Flask Slither Documentation

Release 0.3

Nico Gevers

Sep 27, 2017

Contents

Slither is an API 'framework' for Flask which interfaces with MongoDB. It allows for rapid API development as well as customisations needed for bigger projects.

CHAPTER 1

Getting Started with Slither

Flask-Slither allows you to sublass a *resource* which provides the entry point for all http methods (GET, POST, PUT, PATCH, DELETE, OPTION). A resource and URI are fairly synonymous. Ie goign to the url */foo* will provide the Foo object while goign to the url */bar* gives you access to the bar object. These need map to a MongoDB collection by default (foos and bars respectively)

Note: Currently there is no mailing list for slither, so please create an issue on slither's github repo to get help.

A good knowledge of Flask is useful, but not necessary. At least a basic knowledge of Flask is beneficial, even though this tutorial will cater for the absolute beginner.

This tutorial will walk you through the creating of a simple library application. By the end of this tutorial you will know how to CRUD books in the library as well lend out and return books.

Installation

Make sure you have at least Python 2.5+ installed, along with virtualenv.

It is always a good idea to run any project in a virtualenv. First setup the virtualenv for your project then load the needed dependencies.:

```
$ virtualenv --no-site-packages library
$ cd library
~/library $ source ./bin/activate
~/library $ pip install Flask-Slither python-dateutil pytz blinker
```

All dependencies needed for Flask-Slither will be installed, so there is no need to explicitly install Flask first. The output will reveal that Flask is, in fact, installed as well.

Note: The dateutil, pytz and blinker libraries a are used by some of the authentication module. If you're running a lean system and don't want the overhead, you can override the modules that need the libraries and remove them from

your virtualenv.

To make sure that your install was successful open a python prompt and type the following:

```
>>> from flask.ext.slither import register_api
>>>
```

Creating the App

We'll be creating a very simple Flask app. First we create a directory for our application and then edit the *__init__.py* file using our favourite editor:

```
$ mkdir app
$ cd app
$ vi __init__.py
```

Next we create a basic Flask application (in *app/__init__.py*).

```
# -*- coding: utf-8 -*-
from flask import Flask
from pymongo import MongoClient
from flask.ext.slither.resources import BaseResource
from flask.ext.slither import register_api
from werkzeug.routing import BaseConverter

app = Flask(__name__)


class RegexConverter(BaseConverter):
    def __init__(self, url_map, *items):
        super(RegexConverter, self).__init__(url_map)
        self.regex = items[0]

app.url_map.converters['regex'] = RegexConverter

# setup the connection to our mongo database
app.config['DB_HOST'] = 'localhost'
app.config['DB_PORT'] = 27017
app.config['DB_NAME'] = 'library'
client = MongoClient(app.config['DB_HOST'], app.config['DB_PORT'])
app.db = client[app.config['DB_NAME']]

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

To check that we're on the right track, run the application by issuing the following command:

```
~/library $ python __init__.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

Note: If the server doesn't come up, ensure that you have mongodb installed and that it is up and running.

Note: One caveat currently is installing the *RegexConverter*. This isn't strictly needed but is a useful addition. It allows us to reference a resource not just by its unique mongo id, but also by a field. By default it is the name field, but can be set per resource.

Interacting with the API (Part I)

Now that we know our setup is good, lets create the resources. We want our API to support the following functions:

- Create a book with a name, ISBN number and quantity available
- Edit the details of a book
- Delete a book
- Get a list of all the books
- Check a book out of the library
- Return a book

These functions can be split into two logical sections. The first four items will be covered by the *book* resource, and the last two by the *lending* resource. The first resource maps explicitly to the MonboDB books collection. For simplicity's sake, we'll map the lending resource to the books collection as well, so that we can easily manipulate the data. In real life, we'd probably want to track who has books, but for now we're keeping it simple. To start with, lets create our two resources (in *__init__.py*).

```
...
app.db = client[app.config['DB_NAME']]


class BookResource(BaseResource):
    collection = 'books'


class LendingResource(BaseResource):
    collection = 'books'


register_api(app, BookResource)
register_api(app, LendingResource)

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

As you can see the definition is pretty simple. Firstly we subclass Slither's BaseResource, and then we register the endpoints for the resource. As you probably noticed, except for the endpoint name, accessing both these resources will yield the same result. That's because they reference the same MongoDB collection. We'll change the *LendingResource* later.

Lets test this out. Start up your server and run the following cURL request from the command line.:

```
$ curl http://127.0.0.1:5000/books
{"books": []}
```

Ah, its working. But we have no books in the library just yet. Lets add one:

```
$ curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"books": {
↪"name": "Python Cookbook, 3rd Edition", "quantity": 8, "ISBN":"978-1449340377"}}'␣
↪http://127.0.0.1:5000/books
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 0
Cache-Control: max-age=30,must-revalidate
Access-Control-Allow-Origin: *
Location: http://127.0.0.1:5000/books/51a8feb6421aa965ffaf1435
Expires: Fri, 31 May 2013 19:51:30 GMT
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Fri, 31 May 2013 19:51:00 GMT
```

You'll see from the responses that each of the books was created successfully. Notice that the header also returned the URI of the book. We should be able to access that book from the link:

```
$ curl http://127.0.0.1:5000/books/51a8feb6421aa965ffaf1435
{"books": {"_id": {"$oid": "51a8feb6421aa965ffaf1435"}, "ISBN": "978-1449340377",
↪"name": "Python Cookbook, 3rd Edition", "quantity": 8}

$ curl http://127.0.0.1:5000/books
{"books": [{"_id": {"$oid": "51a8feb6421aa965ffaf1435"}, "ISBN": "978-1449340377",
↪"name": "Python Cookbook, 3rd Edition", "quantity": 8}]
```

Note: The actual location of the book will differ on your setup, so copying of the cURL command verbatim will not work. Rather copy it from the location header.

Great. We've managed to easly create a book and see if its there. Now we've decided we'd rather have the edition in a separate field. Lets update the book as follows:

```
$ curl -H "Content-Type: application/json" -X PATCH --data '{"books": {"name":
↪"Python Cookbook", "edition": "3rd"}}' http://127.0.0.1:5000/books/
↪51a8feb6421aa965ffaf1435
{"books": {"edition": "3rd", "_id": {"$oid": "51a8feb6421aa965ffaf1435"}, "ISBN":
↪"978-1449340377", "name": "Python Cookbook", "quantity": 8}}
```

We've easily been able to update the record with a PATCH command. Here we're leveraging the power of MongoDB in adding new fields at will. We could have decided to use the PUT request, instead of PATCH. They work in much the same way, except that PUT requires that all fields be passed to the server, while PATCH only requires changed fields.

Lets create a few more books for our library:

```
$ curl -H "Content-Type: application/json" -X POST --data '{"books": {"name": "The␣
↪Quick Python Book", "edition": "2nd", "quantity": 12, "ISBN":"978-1935182207"}}'␣
↪http://127.0.0.1:5000/books
$ curl -H "Content-Type: application/json" -X POST --data '{"books": {"name": "Python␣
↪for Kids", "edition": "3rd", "quantity": 1}}' http://127.0.0.1:5000/books
$ curl -H "Content-Type: application/json" -X POST --data '{"books": {"name": "Invent␣
↪Your Own Computer Games with Python", "edition": "2nd", "quantity": 2, "ISBN": "978-
↪0982106013"}}' http://127.0.0.1:5000/books
```

We now have a total of four different books in the library. Unfortunately there has been a change in management, and the new boss tells us to remove all books without an ISBN number. We decide to run a query to find all books without an ISBN number and then delete each one manually:

```
$ curl -g 'http://127.0.0.1:5000/books?where={"ISBN":{"$exists":false}}'
{"books": [{"edition": "3rd", "quantity": 1, "id": "51aa38b5421aa90e83a40e0b", "name
↪": "Python for Kids"}]}
```

Only one book found, and that's the book "Python for Kids" To delete it we run the following command:

```
$ curl -X DELETE http://127.0.0.1:5000/books/51aa38b5421aa90e83a40e0b
$ curl -g 'http://127.0.0.1:5000/books?where={"ISBN":{"$exists":false}}'
{"books": []}
```

Interacting with the API (Part II)

In the first part we set up the endpoints and tested that the resource responded correctly to the API calls. In this part we'll focus on the *lending* resource and manipulate the quantity of books through that api endpoint.

CHAPTER 2

Requirements

The following are base requirements for Slither to function out the box:

Required

- Python 2.5+
- MongoDB
- Flask 0.9
- pymongo 2.5

Optional

- mongokit (For form validations)
- Flask-Testing (to run the tests)
- nose (to execute the tests)
- pytz, python-dateutil, blinker (for signed request authentication)

CHAPTER 3

Why Slither?

Building a RESTful application is nothing new. When using mongo as a backend, the payload is in JSON, and if the API produces (mostly) JSON, then only a thin layer of logic is needed between the two. Here are some features:

- Uses Flask's `MethodView` as a basis for API endpoints
- Its fast because it uses pymongo directly with little overhead
- Quick and easy setup
- Optional validation avaliable using MongoKit's validation engine

Slither is based on Django's Tastypie, and uses constructs that will be familiar tho developers who've used tastypie before.

CHAPTER 4

Testing Slither

The easiest way to run the unit tests, is to install nose and Flask-Testing. Simply navigate to the root of the flask_slither code and run `nosetests` and the tests will run. Make sure you have a mongo instance up and running otherwise the tests will fail.