bind_param`(*value*, *dialect*)
Called when setting the value be stored in the database field, this will be the files relative file path.

Parameters

- **`value`** (*`werkzeug.datastructures.FileStorage`*) – The uploaded file to save
- **`dialect`** (*`sqlalchemy.engine.interfaces.Dialect`*) – The dialect

Returns The files realtive path on what ever storage backend defined in the Flask Application configuration

Return type str

`process_result_value`(*value*, *dialect*)
Called when accessing the value from the database and returning the appropriate provider file wrapper.

Parameters

- **`value`** (*`str`*) – The stored relative path in the database
- **`dialect`** (*`sqlalchemy.engine.interfaces.Dialect`*) – The dialect

Returns An instance of the Store Provider class

Return type obj

4.1.4 flask_store.utils

`flask_store.utils.`**`is_directory`**(*f*)
Checks if an object is a string, and that it points to a directory. Taken from Pillow, all credit goes to the Pillow / PIL team.

Parameters **`f`** – Could be anything

Returns Is a path to a directory or not

Return type bool

`flask_store.utils.`**`is_path`**(*f*)
Determines if the passed argument is a string or not, if is a string it is assumed to be a path. Taken from Pillow, all credit goes to the Pillow / PIL team.

Parameters **`f`** – Could be anything

Returns Is a string or not

Return type bool

`flask_store.utils.`**`path_to_uri`**(*path*)
Swaps for / Other stuff will happen here in the future.

4.1.5 flask_store.providers

Base store functionality and classes.

class `flask_store.providers.`**`Provider`**(*fp*, *location=None*)

Base provider class all storage providers should inherit from. This class provides some of the base functionality for all providers. Override as required.

`absolute_path`

Returns the absollute file path to the file.

Returns Absolute file path

Return type str

`absolute_url`

Absolute url contains a domain if it is set in the configuration, the url predix, location and the actual file name.

Returns Full absolute URL to file

Return type str

`exists`(**args*, ***kwargs*)

Placeholder "exists" method. This should be overridden by custom providers and return a `boolean` depending on if the file exists of not for the provider.

Raises `NotImplementedError` – If the "exists" method has not been implemented

`join`(**args*, ***kwargs*)

Each provider needs to implement how to safely join parts of a path together to result in a path which can be used for the provider.

Raises `NotImplementedError` – If the "join" method has not been implemented

`register_route` **= False**

By default Providers do not require a route to be registered

`relative_path`

Returns the relative path to the file, so minus the base path but still includes the location if it is set.

Returns Relative path to file

Return type str

`relative_url`

Returns the relative URL, basically minus the domain.

Returns Realtive URL to file

Return type str

`safe_filename`(*filename*)

If the file already exists the file will be renamed to contain a short url safe UUID. This will avoid overwtites.

Parameters **`filename`** (*`str`*) – A filename to check if it exists

Returns A safe filenaem to use when writting the file

Return type str

`save`(**args*, ***kwargs*)

Placeholder "sabe" method. This should be overridden by custom providers and save the file object to the provider.

Raises `NotImplementedError` – If the "save" method has not been implemented

`url_join`(**parts*)

Safe url part joining.

Parameters **`*parts`** (*list*) – List of parts to join together

Returns Joined url parts

Return type str

4.1.6 flask_store.providers.local

Local file storage for your Flask application.

Example

```
from flask import Flask, request
from flask.ext.store import Provider, Store
from wtforms import Form
from wtforms.fields import FileField

class FooForm(Form):
    foo = FileField('foo')

app = Flask(__app__)
app.config['STORE_PATH'] = '/some/file/path'

store = Store(app)

@app,route('/upload')
def upload():
    form = FooForm()
    form.validate_on_submit()

    if not form.errors:
        provider = store.Provider(request.files.get('foo'))
        provider.save()
```

class `flask_store.providers.local.`**`LocalProvider`**(*fp*, *location=None*)

The default provider for Flask-Store. Handles saving files onto the local file system.

static **`app_defaults`**(*app*)

Sets sensible application configuration settings for this provider.

Parameters **`app`** (*flask.app.Flask*) – Flask application at init

`exists`(*filename*)

Returns boolean of the provided filename exists at the compiled absolute path.

Parameters **`name`** (*str*) – Filename to check its existence

Returns Whether the file exists on the file system

Return type bool

`join`(**parts*)

Joins paths together in a safe manor.

Parameters **`*parts`** (*list*) – List of arbitrary paths to join together

Returns Joined paths

Return type str

open()
Opens the file and returns the file handler.

Returns Open file handler

Return type file

register_route = True
Ensure a route is registered for serving files

save()
Save the file on the local file system. Simply builds the paths and calls `werkzeug.datastructures.FileStorage.save()` on the file object.

4.1.7 flask_store.providers.s3

AWS Simple Storage Service file Store.

Example

```
from flask import Flask, request
from flask.ext.Store import Provider, Store
from wtforms import Form
from wtforms.fields import FileField

class FooForm(Form):
    foo = FileField('foo')

app = Flask(__app__)
app.config['STORE_PROVIDER'] = 'flask_store.providers.s3.S3Provider'
app.config['STORE_S3_ACCESS_KEY'] = 'foo'
app.confog['STORE_S3_SECRET_KEY'] = 'bar'

store = Store(app)

@app,route('/upload')
def upload():
    form = FooForm()
    form.validate_on_submit()

    provider = Provider(form.files.get('foo'))
    provider.save()
```

class `flask_store.providers.s3.`**S3GeventProvider**(**args*, ***kwargs*)
A Gevent Support for *S3Provider*. Calling *save()* here will spawn a greenlet which will handle the actual upload process.

save()
Acts as a proxy to the actual save method in the parent class. The save method will be called in a `greenlet` so `gevent` must be installed.

Since the origional request will close the file object we write the file to a temporary location on disk and create a new `werkzeug.datastructures.FileStorage` instance with the stram being the temporary file.

class `flask_store.providers.s3.`**S3Provider**(*fp*, *location=None*)
Amazon Simple Storage Service Store (S3). Allows files to be stored in an AWS S3 bucket.

REQUIRED_CONFIGURATION = ['STORE_S3_ACCESS_KEY', 'STORE_S3_SECRET_KEY', 'STORE_S3_BUCKET',
Required application configuration variables

static **app_defaults**(*app*)
Sets sensible application configuration settings for this provider.

Parameters **app** (`flask.app.Flask`) – Flask application at init

bucket(*s3connection*)
Returns an S3 bucket instance

connect()
Returns an S3 connection instance.

exists(*filename*)
Checks if the file already exists in the bucket using Boto.

Parameters **name** (`str`) – Filename to check its existence

Returns Whether the file exists on the file system

Return type bool

join(**parts*)
Joins paths into a url.

Parameters ***parts** (`list`) – List of arbitrary paths to join together

Returns S3 save joined paths

Return type str

open()
Opens an S3 key and returns an oepn File Like object pointer.

Returns In memory file data

Return type _io.BytesIO

save()
Takes the uploaded file and uploads it to S3.

Note: This is a blocking call and therefore will increase the time for your application to respond to the client and may cause request timeouts.

4.2 Change Log

4.2.1 0.0.4.3 - Alpha

- Bugfix: Python3 str error in setup

4.2.2 0.0.4.2 - Alpha

- Minor Feature: New `STORE_S3_ACL` optional setting. S3 Uploads will auto be set to `private` unless `STORE_S3_ACL` specifies a different ACL.

4.2.3 0.0.4.1 - Alpha

- Hotfix: Filename changed when saved is set on the provider instance

4.2.4 0.0.4 - Alpha

- Changed: Minor change to API, Provider now requires file instance or path

4.2.5 0.0.3.1 - Alpha

- Hotfix: Bug in FlaskStoreType where settings a `None` value would break the Provider, now checks the value is the expected instance type

4.2.6 0.0.3 - Alpha

- Feature: SQLAlchemy Store Type
- Changed: Renamed `stores` to `providers`
- Removed: Removed `FileStore` wrapper class - it was a bad idea.

4.2.7 0.0.2 - Alpha

- Feature: FileStore wrapper around provider files
- Bugfix: S3 url generation

4.2.8 0.0.1 - Alpha

- Feature: Local File Storage
- Feature: S3 File Storage
- Feature: S3 Gevented File Storage

4.3 Contributors

Without the work of these people or organisations this project would not be possible, we salute you.

- Soon London: http://thisissoon.com | @thisissoon
- Chris Reeves: @krak3n
- Greg Reed: @peeklondon
- Radek Los: @radeklos

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index