feather Documentation

Release 0.1.0

James Ramm

Jun 09, 2021

Contents

Feather is a library to help you create Falcon API's backed by MongoDB.

Using feather, you can make *schemas* which are backed by MongoDB. Feather also provides default Falcon resources which provide full CRUD functionality for your schemas.

Contents:

CHAPTER 1

Feather

A library to help you make Falcon web apps backed by MongoDB.

1.1 Features

- Simple interface to MongoDB using `marshmallow` schemas. This allows a single document definition which also provides serialization and validation
- Standard `Resource` classes for creating a full CRUD JSON API for REST collections and items.
- Easy filtering/projection of documents per request
- The `FileCollection` and `FileItem` resources provide file upload functionality. They can be configured to use feathers' basic `FileStore` or your own storage backend (e.g. GridFS)
- Useful extra fields for marshmallow (`Choice`, `Slug`, `MongoId`, `Password`...)

1.2 Example

The following example creates a basic JSON API for a representation of a user.

```
from datetime import datetime
from feather import create_app, schema, Collection, Item
from feather.connection import connect
from feather.fields import Slug
from marshmallow import fields, Schema
```

(continues on next page)

(continued from previous page)

```
class UserSchema(schema.MongoSchema):
    name = fields.Str(required=True)
    email = fields.Email(required=True)
    created = fields.DateTime(
            missing=lambda: datetime.utcnow().isoformat(),
            default=lambda: datetime.utcnow().isoformat()
    )
    profile = fields.Nested("ProfileSchema")
    slug = Slug(populate_from='name')

class ProfileSchema(Schema):
    """Example of nesting a schema.
    In mongodb, this will be a nested document
    """
    biography = fields.Str()
    profile_image = fields.Url(load_from='profileImage', dump_to='profileImage')


def get_app(database_name='myapp')
    """Creates the falcon app.
    We pass the database name so we can use a different db for testing
    """
    # Connect to the database *before* making schema instance.
    # The ``connect`` function takes the same arguments as pymongo's
    # ``MongoClient``. Here we connect to localhost.
    connect(database_name)
    user = UserSchema()
    resources = (Collection(user, '/users'), Item(user, '/users/{email}'))
    return create_app(resources)
```

Name this file `app.py` and run it with gunicorn:

gunicorn 'app:get_app()'

1.3 Design

Feather intends to be a light and transparent library. It should compliment and enhance Falcon & MongoDB usage but not get in the way of custom development. To this end I have a small number of rules:

- No magic. Like falcon itself, it should be easy to follow inputs to outputs. To this end we have a few soft rules such as:
  - Avoid mixins. Mixins introduce implicit dependencies and make it harder to reason about code.
  - Don't mess with metaclasses and double underscore methods without good reason. There is often an easier, clearer way to achieve the same result.
- No reinvention. We try to use well proven existing solutions before rolling our own. Hence the use of `marshmallow` for the ORM/serialization framework.
- No hijacking. Feather is complimentary or an 'add-on' to Falcon. It does not replace direct usage of Falcon (what you might expect from a framework). It solves some common use cases and provides some useful tools. When you want to do something unsupported and go direct to falcon, it doesnt get in your way.

CHAPTER 2

Installation

2.1 Stable release

To install feather, run this command in your terminal:

```
$ pip install feather
```

This is the preferred method to install feather, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for feather can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/JamesRamm/feather
```

Or download the tarball:

```
$ curl  -OL https://github.com/JamesRamm/feather/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

User Guide

Contents:

3.1 Schemas

Feather uses marshmallow `Schema`'s to define MongoDB documents. If you are not familiar with marshmallow, take a look at http://marshmallow.readthedocs.io/en/latest/index.html The schema integrates with pymongo in order to deliver data to and from the database. To keep this efficient, the pymongo integration is very light; it is worth being familiar with pymongo if you need to do more complex database logic.

A feather schema is defined exactly like a marshmallow schema except it inherits from `MongoSchema`. This provides a few new methods which will both materialize schema data to the database and get documents from the database.

Here is an example of defining a schema:

```
from feather import schema
from marshmallow import fields

class Person(schema.MongoSchema):

    name = fields.Str()
    email = fields.Str(required=True)
```

Like a regular marshmallow schema, you can call `dumps` and `loads` to serialize and deserialize data. You can do this safely with no impact on the database (just like marshamllow). In order to save/get data from the database, Feather provides new methods and resource classes to work directly with these methods.

Since the schema will be backed by MongoDB, we must connect before creating an instance:

```
from feather import connection

# Without arguments we connect to the default database on localhost
client = connect()
person = Person()
```

3.1.1 Database constraints

Feather schemas have support for creating constraints on the database. These are defined in the marshmallow `Meta` options class:

```
import simplejson
from feather import schema
from marshmallow import fields

class Person(schema.MongoSchema):

    class Meta:
        json_module = simplejson
        constraints = (('email', {'unique': True}), ('name', {}))

    name = fields.Str()
    email = fields.Str(required=True)
```

The constraints are specified as an iterable of 2-tuples, each comprising a 'key' and a dictionary of keyword arguments passed directly to pymongos' `create_index`. This requires you to know a little about how `create_index` works (https://api.mongodb.com/python/current/api/pymongo/collection.html#pymongo.collection.Collection.create_index) but has the advantage of being able to easily and transparently support all indexing possibilities.

3.1.2 Nested Documents & Relations

You can represent nested documents using marshmallows `Nested` field. The schema you intend to nest can just inherit directly from `Schema` since the parent schema will handle its' creation:

```
import simplejson
from feather import schema
from marshmallow import fields, Schema

class Person(schema.MongoSchema):

    class Meta:
        json_module = simplejson
        constraints = (('email', {'unique': True}), ('name', {}))

    name = fields.Str()
    email = fields.Str(required=True)
    profile = fields.Nested('Profile')


class Profile(Schema):

    biography = fields.Str()
    ...
```

3.1.3 Further Usage

- Feather supplies a small number of extra fields for use with your schemas, such as `Choice`, `Slug` and `MongoId`.
- If you wish to interact with the pymongo `collection` instance directly, you can call `get_collection` on any

class inheriting from `MongoSchema`. - By implementing the `get_filter` method on your schema class, you can provide per request

> filtering. Coupled with appropriate middleware, this can let you restrict/modify the queryset by user characteristics.

3.2 Resources

Default `Collection` and `Item` resources are provided to easily provide endpoints for your schemas. Each resource has the following features:

- Schema instances are passed into the resource for it to work on
- URI template is encapsulated in the resource
- Restricting the HTTP methods it will handle
- Changing the content types it will accept
- Custom error handlers for schema validation errors

A `Collection` resource by default provides POST and GET handlers, with GET returning a JSON list of the requested resource. An `Item`

Using the `Person` schema we created in the previous chapter, we can declare our resources:

```
from feather import Collection, Item

person = Person()

resources = (
    Collection(person, '/people'),
    Item(person, '/people/{name}')
)
```

With the resources ready, you can use a factory function to create a Falcon app:

```
from feather import create_app

# ``application`` is an instance of ``falcon.API``
application = create_app(resources)
```

All `create_app` does is instantiate an app and call Falcons' `add_route` for each resource in the given list.

3.2.1 File Storage

Feather also provides basic `FileCollection` and `FileItem` resource classes, specifically intended for serving and accepting file data. As with `Collection` and `Item` resources, you can configure the uri template, allowed content types and HTTP methods. You also expected to pass a storage class to the resource. This is essentially the same as in the Falcon tutorial.

The storage class should provide `save`, `open` and `list` methods. `save` and `open` are fairly clear and are as explained in the falcon tutorial. `list` should return the URL's of all available files in the store.

Feather provides a basic file store - `feather.FileStore` which can be used.

All this makes it easy to add file handling. Expanding the resources example:

```
import os
from feather import Collection, Item, FileCollection, FileItem, FileStore

# Setup the storage
path = os.path.dirname(__file__)
store = FileStore(path)

person = Person()

resources = (
    Collection(person, '/people'),
    Item(person, '/people/{name}'),
    FileCollection(store), # The uri_template argument defaults to ``/files``
    FileItem(store)
)
```

Handling files in schemas

If you come from django, you might be expecting some sort of `FileField` you can declare on a schema. Feather does not provide this; This keeps your file storage logic completely separate from the rest of the app, meaning you could potentially swap out your file store for a GridFS backed store, or switch to a completely different service for hosting files.

I reccomend that you declare files as Url fields on your schema, with the `relative=True` parameter set.

The other advantage over tighter coupling is that your file fields could simply be a URL to an entirely different website (e.g. some stock image provider, or a facebook profile picture).

There are disadvantages which we need to overcome:

- **You now need to make 2 requests from a client. One to upload the file and one to update the resource with the file url.**

  (It is a matter of some debate as to whether this should in fact be considered the best practice for REST API's since multipart form data is not truly JSON or XML)

- Feather offers no validation or method by which to link a file upload to a subsequent patch request other than what the client tells it. E.g. imagine a client successfully uploads the file but the patch to update the resource with the new URL goes wrong. To overcome this, you could take a look at 'Resumable Uploads'. We will be looking at whether Feather can provide any nice api to help with this in the future.

3.3 Recipes

3.3.1 Wrapping serialized results

By default, the output from serialization is simply a JSON object (if serializing a single model) or array (for many models). e.g.:

```
[
    {
        'name': 'John Cleese',
        'email': 'john.cleese@fake.com'
    },
```

(continues on next page)

(continued from previous page)

```
    {
        'name': 'Michael Palin',
        'email': 'micahel.palin@fake.com'
    }
]
```

However, we may wish to return a 'wrapped' response, e.g:

```
{
    'meta': {},
    'errors': [],
    'data': [
                {
                    'name': 'John Cleese',
                    'email': 'john.cleese@fake.com'
                },

                {
                    'name': 'Michael Palin',
                    'email': 'micahel.palin@fake.com'
                }
            ]
}
```

We can use marshmallows' `post_dump` decorator to achieve this in our schema:

```
class Person(MongoSchema):

    name = field.Str()
    email = field.Str()

    @post_dump(pass_many=True)
    def wrap_with_envelope(self, data, many):
        return {data: data, meta: {...}, errors: [...]}
```

3.3.2 Filtering output per user

We want to filter/modify the responses of GET requests depending on the connected user.

You can provide a `get_filter` method on the schema definition which accepts a falcon `Request` object and returns a dictionary of keyword arguments compatible with pymongos' `find` method:

```
class MySchema(MongoSchema):

    ...

    def get_filter(self, req):
        return {
            'filter': {<the desired filter params>},
            'projection': (<subset of fields to include in the returned documents>)
        }
```

In order to customise `get_filter` for each user, the `Request` object needs to have some useful information attached. This is where we would make use of Falcons' middleware in order to attach information about the user. For example, you could use falcon-auth to add the user to your request. A 'loader' function for `falcon-auth` (see the falcon-auth readme) might look something like:

We can now access `req.context['user']` in our `get_filter`:

```
class MySchema(MongoSchema):

    # Username of the 'owner' of this document
    owner = fields.Str()

    ...

    def get_filter(self, req):
        user = req.context['user']
        if user
        return {
            'filter': {'owner': user['username']},
        }
```

CHAPTER 4

API Reference

4.1 Schemas and fields

Inherit from `MongoSchema` to start creating schemas which are materialized to MongoDB. A `MongoSchema` is just a marshmallow schema with extra functions to give it ORM-like abilities.

Connect to MongoDB and provide a base schema which will save deserialized data to a collection

The connections to mongodb are cached. Inspired by MongoEngine

class `feather.schema.`**MongoSchema**(**args*, ***kwargs*)

A Marshmallow schema backed by MongoDB

When data is loaded (deserialized) it is saved to a mongodb document in a collection matching the Schema name (and containing app - similar to Django table names)

This enables marshmallow to behave as an ORM to MongoDB

`MongoSchema` does not override any marshmallow methods. Instead it provides new methods which are recognised by feathers 'Resource' classes. Therefore, the database will not be affected if you call `dump`/`dumps`

or `load`/`loads`

Note: Currently we attempt to create the database constraints when the schema is initialized. Therefore, you must connect to a database first.

OPTIONS_CLASS

alias of `MonogSchemaOpts`

count()

Wraps pymongo's *count* for this collection.

Returns the count of all documents in the collection

delete(*filter_spec*)

Delete an existing document

find(**args*, ***kwargs*)

Wraps pymongo's *find* for this collection

get(*filter_spec*, **args*, ***kwargs*)
Wraps pymongo's *find_one* for this collection

get_collection()
Return the pymongo collection associated with this schema.

get_filter(*req*)
Create a MongoDB filter query for this schema based on an incoming request. It is intended that this method be overridden in child classes to provide per-request filtering on `GET` requests.

Parameters **req** (*falcon.Request*) – processed

Returns

A dictionary containing keyword arguments which can be passed directly to pymongos' `find` method. defaults to an empty dictionary (no filters applied)

Return type dict

patch(*filter_spec*, *data*)
'Patch' (update) an existing document

Parameters

- **filter_spec** (*dict*) – The pymongo filter spec to match a single document to be updated
- **data** – JSON data to be validated, deserialized and used to update a document

post(*data*)
Creates a new document in the mongodb database.

Uses marshmallows' `loads` method to validate and complete incoming data, before saving it to the database.

Parameters **data** (*str*) – JSON data to be validated against the schema

Returns

Tuple of (data, errors) containing the validated & deserialized data dict and any errors.

Return type validated

put(*filter_spec*, *data*)
'Put' (replace) an existing document

See documentation for `MongoSchema.patch`

Some useful fields for using marshmallow as a MongoDB ORM are also provided.

class feather.fields.**Choice**(*choices=None*, **args*, ***kwargs*)
The input value is validated against a set of choices passed in the field definition. Upon serialization, the full choice list along with the chosen value is returned (in a dict). Only the chosen value should be passed in deserialization.

class feather.fields.**MongoId**(*default=<marshmallow.missing>*, *attribute=None*, *load_from=None*, *dump_to=None*, *error=None*, *validate=None*, *required=False*, *allow_none=None*, *load_only=False*, *dump_only=False*, *missing=<marshmallow.missing>*, *error_messages=None*, ***metadata*)
Represents a MongoDB object id

Serializes the ObjectID to a string and deserializes to an ObjectID

class feather.fields.**Slug**(*populate_from=None*, **args*, ***kwargs*)
Represents a slug. Massages the input value to a lowercase string without spaces.

4.2 Resources and hooks

Resource classes for creating a JSON restful API.

class feather.resource.**Collection**(*schema*, *uri_template*, *content_types='application/json'*, *methods=('get', 'post')*, *error_handler=<function basic_error_handler>*)

Generic class for listing/creating data via a schema

Remembering that the @ operator is just syntactic sugar, if we want to apply a decorator we could do it with minimal effort like this:

resource = Collection(...) resource.on_post = falcon.before(my_function)(resource.on_post)

Alternatively, we could create a subclass:

class MyResource(Collection): on_post = falcon.before(my_function)(Collection.on_post.__func__)

Also note that when overriding, you will need to manually add back the content type validation for the `_post` method if appropriate.

Parameters

- **schema** (feather.schema.MongoSchema) – An instance of a `MongoSchema` child class on which the `Collection` instance should operate.
- **uri_template** (*str*) – See `feather.resource.FeatherResource`
- **content_types** (*tuple or list*) – See `feather.resource.FeatherResource`. Defaults to `'application/json'`
- **methods** (*str*) – See `feather.resource.FeatherResource`. Defaults to `('get', 'post')`
- **error_handler** (*callable*) – See `feather.resource.FeatherResource`.

class feather.resource.**FeatherResource**(*uri_template*, *content_types={'application/json'}*, *methods=('get', 'patch', 'put', 'delete', 'post')*, *error_handler=<function basic_error_handler>*)

Base class used for setting a uri_template, allowed content types and HTTP methods provided.

By encapsulating the URI, we can provide factory methods for routing, allowing us to specify the resource and its' uri in one place

HTTP handler methods (`on_<method>` in falcon) are dynamically assigned in order to allow Resource instances to be created for with different sets of requiremets. (E.g. create a read-only collection by only passing `('get',)` when instantiating). This explains why the method handlers below are not named `on_<method>` but simple `_<method>`.

Allowed content types are passed for the same reason. A sub class could check these using the `validated_content_type` hooks. This is mostly useful for file uploads (see `FileCollection` or `FileItem`) where you might wish to restrict content types (e.g. images only)

Parameters

- **uri_template** (*str*) – A URI template for this resource which will be used when routing (using the `feather.create_app` factory function) and for setting `Location` headers.
- **content_types** (*tuple, set or list*) – List of allowed content_types. This is not used by default. Instead, decorate desired handler methods with @falcon.before(validate_content_type). A `set` is reccomended as the validation performs an exclusion (`not in`) operation

- **methods** (*tuple or list*) – List of HTTP methods to allow.
- **error_handler** (*callable*) – A function which is responsible for handling validation errors returned by a marshmallow schema. Defaults to `feather.resource.basic_error_handler`

uri_template
The URI template for this resource

class feather.resource.**FileCollection**(*store*, *uri_template='/files'*, *content_types=None*, *methods=('get', 'post')*)
Collection for posting/listing file uploads.

By default, all content types are allowed - usually you would want to limit this, e.g. just allow images by passing `('image/png', 'image/jpeg')`

class feather.resource.**FileItem**(*store*, *uri_template='/files/{name}'*, *content_types=None*, *methods=('get',)*)
Item resource for interacting with single files

class feather.resource.**Item**(*schema*, *uri_template*, *content_types='application/json'*, *methods=('get', 'patch', 'put', 'delete')*)
Generic class for getting/editing a single data item via a schema

Parameters

- **schema** (`feather.schema.MongoSchema`) – An instance of a `MongoSchema` child class on which the `Item` instance should operate.
- **uri_template** (*str*) – See `feather.resource.FeatherResource`
- **content_types** (*tuple or list*) – See `feather.resource.FeatherResource`. Defaults to `'application/json'`
- **methods** (*str*) – See `feather.resource.FeatherResource`. Defaults to `('get', 'put', 'patch', 'delete')`
- **error_handler** (*callable*) – See `feather.resource.FeatherResource`.

feather.resource.**basic_error_handler**(*error_dict*)
Handle an error dictionary returned by a marshmallow schema.

This basic handler either returns a 409 conflict error if the error dictionary indicates a duplicate key, or a 400 bad request error with the error dictionary attached.

Hooks for working with a `FeatherResource`

feather.hooks.**validate_content_type**(*req*, *resp*, *resource*, *params*)
Validate the content type of a `FeatherResource`

4.3 Storage

feather.storage.**unique_id**()
Simplistic unique ID generation. The returned ID is just the current timestamp (in ms) converted to hex

4.4 Connecting to a database

Connect to MongoDB and provide a base schema which will save deserialized data to a collection

The connections to mongodb are cached. Inspired by MongoEngine

feather.connection.**connect**(*database='default'*, *alias='default'*, ***kwargs*)
Connect to a database or retrieve an existing connection

feather.connection.**disconnect**(*alias='default'*)
Close the connection with a given alias.

feather.connection.**get_database**(*alias='default'*)
Get an existing database

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/JamesRamm/feather/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" is open to whoever wants to implement it.

5.1.4 Write Documentation

feather could always use more documentation, whether as part of the official feather docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/JamesRamm/feather/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *feather* for local development.

1. Fork the *feather* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/feather.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv feather
$ cd feather/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 feather tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/JamesRamm/feather/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ py.test tests.test_feather
```

CHAPTER 6

Credits

6.1 Development Lead

- James Ramm <jamessramm@gmail.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

7.1 0.1.0 (2017-09-25)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index