

---

# **plume Documentation**

***Release 0.1.0***

**James Ramm**

**Apr 08, 2018**



---

## Contents

---

<b>1</b>	<b>Plume</b>	<b>3</b>
1.1	Features . . . . .	3
1.2	Example . . . . .	3
1.3	Design . . . . .	4
<b>2</b>	<b>Installation</b>	<b>5</b>
2.1	Stable release . . . . .	5
2.2	From sources . . . . .	5
<b>3</b>	<b>User Guide</b>	<b>7</b>
3.1	Schemas . . . . .	7
3.2	Resources . . . . .	9
3.3	Recipes . . . . .	11
<b>4</b>	<b>API Reference</b>	<b>13</b>
4.1	Schemas and fields . . . . .	13
4.2	Resources and hooks . . . . .	15
4.3	Storage . . . . .	17
4.4	Connecting to a database . . . . .	17
<b>5</b>	<b>Contributing</b>	<b>19</b>
5.1	Types of Contributions . . . . .	19
5.2	Get Started! . . . . .	20
5.3	Pull Request Guidelines . . . . .	21
5.4	Tips . . . . .	21
<b>6</b>	<b>Credits</b>	<b>23</b>
6.1	Development Lead . . . . .	23
6.2	Contributors . . . . .	23
<b>7</b>	<b>History</b>	<b>25</b>
7.1	0.1.0 (2017-09-25) . . . . .	25
<b>8</b>	<b>Indices and tables</b>	<b>27</b>
	<b>Python Module Index</b>	<b>29</b>



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

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

Contents:



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
- Easy filtering/projection of documents per request
- Useful extra fields for Schemas (Choice, Slug, MongoId, Password...)
- Standard `Resource` classes for creating a full CRUD JSON API for REST collections and items.
- Resource classes for generic file uploads. They can be configured to use different storage backends, and validate different content types

## 1.2 Example

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

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

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

Plume 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. Plume 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 doesn't get in your way.



### 2.1 Stable release

To install plume, run this command in your terminal:

```
$ pip install plume
```

This is the preferred method to install plume, 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 plume can be downloaded from the [Github repo](#).

You can either clone the public repository:

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

Or download the [tarball](#):

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

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

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



Contents:

### 3.1 Schemas

Plume builds upon marshmallow's `Schema` class both for serialization and as a lightweight 'ODM' (Object-Document Mapper) to MongoDB. If you are not familiar with marshmallow, take a look at the [documentation](#).

A plume 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.

`MongoSchema` uses `pymongo` in order to deliver data to and from the database. To keep this efficient, the `pymongo` integration is simple and streamlined, with `pymongo` objects being readily accessible from various `MongoSchema` methods. As such it is worth being familiar with `pymongo` if you need to do more complex database logic.

Here is an example of defining a schema:

```
from plume 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 marshmallow). In order to save/get data from the database, Plume 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 plume import connection

# Without arguments we connect to the default database on localhost
```

```
client = connect()
person = Person()
```

### 3.1.1 Database constraints

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

```
import simplejson
from plume 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 pymongo’s `create_index`. This requires you to know a little about how `create_index` works 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 plume 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()
    ...
```

MongoDB does not support foreign keys, nor does pymongo create abstractions for relations such as many to many, one to one etc. In my view, this is a good thing. Handling relations at the app level allows for a more app-specific implementation. It allows us to:

- Keep apps completely decoupled at the database level, making it easier to drop certain schemas for different situations. E.g. if we were to split our app into microservices, switch to using a 3rd party service for some part

of the app or simply refactor into a different structure

- We generally know more about what is and isn't acceptable to be missing/broken at an app level. I have often had to deal with painful migrations due to complex relationships between apps, when the front-end could actually be easily modified to be agnostic as to whether some data appears or not.
- We can choose which field of the related data to use for our relation. For example, imagine we are representing a client-supplier relationship. The supplier has a list of clients they deal with. We may decide as a first iteration that we only wish to simply let the supplier see a list of client emails. We can simply use a `List` field and embed the emails right in the supplier schema. We have represented the relationship but we don't need any complex joins and lookups to fetch the data. At a later date we may wish to give the client a full profile. We can simply keep the suppliers list of emails and create a new schema representing a client profile, with an index on the email field. The front-end can now - either work as normal (just showing a list of emails), or make a second call to fetch the client profile for each email. You might then decide there is little business value in having the client profile, so lets try dropping it for a month and get some feedback. All this kind of stuff is much easier to do when you handle relations at app level rather than database level.

In short, define your relations and rules within your app.

### 3.1.3 Further Usage

- Plume 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 plume 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 plume 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

Plume 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.

Plume provides a basic file store - `plume.FileStore` which can be used.

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

```
import os
from plume 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. Plume 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 recommend 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)

- Plume 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 Plume 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'
  },
  {
    '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 pymongoos' `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']},  
            }
```



## 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 plume.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 plumes '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 pymongo's *find* method. defaults to an empty dictionary (no filters applied)

**Return type** dict

**patch** (*filter\_spec*, *data*, *jsonpatch=False*)

'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. By default, JSON data is expected to be expressed using MongoDB update operators (<https://docs.mongodb.com/manual/reference/operator/update/>) By passing *jsonpatch=True* data can be formatted according to the JSONPatch specification (<http://jsonpatch.com/>). This support is experimental
- **jsonpatch** (*boolean*, *False*) – Enable experimental support for jsonpatch. In this case, *data* should follow the jsonpatch format

**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** `plume.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 plume.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 plume.fields.Password (password_checker=<class 'passlib.handlers.sha2_crypt.sha256_crypt'>,
                              *args, **kwargs)
```

Password field

```
class plume.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 plume.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** (`plume.schema.MongoSchema`) – An instance of a `MongoSchema` child class on which the `Collection` instance should operate.
- **uri\_template** (`str`) – See `plume.resource.PlumeResource`
- **content\_types** (`tuple` or `list`) – See `plume.resource.PlumeResource`. Defaults to `'application/json'`
- **methods** (`str`) – See `plume.resource.PlumeResource`. Defaults to `('get', 'post')`
- **error\_handler** (`callable`) – See `plume.resource.PlumeResource`.

```
class plume.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 plume.resource.FileItem (store, uri_template='/files/{name}', content_types=None, methods=
                                ('get', ))
```

Item resource for interacting with single files

```
class plume.resource.Item(schema, uri_template, content_types='application/json', meth-
                        ods=('get', 'patch', 'put', 'delete'), error_handler=<function ba-
                        sic_error_handler>, use_jsonpatch=False)
```

Generic class for getting/editing a single data item via a schema

#### Parameters

- **schema** (`plume.schema.MongoSchema`) – An instance of a MongoSchema child class on which the Item instance should operate.
- **uri\_template** (*str*) – See `plume.resource.PlumeResource`
- **content\_types** (*tuple or list*) – See `plume.resource.PlumeResource`. Defaults to 'application/json'
- **methods** (*str*) – See `plume.resource.PlumeResource`. Defaults to ('get', 'put', 'patch', 'delete')
- **error\_handler** (*callable*) – See `plume.resource.PlumeResource`.

```
class plume.resource.PlumeResource(uri_template, content_types=('application/json', ),
                                methods=('get', 'patch', 'put', 'delete', 'post'),
                                error_handler=<function basic_error_handler>,
                                use_jsonpatch=False, schema=None)
```

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 requirements. (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 `plume.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 recommended 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 `plume.resource.basic_error_handler`

#### **uri\_template**

The URI template for this resource

```
plume.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 `PlumeResource`

`plume.hooks.validate_content_type(req, resp, resource, params)`  
Validate the content type of a `PlumeResource`

## 4.3 Storage

`plume.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`

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

`plume.connection.disconnect(alias='default')`  
Close the connection with a given alias.

`plume.connection.get_database(alias='default')`  
Get an existing database



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/plume/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

plume could always use more documentation, whether as part of the official plume 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/plume/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 *plume* for local development.

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

```
$ git clone git@github.com:your_name_here/plume.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 plume
$ cd plume/
$ 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 plume 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/plume/pull\\_requests](https://travis-ci.org/JamesRamm/plume/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_plume
```



## CHAPTER 6

---

### Credits

---

#### 6.1 Development Lead

- James Ramm <jamessramm@gmail.com>

#### 6.2 Contributors

None yet. Why not be the first?



#### 7.1 0.1.0 (2017-09-25)

- First release on PyPI.



## CHAPTER 8

---

### Indices and tables

---

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





### p

- `plume.connection`, [17](#)
- `plume.fields`, [14](#)
- `plume.hooks`, [16](#)
- `plume.resource`, [15](#)
- `plume.schema`, [13](#)
- `plume.storage`, [17](#)



## B

`basic_error_handler()` (in module `plume.resource`), 16

## C

`Choice` (class in `plume.fields`), 14

`Collection` (class in `plume.resource`), 15

`connect()` (in module `plume.connection`), 17

`count()` (`plume.schema.MongoSchema` method), 13

## D

`delete()` (`plume.schema.MongoSchema` method), 13

`disconnect()` (in module `plume.connection`), 17

## F

`FileCollection` (class in `plume.resource`), 15

`FileItem` (class in `plume.resource`), 15

`find()` (`plume.schema.MongoSchema` method), 13

## G

`get()` (`plume.schema.MongoSchema` method), 13

`get_collection()` (`plume.schema.MongoSchema` method), 14

`get_database()` (in module `plume.connection`), 17

`get_filter()` (`plume.schema.MongoSchema` method), 14

## I

`Item` (class in `plume.resource`), 15

## M

`MongoId` (class in `plume.fields`), 14

`MongoSchema` (class in `plume.schema`), 13

## O

`OPTIONS_CLASS` (`plume.schema.MongoSchema` attribute), 13

## P

`Password` (class in `plume.fields`), 15

`patch()` (`plume.schema.MongoSchema` method), 14

`plume.connection` (module), 17

`plume.fields` (module), 14

`plume.hooks` (module), 16

`plume.resource` (module), 15

`plume.schema` (module), 13

`plume.storage` (module), 17

`PlumeResource` (class in `plume.resource`), 16

`post()` (`plume.schema.MongoSchema` method), 14

`put()` (`plume.schema.MongoSchema` method), 14

## S

`Slug` (class in `plume.fields`), 15

## U

`unique_id()` (in module `plume.storage`), 17

`uri_template` (`plume.resource.PlumeResource` attribute), 16

## V

`validate_content_type()` (in module `plume.hooks`), 16