nap Documentation

Release 0.1

Jacob Burch

October 27, 2016

Contents

CHAPTER 1

nap

Accessing APIs open-endily is an easy affair. pip install requests, pass in your data get data back. But the slight differences and demands of different API creating code that's structured, re-usable and simple proves difficult.

nap hopes to help.

nap aims to be:

- Support Read (GET) and Write (POST/PUT/PATCH)
- Little to no configuration needed for to-spec REST APIs
- Easy to configure to fit any REST-like API
- Customize to fit even edgier use cases

Contents:

1.1 Quickstart

1.1.1 Step One: Declare your resource

```
# note/client.py
from nap.resources import ResourceModel, Field


class Note(ResourceModel):

    pk = Field(api_name='id', resource_id=True)
    title = Field()
    content = Field()

    class Meta:
        root_url = 'http://127.0.0.1:8000/api/'
        resource_name = 'note'
```

1.1.2 Step Two: Access your api

```
from note.client import Note

n = Note(title='Some Title', content="some content")
```

```
# POST http://127.0.0.1:8000/api/note/
n.save()

n = Note.objects.get('note/1/')
# Some Title
n.title

# GET http://127.0.0.1:8000/api/note/1/
n = Note.objects.lookup(pk=1)
n.title = "New Title"
n.content = "I sure do love butterflies"

# PUT http://127.0.0.1:8000/api/note/1/
n.save()

n = Note.objects.get('note/1/')
# "New Title"
n.title
```

1.1.3 Step Three: Set up custom lookup_urls

```
from nap.resources import ResourceModel, Field
from nap.lookup import nap_url

class Note(ResourceModel):

    pk = Field(api_name='id', resource_id=True)
    title = Field()
    content = Field()

    class Meta:
        root_url = 'http://127.0.0.1:8000/api/'
        resource_name = 'note'
        additional_urls = (
            nap_url(r'%(resource_name)s/title/%(title)s/'),
        )

# GET http://127.0.0.1:8000/api/note/title/butterflies/
n = Note.objects.lookup(title='New Title')
# "I sure do love butterflies"
n.content
```

1.1.4 Step Four: What's next?

- Learn more about tweaking `ResourceModel` by looking at `tutorial/tutorial1`
- Learn about LookupURLs, the glue between your resource and its API
- **Look deeper into the core modules behind nap:**
  - ResourceModel API, The Pythonic representation of your resource.
  - `engine`, all the HTTP nuts-and-bolts powering nap.
- **Tutorial.**
  - `Part 1: a Tastypie API`

```
- Part 2:  Github Gist API
```

1.2 ResourceModel API

The ResourceModel is the main component to interaction with nap. While not intereacted with directly, subclasses of nap provide the functionality needed to working with APIs. Because of this, ResourceModel is designed to have as many hooks as possible to tweak the functionality of it's primary method calls.

1.2.1 Fields

Main article: Fields

Each field is a one-to-one mapping with a attribute returned in your API's response. It handles any validation and coerition (*scrubbing*) necessary to turn the API data into Python (and back to an API data string again). Special fields can be used to group API data into sub-collections of ResourceModels.

1.2.2 Lookup URLs

Main article: URLs

LookupURLs power the main engine of nap. By defining dynamic URLs, API get/create/update operations can be issued without specifying a raw URL, and instead the necessary data to complete the operation.

How LookupURLs work and

1.2.3 Options

Main article: Options

1.2.4 API

1.3 Fields

1.4 URLs

Proper URL patterns are the backbone of a ResourceModel. URLs are defined in a ResourceModel as a tuple of `nap_urls`–a thin wrapper around a python-formatted string. These are defined and tied to a ResourceModel through the *urls*, *prepend_urls*, and *append_urls*. These are stored in the ResourceModel's `_meta`

By default, ResourceModels have two nap_urls that allow them to make all common calls to a to-spec REST API:

```
(
    nap_url('%(resource_name)s/', create=True, lookup=False, collection=True),
    nap_url('%(resource_name)s/%(resource_id)s/', update=True),
)
```

1.4.1 How a URL lookup works

nap_urls contain three parts of information:

1. The kinds of lookups that this url can be used for.
2. The URL String itself
3. The names of variables needed to generate

On calls that are backed by a URL, Nap will iterate through every URL in it's url list looking for a match. A match is considered a URL where

1. The URL is valid for the type of request being attempted
2. The variablres required to generated a valid URL are available.

Let's dive into each part of a URL to understand this process a bit better.

Lookup Types

Lookup types closely match the kind of operations possible with an API. They are `create`, `lookup`, `update`, `collection`.

- **create**: URLs that can be used to create new resources. Used for the `create()` method.
- **lookup**: URLs that can be used to retreive a single resource. Used for the `get()` method when using keyword arguments.
- **update**: URLs that can be used to create update an existing resource. Used for the `update()` method.
- **collection**: URLs that can be used to retrieve collections of resources. Used for the `filter()` and `all()` methods.

Valid URL Strings

A URL string is simply a python string, optionally containing dictionary-format variables.

nap bases it's *required variables* partially on any format variables contained in the URL string.

URL Variables

LookupURLs may require variables to fully resolve. Required variables are either

1. Python string format variables contained in the url_string, or
2. Any variables named passed into a `__init__`'s `param` parameter. These variables are passed into the URL via a URL query string.

`ResourceModel` passes in three kinds of variables into the LookupURL's match function to determine if all required variables are available for URL resolution:

1. Keyword arguments passed to lookup function (eg, `ResourceModel.get()`, `ResourceModel.update()`)
2. The values of fields, where the name of the field is passed as the variable name.
3. Meta variables specific to the subclass of ResourceModel

The above lists these groups in order of presidence–eg, If `update()` is called on a ResourceModel with a resource_name of 'person', but a keyword argument of `resource_name='author'`, %(resource_name)s will resolve to `author`.

Meta Variables available for URLs

Currently, there is only one meta variable passed to LookupURL.

`resource_name`

The resource name of the ResourceModel. Equal to *resource_name*

1.4.2 URL API

class `nap.lookup.`**`LookupURL`**
Class in charge of resolving variable URLs based on keyword arguments. Used for any dynamic API method on `ResourceModel`

`LookupURL.`**`__init__`**(*url_string*[, *params=None*, *create=False*, *update=False*, *lookup=False*, *collection=False*])

Parameters

- **`url_string`** – python-formatted string representing a URL
- **`params`** – an iterable of variables names required by the URL, as passed in a GET query string.
- **`create`** – Designates whether or not the URL is valid for create operations
- **`update`** – Designates whether or not the URL is valid for create operations
- **`lookup`** – Designates whether or not the URL is valid for create operations
- **`collection`** – Designates whether or not the URL is valid for create operations

`LookupURL.`**`url_vars`**
Returns a tuple the names of variables contained within a `LookupURL`

`LookupURL.`**`required_vars`**
Returns a tuple of the the names of all variables required to successfully resolve a URL.

`LookupURL.`**`match`**(***lookup_vars*)
Attempts to resolve a string of a URL, resolving any variables based on `lookup_vars`.

Returns a two tuple of the matching URL string and extra lookup variables that were passed in but not part of the required values.

If no match is found, a two tuple of (`None`, `None`) is returned.

Parameters **`lookup_vars`** – A dict-like variable mapping URL variables names to

1.5 Options

1.5.1 `resource_name`

Optional

The name used to refer to a source in URLs. `resource_name` is appended to `root_url` to create the default url set.

Defaults to: ResourceModel's class name, in all lower case. eg:

```
class FooBar(ResourceModel):
    # fields here..
    # resource_name is `foobar`
```

1.5.2 root_url

Optional

Defaults to: `None`

1.5.3 urls

Optional

URLs used to lookup API requests. See URLs for more information on definging ResourceModel urls.

Defaults to: A tuple of urls set to:

```
(
    nap_url('%(resource_name)s/',
        create=True,
        lookup=False,
        collection=True
    ),
    nap_url('%(resource_name)s/%(resource_id)s', update=True),
)
```

1.5.4 append_urls

Optional

URLs to be added after ResourceModel's default_urls. See URLs for more information on definging ResourceModel urls.

Defaults to: () (an empty tuple)

1.5.5 prepend_urls

Optional

URLs to be added before ResourceModel's default_urls. See URLs for more information on definging ResourceModel urls.

Defaults to: () (an empty tuple)

1.5.6 add_slash

Optional

Determines whehter or not slashes are appended to a url.

- If `True`, slashes will always be added to the end of URLs.
- If `False`, slashes will always be removed from the end of URLs
- If `None`, URLs will follow what is defined in the `nap_url` string.

Defaults to: `None`

1.5.7 update_from_write

Optional

Determines whether or not nap attempts to change an object's field data based on the HTTP content of create and update requests.

Defaults to: `True`

1.5.8 update_method

Optional

String representing HTTP method used to update (edit) resource objects.

Defaults to: "PUT"

1.5.9 auth

Iterable of authorization classes. See :doc:auth for more information on Authorization classes.

Defaults to: () (an empty tuple)

1.6 Authorization

class HttpAuthorization

1.7 About

1.7.1 Warnings about API Design.

REST, similarly so many wonderful technological buzzronyms before it, was something specific that has come to mean something vaguely "not SOAP." Because of this, nap triesto pick safe, undestructive defaults. The tradeoff with this decision is there is a little bit more customization required to make things work than a more traditional modeling backend (such as a relational database). To facilitate this, nap has an extensive list of options to make setting these configuration easy as possible.

Your nap models will only go as far as your API allows. For instance, if your API's collections only return partial data about your object, you won't have access to the left out fields (and risk saving over them without proper configuration!). The more you expose in your API the easier using nap will be.

1.7.2 Thanks

nap is the spiritual descendant of remote objects, and owes the core idea to it's leg work – and both owe a great deal to the declarative syntax of Django models and SQLAlchemy

1.8 Roadmap

1.8.1 Future Version Feature List

0.1.1

- Minimal Feature complete.
- Handling of Tastypie-stype collections (contained within 'objects')
- Proper Error Raising

0.2.0

- Inheritence Features and Tests

n

Symbols

H

L

M

N

R

U