Pyresto

Release 0.4-dev

Sep 27, 2017

Contents

An ORM package to prevent repetitive work for writing clients for RESTful APIs.

Source https://github.com/BYK/pyresto/

Issues https://github.com/BYK/pyresto/issues/

PyPi http://pypi.python.org/pypi/pyresto/

CHAPTER 1

Getting Started

Install with **pip** or **easy_install**:

```
pip install pyresto
easy_install pyresto
```

or download the latest version from GitHub:

```
git clone git://github.com/BYK/pyresto.git
cd pyresto
python setup.py develop
```

CHAPTER 2

Documentation

Tutorial

Welcome to the pyresto tutorial. This tutorial will guide you through the development of a REST interface for the Github API. The implementation can be found in the Pyresto source repository in the `pyresto.apis.github` module.

The Base

Start off by creating a base model class for the service you are using which will hold the common values such as the API host, the common model representation using `__repr__` etc:

```
class GitHubModel(Model):
    _url_base = 'https://api.github.com'

    def __repr__(self):
        if hasattr(self, '_links'):
            desc = self._links['self']
        elif hasattr(self, 'url'):
            desc = self.url
        else:
            desc = self._current_path

        return '<GitHub.{0} [{1}]>'.format(self.__class__.__name__, desc)
```

Simple Models

Then continue with implementing simple models which does not refer to any other model, such as the `Comment` model for GitHub:

```
class Comment(GitHubModel):
    _path = '/repos/{repo_name}/comments/{id}'
    _pk = ('repo_name', 'id')
```

Note that we didn't define *any* attributes except for the mandatory `_path` and `_pk` attributes since pyresto automatically fills all attributes provided by the server response. This inhibits any possible efforts to implement client side verification though since the server already verifies all the requests made to it, and results in simpler code. This also makes the models "future-proof" and conforms to the best practices for "real" RESTful or Hypermedia APIs, which many recently started to use as a term instead of "real RESTful".

Relations

After defining some "simple" models, you can start implementing models having relations with each other:

```
class Commit(GitHubModel):
    _path = '/repos/{repo_name}/commits/{sha}'
    _pk = ('repo_name', 'sha')
    comments = Many(Comment, '{self._current_path}/comments?per_page=100')
```

Note that we used the attribute name `comments` which will "shadow" any attribute named "comments" sent by the server as documented in *Model*, so be wise when you are choosing your relation names and use the ones provided by the service documentation if there are any.

Note that we used the *Many* relation here. We provided the model class itself, which will be the class of all the items in the collection and, the path to fetch the collection. We used `commit.url` in the path format where `commit` will be the commit instance we are bound to, or to be clear, the commit "resource" which we are trying to get the comments of.

Since we don't expect many comments for a given commit, we used the default *Many* implementation which will result in a *WrappedList* instance that can be considered as a `list`. This will cause a chain of requests when this attribute is first accessed until all the comments are fetched and no "next" link can be extracted from the `Link` header. See *Model._continuator* for more info on this.

If we were expecting lots of items to be in the collection, or an unknown number of items in the collection, we could have used `lazy=True` like this:

```
class Repo(GitHubModel):
    _path = '/repos/{full_name}'
    _pk = 'full_name'
    commits = Many(Commit, '{self._current_path}/commits?per_page=100', lazy=True)
    comments = Many(Comment, '{self._current_path}/comments?per_page=100')
    tags = Many(Tag, '{self._current_path}/tags?per_page=100')
    branches = Many(Branch, '{self._current_path}/branches?per_page=100')
    keys = Many(Key, '{self._current_path}/keys?per_page=100')
```

Using `lazy=True` will result in a *LazyList* type of field on the model when accessed, which is basically a generator. So you can iterate over it but you cannot directly access a specific element by index or get the total length of the collection.

You can also use the *Foreign* relation to refer to other models:

```
class Tag(GitHubModel):
    _path = '/repos/{repo_name}/tags/{name}'
    _pk = ('repo_name', 'name')
    commit = Foreign(Commit, embedded=True)
```

When used in its simplest form, just like in the code above, this relation expects the primary key value for the model it is referencing, `Commit` here, to be provided by the server under the **same** name. So we expect from GitHub API to provide the commit sha, which is the primary key for `Commit` models, under the label `commit` when we fetch the data for a `Tag`. When this property is accessed, a simple *`Model.get`* call is made on the `Commit` model, which fetches all the data associated with the it and puts them into a newly created model instance.

Late Bindings

Since all relation types expect the class object itself for relations, it is not always possible to put all relation definitions inside the class definition. For those cases, you can simply late bind the relations as follows:

```
# Late bindings due to circular references
Commit.committer = Foreign(User, '__committer', embedded=True)
Commit.author = Foreign(User, '__author', embedded=True)
Repo.contributors = Many(User,
                         '{self._current_path}/contributors?per_page=100')
Repo.owner = Foreign(User, '__owner', embedded=True)
Repo.watcher_list = Many(User, '{self._current_path}/watchers?per_page=100')
User.follower_list = Many(User, '{self._current_path}/followers?per_page=100')
User.watched = Many(Repo, '{self._current_path}/watched?per_page=100')
```

Authentication

Most services require authentication even for only fetching data so providing means of authentication is essential. Define the possible authentication mechanisms for the service:

```
from ...auth import HTTPBasicAuth, AppQSAuth, AuthList, enable_auth
# Define authentication methods
auths = AuthList(basic=HTTPBasicAuth, app=AppQSAuth)
```

Make sure you use the provided authentication classes by `requests.auth` if they suit your needs. If you still need a custom authentication class, make sure you derive it from `Auth`.

After defining the authentication methods, create a module-global function that will set the default authentication method and credentials for all requests for convenience:

```
# Enable and publish global authentication
auth = enable_auth(auths, GitHubModel, 'app')
```

Above, we provide the list of methods/classes we have previously defined, the base class for our service since all other models inherit from that and will use the authentication defined on that, unless overridden. And we also set our default authentication mechanism to remove the burden from the shoulders of the users of our API library.

CHAPTER 3

API Documentation

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

API

pyresto.core.ModelBase

class pyresto.core.**ModelBase**
Bases: abc.ABCMeta

Meta class for *Model* class. This class automagically creates the necessary *Model._path* class variable if it is not already defined. The default path pattern is /modelname/{id}.

pyresto.core.Model

class pyresto.core.**Model**(***kwargs*)
Bases: object

The base model class where every data model using pyresto should be inherited from. Uses *ModelBase* as its metaclass for various reasons explained in *ModelBase*.

__init__(***kwargs*)
Constructor for model instances. All named parameters passed to this method are bound to the newly created instance. Any property names provided at this level which are interfering with the predefined class relations (especially for *Foreign* fields) are prepended "__" to avoid conflicts and to be used by the related relation class. For instance if your class has father = Foreign(Father) and father is provided to the constructor, its value is saved under __father to be used by the *Foreign* relationship class as the id of the foreign *Model*.

Constructor also tries to populate the Model._current_path instance variable by formatting *Model._path* using the arguments provided.

_url_base = None
The class variable that holds the bae uel for the API endpoint for the *Model*. This should be a "full" URL including the scheme, port and the initial path if there is any.

_path = None
The class variable that holds the path to be used to fetch the instance from the server. It is a format string using the new format notation defined for `str.format()`. The primary key will be passed under the same name defined in the *_pk* property and any other named parameters passed to the *Model.get()* or the class constructor will be available to this string for formatting.

_auth = None
The class variable that holds the default authentication object to be passed to `requests`. Can be overridden on either class or instance level for convenience.

_parser = <function loads>
The class method which receives the class object and the body text of the server response to be parsed. It is expected to return a dictionary object having the properties of the related model. Defaults to a "staticazed" version of `json.loads()` so it is not necessary to override it if the response type is valid JSON.

_fetched = False
The instance variable which is used to determine if the *Model* instance is filled from the server or not. It can be modified for certain usages but this is not suggested. If *_fetched* is `False` when an attribute, that is not in the class dictionary, tried to be accessed, the `__fetch()` method is called before raising an `AttributeError`.

_get_params = {}
The instance variable which holds the additional named get parameters provided to the *Model.get()* to fetch the instance. It is used internally by the *Relation* classes to get more info about the current *Model* instance while fetching its related resources.

classmethod _continuator(*response*)
The class method which receives the response from the server. This method is expected to return a continuation URL for the fetched resource, if there is any (like the next page's URL for paginated content) and `None` otherwise. The default implementation uses the standard HTTP link header and returns the url provided under the label "next" for continuation and `None` if it cannot find this label.

Parameters **response** (`requests.Response`) – The response for the HTTP request made to fetch the resources.

_id
A property that returns the instance's primary key value.

_pk
The class variable where the attribute name for the primary key for the *Model* is stored as a string. This property is required and not providing a default is intentional to force developers to explicitly define it on every *Model* class.

classmethod _rest_call(*url*, *method='GET'*, *fetch_all=True*, ***kwargs*)
A method which handles all the heavy HTTP stuff by itself. This is actually a private method but to let the instances and derived classes to call it, is made `protected` using only a single _ prefix.

All undocumented keyword arguments are passed to the HTTP request as keyword arguments such as method, url etc.

Parameters **fetch_all** (*boolean*) – (optional) Determines if the function should recursively fetch any "paginated" resource or simply return the downloaded and parsed data along with a continuation URL.

Returns Returns a tuple where the first part is the parsed data from the server using

Model._parser, and the second half is the continuation URL extracted using *Model._continuator* or None if there isn't any.

Return type tuple

classmethod get (**args*, ***kwargs*)

The class method that fetches and instantiates the resource defined by the provided pk value. Any other extra keyword arguments are used to format the *Model._path* variable to construct the request URL.

Parameters **pk** (*string*) – The primary key value for the requested resource.

Return type *Model* or None

pyresto.core.Auth

pyresto.core.AuthList

pyresto.core.enable_auth

pyresto.core.Relation

class pyresto.core.**Relation**

Bases: object

Base class for all relation types.

pyresto.core.Many

class pyresto.core.**Many** (*model*, *path=None*, *lazy=False*, *preprocessor=None*)

Bases: *pyresto.core.Relation*

Class for 'many' *Relation* type which is essentially a collection for a certain model. Needs a base *Model* for the collection and a *path* to get the collection from. Falls back to provided model's Model.path if not provided.

__init__ (*model*, *path=None*, *lazy=False*, *preprocessor=None*)

Constructor for Many relation instances.

Parameters

- **model** (Model) – The model class that each instance in the collection will be a member of.
- **path** (*string or None*) – (optional) The unicode path to fetch the collection items, if different than *Model._path*, which usually is.
- **lazy** (*boolean*) – (optional) A boolean indicator to determine the type of the *Many* field. Normally, it will be a *WrappedList* which is essentially a list. Use lazy=True if the number of items in the collection will be uncertain or very large which will result in a *LazyList* property which is practically a generator.

_Many__make_fetcher (*url*, *instance*)

A function factory method which creates a simple fetcher function for the *Many* relation, that is used internally. The *Model._rest_call()* method defined on the models is expected to return the data and a continuation URL if there is any. This method generates a bound, fetcher function that calls the internal *Model._rest_call()* function on the *Model*, and processes its results to satisfy the requirements explained above.

Parameters **url** (*unicode*) – The url which the fetcher function will be bound to.

_with_owner(*owner*)
A function factory method which returns a mapping/wrapping function. The returned function creates a new instance of the *Model* that the *Relation* is defined with, sets its owner and "automatically fetched" internal flag and returns it.

Parameters **owner** (Model) – The owner Model for the collection and its items.

pyresto.core.Foreign

class pyresto.core.**Foreign**(*model*, *key_property=None*, *key_extractor=None*, *embedded=False*)
Bases: *pyresto.core.Relation*

Class for 'foreign' *Relation* type which is essentially a reference to a certain *Model*. Needs a base *Model* for obvious reasons.

__init__(*model*, *key_property=None*, *key_extractor=None*, *embedded=False*)
Constructor for the *Foreign* relations.

Parameters

- **model** (Model) – The model class for the foreign resource.
- **key_property** (*string or None*) – (optional) The name of the property on the base *Model* which contains the id for the foreign model.
- **key_extractor** (*function(model)*) – (optional) The function that will extract the id of the foreign model from the provided *Model* instance. This argument is provided to make it possible to handle complex id extraction operations for foreign fields.

pyresto.core.WrappedList

class pyresto.core.**WrappedList**(*iterable*, *wrapper*)
Bases: list

Wrapped list implementation to dynamically create models as someone tries to access an item or a slice in the list. Returns a generator instead, when someone tries to iterate over the whole list.

pyresto.core.LazyList

class pyresto.core.**LazyList**(*wrapper*, *fetcher*)
Bases: object

Lazy list implementation for continuous iteration over very large lists such as commits in a large repository. This is essentially a chained and structured generator. No caching and memoization at all since the intended usage is for small number of iterations.

pyresto.core.PyrestoException

pyresto.core.PyrestoServerResponseException

pyresto.core.PyrestoInvalidRestMethodException

pyresto.core.PyrestoInvalidAuthTypeException

Python Module Index

p

Index

Symbols

F

G

L

M

P

R

W