Durga Documentation

Release 0.2.0.dev2

transcode

June 30, 2015

Contents

Durga is a Python library to consume REST resources with optional schema validation.

It is named after the powerful and wise Hindu goddess Durga.

> "Durga the mahashakti, the form and formless, is the root cause of creation, preservation and annihilation. According to legend, Durga was created for the slaying of the buffalo demon Mahisasura by Brahma, Vishnu, Shiva, and the lesser gods, who were otherwise powerless to overcome him."
>
> —Wikipedia

CHAPTER 1

Features

- Lightweight REST client
- Easy to use API concept consisting of *Resource*, *Collection*, *Element* that is inspired by Django's ORM
- A Resource can be described using a schema which is used to validate the JSON data returned by a REST API
- Works with Python 2.7, 3.4 and PyPy

CHAPTER 2

Contents

2.1 Installation

Use pip to install Durga:

```
$ pip install durga
```

If you want to install the latest development version use **pip**'s `--pre` option:

```
$ pip install --pre durga
```

2.2 Usage

2.2.1 Flickr example

To use Durga in a project define a `class` that extends *durga.Resource*. This example uses the Flickr API flickr.photos.search with Python 3:

```
import durga


class FlickrResource(durga.Resource):
    base_url = 'https://api.flickr.com/services'
    path = 'rest'
    objects_path = ('photos', 'photo')
    schema = durga.schema.Schema({
        'farm': durga.schema.Use(int, error='Invalid farm'),
        'id': durga.schema.Use(int, error='Invalid id'),
        'isfamily': durga.schema.Use(bool, error='Invalid isfamily'),
        'isfriend': durga.schema.Use(bool, error='Invalid isfriend'),
        'ispublic': durga.schema.Use(bool, error='Invalid ispublic'),
        'owner': durga.schema.And(str, len, error='Invalid owner'),
        'secret': durga.schema.And(str, len, error='Invalid secret'),
        'server': durga.schema.Use(int, error='Invalid server'),
        'title': durga.schema.And(str, len, error='Invalid title'),
    })
    query = {
        'method': 'flickr.photos.search',
        'api_key': 'a33076a7ae214c0d12931ae8e38e846d',
        'format': 'json',
```

```
        'nojsoncallback': 1,
    }
```

Note: For convenience *durga.Resource*. and the schema library are available at the top module level.

Now you can search for the first 10 cat images:

```
cats = FlickrResource().collection.filter(text='Cat', per_page=10)
```

This will return a *durga.Collection* with a *durga.Element* for each result.

2.2.2 MusicBrainz example

Musicbrainz is an open music encyclopedia that collects music metadata and makes it available to the public. With Artist you get detailed entry for a single artist.

```
from uuid import UUID

from dateutil import parser

import durga


class MusicBrainzResource(durga.Resource):
    base_url = 'http://musicbrainz.org/ws/2'
    id_attribute = 'id'
    path = 'artist'
    schema = durga.schema.Schema({
        'country': durga.schema.And(str, len, error='Invalid country'),
        'ipis': [durga.schema.Optional(str)],
        'area': {
            'disambiguation': durga.schema.Optional(str),
            'iso_3166_3_codes': [durga.schema.Optional(str)],
            'sort-name': str,
            'name': str,
            'id': durga.schema.And(str, len, lambda n: UUID(n, version=4)),
            'iso_3166_2_codes': [durga.schema.Optional(str)],
            'iso_3166_1_codes': [durga.schema.Optional(str)],
        },
        'sort-name': str,
        'name': str,
        'disambiguation': durga.schema.Optional(str),
        'life-span': {
            'ended': bool,
            'begin': durga.schema.And(str, len, parser.parse, error='Invalid begin'),
            'end': durga.schema.Or(
                None,
                durga.schema.And(str, len, parser.parse, error='Invalid end')),
        },
        'end_area': durga.schema.Or(None, {
            'disambiguation': durga.schema.Optional(str),
            'iso_3166_3_codes': [durga.schema.Optional(str)],
            'sort-name': str,
            'name': str,
            'id': durga.schema.And(str, len, lambda n: UUID(n, version=4)),
            'iso_3166_2_codes': [durga.schema.Optional(str)],
            'iso_3166_1_codes': [durga.schema.Optional(str)],
```

```
        }),
        'id': durga.schema.And(str, len, lambda n: UUID(n, version=4), error='Invalid id'),
        'type': str,
        'begin_area': {
            'disambiguation': durga.schema.Optional(str),
            'iso_3166_3_codes': [durga.schema.Optional(str)],
            'sort-name': str,
            'name': str,
            'id': durga.schema.And(str, len, lambda n: UUID(n, version=4)),
            'iso_3166_2_codes': [durga.schema.Optional(str)],
            'iso_3166_1_codes': [durga.schema.Optional(str)],
        },
        'gender': durga.schema.Or(None, str),
    })
    query = {
        'method': '',
        'fmt': 'json',
        'nojsoncallback': 1,
    }
```

Note: In the example above you can see a more complex usage of validation.

For example to validate UUIDs:

```
'id': durga.schema.And(str, len, lambda n: UUID(n, version=4)),
```

For example to validate date:

```
'begin': durga.schema.And(str, len, parser.parse, error='Invalid begin'),
```

Now let's use the MusicBrainzResource:

```
MusicBrainzResource().collection.get(id='05cbaf37-6dc2-4f71-a0ce-d633447d90c3').name
```

That returns name of artist with given id:

```
''
```

2.3 How to define a resource

2.3.1 base_url *(required)*

Each REST API has a fixed main URL behind that contains all other resources. You can find this value for your application of durga in the used API documentation.

Examples

```
base_url = 'https://api.flickr.com/services'
```

```
base_url = 'http://musicbrainz.org/ws/2'
```

2.3.2 path *(required)*

Defines path of API resource you would like to use. You can find it in your API method description.

2.3.3 path_params

With setting *path_params* you lists your all your placeholder in *path*.

Example

```
path = 'movies/{movie_name}/{movie_year}/actors'
path_params = ('movie_name', 'movie_year')
```

2.3.4 id_attribute

Attribute *url* is taken per default to has a complete unique resource url. If you would like to change this, you can define a *id_attribute* by your own. And set your defined attribute manually.

Default

```
url = 'https://api.example.com/movies/23'
```

Changed id_attribute

```
id_attribute = 'id'
id = '23'
```

2.3.5 object_path

Indicates the sub path behind the *base_url*.

2.3.6 objects_path

It is used if a single resource is returned where the data is somewhere deeper inside the response. Often your response contains meta information next to your necesssary objects. So you have show the resource the path to that data.

Example

If your JSON response looks like kind of this

```
{
    "meta": {
        "limit": 20,
        "next": null,
        "offset": 0,
        "previous": null,
        "total_count": 4
```

```
    },
    "objects": [
        {
            "id": 1,
            "resource_uri": "https://api.example.com/movies/1",
            "runtime": 154,
            "title": "Pulp Fiction",
            "director": "Quentin Tarantino",
            "year": 1994
        }
    ]
}
```

Then your attribute *objects_path* is defined in this way

```
objects_path = ('objects',)
```

2.3.7 query

Describes all your query specific params like *format, method* or *params* and so on.

Example

```
query = {
    'method': 'flickr.photos.search',
    'api_key': 'a33076a7ae214c0d12931ae8e38e846d',
    'format': 'json',
    'nojsoncallback': 1,
}
```

2.3.8 schema

Is a data type representation of your API response which is necessary if you would like to validate your incoming data. Look at schema documentation and examples in Usage to define your own schema.

2.4 durga

2.4.1 durga package

Submodules

durga.collection module

class durga.collection.**Collection**(*url*, *resource*)

Bases: object

all()

count()

create(*data*)
Create a new remote resource from a dictionary.

At first the data will be validated. After successful validation the data is converted to JSON. The response of the POST request is returned afterwards.

delete()
Delete all Elements of this Collection.

Return the response for each deleted Element as a list.

elements

filter(***kwargs*)

get(***kwargs*)

get_element(*data*)
Return an Element instance holding the passed data dictionary.

get_element_url(*id*)

get_values(*data*)
Return either a dictionary, a tuple or a single field.

The data type and the fields returned are defined by using values() or values_list().

order_by()

response = None

update(*data*)
Update all Elements of this Collection with data from a dictionary.

The data dictionary is used to update the data of all Elements of this Collection. The updated Elements are validated and their data is converted to JSON. A PUT request is made for each Element. Finally a list of all responses is returned.

values(**fields*)
Return a list of dictionaries instead of Element instances.

The optional positional arguments, *fields, can be used to limit the fields that are returned.

values_list(**fields*, ***kwargs*)
Return a list of tuples instead of Element instances.

The optional positional arguments, *fields, can be used to limit the fields that are returned.

If only a single field is passed in, the flat parameter can be passed in too. If True, this will mean the returned results are single values, rather than one-tuples.

durga.element module

class durga.element.**Element**(*resource*, *data*)
Bases: object

delete()

get_data()
Return the Element's data as dictionary.

get_raw()

get_resource()

get_url()

save()
Update the remote resource.

There are two ways to provide data to be saved:

1. Pass it as a dictionary to the `update()` method.

2. Modify the Element's attributes.

The data will be validated before the PUT request is made. After a successful update an updated Element instance is returned.

update(*data*)
Update the attributes with items from the data dictionary.

validate()
Validate the Element's data.

If validation fails a schema.SchemaError is raised.

durga.exceptions module

exception durga.exceptions.**DurgaError**
Bases: `Exception`

Main exception class.

exception durga.exceptions.**MultipleObjectsReturnedError**
Bases: *durga.exceptions.DurgaError*

The request returned multiple objects when only one was expected.

That is, if a GET request returns more than one element.

exception durga.exceptions.**ObjectNotFoundError**
Bases: *durga.exceptions.DurgaError*

The requested object does not exist.

exception durga.exceptions.**ValidationError**
Bases: *durga.exceptions.DurgaError*

The value did not pass the validator.

durga.resource module

class durga.resource.**Resource**
Bases: `object`

collection

dispatch(*request*)
Dispatch the Request instance and return an Response instance.

extract(*response*)
Return a list of JSON data extracted from the response.

headers = {}

id_attribute
`Element` attribute name to be used as primary id.

object_path = ()

objects_path = ()

schema = None

url
Full URL of the resource.

validate(*data*)
Validate the passed data.

If data is empty or no schema is defined the data is not validated and returned as it is.

durga.validators module

durga.validators.**email**(*value*)
Check if `value` is a valid email address.

durga.validators.**url**(*value*)
Check if `value` is a valid URL.

durga.validators.**uuid4**(*value*)
Check if `value` is a valid UUID version 4.

2.5 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:

2.5.1 Types of Contributions

Report Bugs

Report bugs at the GitHub issue tracker.

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.

Fix Bugs

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

Implement Features

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

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at the GitHub issue tracker.

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 :)

2.5.2 Get Started!

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

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

```
$ git clone git@github.com:your_name_here/durga.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 durga
$ cd durga
$ make 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:

```
$ make test
$ make test-all
```

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.

2.5.3 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 2.7 and 3.4. Check Travis CI and make sure that the tests pass for all supported Python versions.

2.5.4 Tips

To run a subset of tests:

```
$ make test TEST_ARGS='-k <EXPRESSION>'
```

2.6 Authors

2.6.1 Development Lead

- Markus Zapke-Gründemann

2.6.2 Contributors

- René Muhl
- Max Brauer
- Arnold Krille
- Simon Jakobi

2.7 Changelog

- #1: Explain how to validate datetime strings using python-dateutil
- #4: Monitor dependencies with requires.io
- #7: Rename Resource.name to Resource.path and add placeholders
- #8: Allow to add headers to the HTTP request
- #9: Do not require a schema
- #11: Overriding the Element class
- #14: Add check for Manifest
- #15: Add Makefile target to make a test release
- #20: Add Coveralls to monitor and display test coverage
- #21: Use separate tox env for flake8
- #22: Use releases to manage changelog
- #2: Write custom schema object to validate URLs
- #40: Improve and extend validators
- #34: Use Shields.io for badges

- #48: Move contribution guideline to separate file
- : Basic functionality.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

d