RAMLfications Documentation

Release 0.1.0

Lynn Root

May 15, 2015

Contents

Release v0.1.0 (What's new?).

`ramlfications` is an Apache 2.0-licensed reference implementation of a RAML parser in Python intended to be used for parsing API definitions (e.g. for static documentation-generation).

If you've never heard of RAML, you're missing out:

> RESTful API Modeling Language (RAML) is a simple and succinct way of describing practically-RESTful APIs. It encourages reuse, enables discovery and pattern-sharing, and aims for merit-based emergence of best practices. The goal is to help our current API ecosystem by solving immediate problems and then encourage ever-better API patterns. RAML is built on broadly-used standards such as YAML and JSON and is a non-proprietary, vendor-neutral open spec.

CHAPTER 1

Why `ramlfications` and not pyraml-parser?

I chose to write a new library rather than wrestle with pyraml-parser as it was not developer-friendly to extend (in my PoV, others may have more success) and did not include required RAML features (e.g. uriParameters, parsing of security schemes, etc), as well as a lot of meta programming that is just simply over my head. However, I do encourage you to check out pyraml-parser! You may find it easier to work with than I did.

CHAPTER 2

About

`ramlfications`'s documentation lives at Read the Docs, the code on GitHub. It's tested on Python 2.6, 2.7, 3.3+, and PyPy. Both Linux and OS X are supported.

CHAPTER 3

User's Guide

3.1 Requirements and Installation

3.1.1 User Setup

The latest stable version can be found on PyPI, and you can install via pip:

```
$ pip install ramlfications
```

`ramlfications` runs on Python 2.6, 2.7, and 3.3+, and PyPy. Both Linux and OS X are supported.

Continue onto usage to get started on using `ramlfications`.

3.1.2 Developer Setup

If you'd like to contribute or develop upon `ramlfications`, be sure to read How to Contribute first.

System requirements:

- C Compiler (gcc/clang/etc.)
- If on Linux - you'll need to install Python headers (e.g. `apt-get install python-dev`)
- Python 2.6, 2.7, 3.3+, or PyPy
- virtualenv

Here's how to set your machine up:

```
$ git clone git@github.com:spotify/ramlfications
$ cd ramlfications
$ virtualenv env
$ source env/bin/activate
(env) $ pip install -r dev-requirements.txt
```

Run Tests

If you'd like to run tests for all supported Python versions, you must have all Python versions installed on your system. I suggest pyenv to help with that.

To run all tests:

```
(env) $ tox
```

To run a specific test setup (options include: `py26`, `py27`, `py33`, `py34`, `pypy`, `flake8`, `verbose`, `manifest`, `docs`, `setup`, `setupcov`):

```
(env) $ tox -e py26
```

To run tests without tox:

```
(env) $ py.test
(env) $ py.test --cov ramlfications --cov-report term-missing
```

Build Docs

Documentation is build with Sphinx, written in rST, uses the Read the Docs theme with a slightly customized CSS, and is hosted on Read the Docs site.

To rebuild docs locally, within the parent `ramlfications` directory:

```
(env) $ tox -e docs
```

or:

```
(env) $ sphinx-build -b docs/ docs/_build
```

Then within `ramlfications/docs/_build` you can open the index.html page in your browser.

3.2 Usage

You can use `ramlfications` to parse and validate a RAML file with Python. With the command line, you can validate or visualize the RAML-defined API as a tree.

3.2.1 Parse

To parse a RAML file, include ramlfications in your project and call the parse function:

```
>>> import ramlfications
>>> RAML_FILE = "/path/to/my-api.raml"
>>> api = ramlfications.parse(RAML_FILE)
>>> api
<RootNode(raml_file="path/to/my-api.raml")>
>>> api.title
'My Foo API'
>>> api.version
'v1'
```

```
>>> api.security_schemes
[<SecurityScheme(name="oauth_2_0")>]
>>> oauth2 = api.security_schemes[0]
>>> oauth2.name
'oauth_2_0'
>>> oauth2.type
'OAuth 2.0'
>>> oauth2.settings.scopes
['playlist-read-private', 'playlist-modify-public',..., 'user-read-email']
```

```
>>> oauth2.settings.access_token_uri
'https://accounts.foo.com/api/token'
```

```
>>> api.resources
[<ResourceNode(method='GET', path='/foo')>, <ResourceNode(method='PUT', path='/foo')>, ..., <Resource
>>> foo_bar = api.resources[-1]
>>> foo_bar.name
'/{id}'
>>> foo_bar.display_name
'fooBarID'
>>> foo_bar.absolute_path
'https://api.foo.com/v1/foo/bar/{id}'
>>> foo_bar.uri_params
[<URIParameter(name='id')>]
```

```
>>> uri_param = foo_bar.uri_params[0]
>>> uri_param.required
True
>>> uri_param.type
'string'
>>> id_param.example
'f00b@r1D'
```

You can pass in an optional config file to add additional values for certain parameters. Find out more within the Extended Usage:

```
>>> CONFIG_FILE = "/path/to/config.ini"
>>> api = ramlfications.parse(RAML_FILE, CONFIG_FILE)
```

For more complete understanding of what's available when parsing a RAML file, check the Extended Usage or the API Definition.

3.2.2 Validate

Validation is according to the RAML Specification.

To validate a RAML file via the command line:

```
$ ramlfications validate /path/to/my-api.raml
Success! Valid RAML file: tests/data/examples/simple-tree.raml
```

```
$ ramlfications validate /path/to/invalid/no-title.raml
Error validating file /path/to/invalid/no-title.raml: RAML File does not define an API title.
```

To validate a RAML file with Python:

```
>>> from ramlfications import validate
>>> RAML_FILE = "/path/to/my-api.raml"
>>> validate(RAML_FILE)
>>>
```

```
>>> from ramlfications import validate
>>> RAML_FILE = "/path/to/invalid/no-title.raml"
>>> validate(RAML_FILE)
InvalidRootNodeError: RAML File does not define an API title.
```

Note: When using `validate` within Python (versus the command line utility), if the RAML file is valid, then nothing is returned. Only invalid files will return an exception.

3.2.3 Tree

To visualize a tree output of a RAML file:

```
$ ramlfications tree /path/to/my-api.raml [-c|--color light/dark] [-v|vv|vvv] [-o|--output]
```

The least verbose option would show something like this:

```
$ ramlfications tree /path/to/my-api.raml
==========
My Foo API
==========
Base URI: https://api.foo.com/v1
|- /foo
|  - /bar
|  - /bar/{id}
```

And the most verbose:

```
$ ramlfications tree /path/to/my-api.raml -vvv
==========
My Foo API
==========
Base URI: https://api.foo.com/v1
|- /foo
|   GET
|     Query Params
|       q: Foo Query
|       type: Item Type
|  - /bar
|     GET
|       Query Params
|         q: Bar Query
|         type: item type
|  - /bar/{id}
|     GET
|       URI Params
|         id: ID of foo
```

3.2.4 Options and Arguments

The full usage is:

```
$ ramlfications [OPTIONS] COMMAND RAMLFILE
```

The `RAMLFILE` is a file containing the RAML-defined API you'd like to work with.

Valid `COMMAND` s are the following:

`validate`
: Validate the RAML file according to the RAML Specification.

`tree`
: Visualize the RAML file via your console.

Valid `OPTIONS` for all commands are the following:

--help
Show a brief usage summary and exit.

Valid `OPTIONS` for the `tree` command are the following:

-c `light|dark`
Use a light color scheme for dark terminal backgrounds [DEFAULT], or dark color scheme for light backgrounds.

--color `light|dark`
Use a light color scheme for dark terminal backgrounds [DEFAULT], or dark color scheme for light backgrounds.

-o
Save tree output desired file

--output
Save tree output desired file

-v
Increase verbose output of the tree one level: adds the HTTP methods

-vv
Increase verbose output of the tree one level: adds the parameter names

-vvv
Increase verbose output of the tree one level: adds the parameter display name

3.3 Extended Usage

To parse a RAML file, include ramlfications in your project and call the parse function:

```
>>> import ramlfications
>>> RAML_FILE = "/path/to/my-api.raml"
>>> api = ramlfications.parse(RAML_FILE)
```

3.3.1 Configuration

Perhaps your API supports response codes beyond what IETF supports (default for this parser). Or maybe you implemented your own authentication scheme that your API uses [I hope not!].

Example configuration file:

```
[main]
validate = True

[custom]
append = True
resp_codes = 420, 421, 422
auth_schemes = oauth_3_0, oauth_4_0
media_types = application/vnd.github.v3, foo/bar
protocols = FTP
raml_versions = 0.8
```

Feed the configuration into the parse function like so:

```
>>> import ramlfications
>>> RAML_FILE = "/path/to/my-api.raml"
>>> CONFIG_FILE = "/path/to/my-config.ini"
>>> api = ramlfications.parse(RAML_FILE, CONFIG_FILE)
```

3.3.2 RAML Root Section

In following the RAML Spec's Root Section definition, here is how you can access the following attributes:

The Basics

```
>>> api.title
'My Other Foo API'
>>>
>>> api.version
v2
>>> api.base_uri
'https://{domainName}.foo.com/v2'
>>> api.base_uri_parameters
[<URIParameter(name='domainName')>]
>>>
>>> api.protocols
['HTTPS']
```

API Documentation

```
>>> api.documentation
[<Documentation(title='The Foo API Docs')>]
>>> doc = api.documentation[0]
>>> doc.title
'The Foo API Docs'
```

Docs written in the RAML file should be written using Markdown. This also applies to any `description` parameter.

With `ramlfications`, documentation content and descriptions can either be viewed raw, or in parsed HTML.

```
>>> doc.content
'Welcome to the _Foo API_ specification. For more information about\nhow to use the API, check out [
>>>
>>> doc.content.html
u'<p>Welcome to the <em>Foo API</em> specification. For more information about\nhow to use the API, c
```

Check out API Definition for full definition of `RootNode` and its associated attributes and objects.

Security Schemes

RAML supports OAuth 1, OAuth 2, Basic & Digest, and any authentication scheme self-defined with an `x-{other}` header.

To parse auth schemes:

```
>>> api.security_schemes
[<SecurityScheme(name='oauth_2_0')>]
>>> oauth2 = api.security_schemes[0]
```

```
>>> oauth2.name
'oauth_2_0'
>>> oauth2.type
'OAuth 2.0'
>>> oauth2.description
'Foo supports OAuth 2.0 for authenticating all API requests.\n'
>>> oauth2.description.html
u'<p>Foo supports OAuth 2.0 for authenticating all API requests.</p>\n'
```

And its related Headers and Responses:

```
>>> oauth2.described_by
{'headers': [<Header(name='Authorization')>], 'responses': [<Response(code='401')>, <Response(code='4
>>> first_header = oauth2.described_by['headers'][0]
>>> first_header
<HeaderParameter(name='Authorization')>
>>> first_header.name
'Authorization'
>>> first_headers.description
'Used to send a valid OAuth 2 access token.\n'
>>> first_headers.description.html
u'<p>Used to send a valid OAuth 2 access token.</p>\n'
>>> resps = oauth2.described_by['responses']
>>> resps
[<Response(code='401')>, <Response(code='403')>]
>>> resp[0].code
401
>>> resp[0].description.raw
'Bad or expired token. This can happen if the user revoked a token or\nthe access token has expired.
```

Authentication settings (available for OAuth1, OAuth2, and any x-header that includes "settings" in the RAML definition).

```
>>> oauth2.settings.scopes
['foo-read-private', 'foo-modify-public',..., 'user-read-email-address']
>>> oauth2.settings.access_token_uri
'https://accounts.foo.com/api/token'
>>> oauth2.settings.authorization_grants
['code', 'token']
>>> oauth2.settings.authorization_uri
'https://accounts.foo.com/authorize'
```

Check out API Definition for full definition of `SecuritySchemes`, `Header`, `Response` and their associated attributes and objects.

Traits & Resource Types

Traits & resource types help when API definitions get a bit repetitive. More information can be found in the RAML spec for resource types and traits.

Resource Types

```
>>> api.resource_types
[<ResourceTypeNode(name='collection')>, <ResourceTypeNode(name='member')>]
>>> collection = api.resource_types[0]
>>> collection.name
```

```
'collection'
>>> collection.description
'The collection of <<resourcePathName>>'
>>> collection.usage
'This resourceType should be used for any collection of items'
>>> collection.method
'get'
>>> get.optional
False
```

Traits

```
>>> api.traits
[<TraitNode(name='filtered')>, <TraitNode(name='paged')>]
>>> paged = api.traits[1]
>>> paged.query_params
[<QueryParameter(name='offset')>, <QueryParameter(name='limit')>]
>>> paged.query_params[0].name
'offset'
>>> paged.query_params[0].description
'The index of the first track to return'
```

Mapping of Properties and Elements from Traits & Resource Types to Resources

When a resource has a trait and/or type assigned to it, or a resource type has another resource type or a trait assigned to it, it inherits its properties.

Also, the RAML Spec allows for parameters within Traits and ResourceTypes, denoted by double brackets within the Trait/ResourceType definition, e.g. `<<parameter>>`. After the parsing of the API definition, the appropriate parameters are filled in for the respective resource.

For example, a simplified RAML file:

```
#%RAML 0.8
title: Example API - Mapped Traits
version: v1
resourceTypes:
  - searchableCollection:
      get:
        queryParameters:
          <<queryParamName>>:
            description: |
              Return <<resourcePathName>> that have their <<queryParamName>>
              matching the given value
          <<fallbackParamName>>:
            description: |
              If no values match the value given for <<queryParamName>>,
              use <<fallbackParamName>> instead
  - collection:
      usage: This resourceType should be used for any collection of items
      description: The collection of <<resourcePathName>>
      get:
        description: Get all <<resourcePathName>>, optionally filtered
      post:
        description: Create a new <<resourcePathName | !singularize>>
traits:
```

```
  - secured:
      description: A secured method
      queryParameters:
        <<tokenName>>:
          description: A valid <<tokenName>> is required
  - paged:
      queryParameters:
        numPages:
          description: The number of pages to return, not to exceed <<maxPages>>
/books:
  type: { searchableCollection: { queryParamName: title, fallbackParamName: digest_all_fields } }
  get:
    is: [ secured: { tokenName: access_token }, paged: { maxPages: 10 } ]
```

When parsed, the Python notation would look like this:

```
>>> RAML_FILE = "/path/to/foo-api.raml"
>>> api = parse(RAML_FILE)
```

```
# accessing API-supported resource types
>>> api.resource_types
[<ResourceTypeNode(method='GET', name='searchableCollection')>,
<ResourceTypeNode(method='POST', name='collection')>,
<ResourceTypeNode(method='GET', name='collection')>]
>>> api.resource_types[0].query_params
[<QueryParameter(name='<<queryParamName>>')>,
<QueryParameter(name='<<fallbackParamName>>')>]
>>> api.resource_types[0].query_params[0].description
Return <<resourcePathName>> that have their <<queryParamName>> matching the given value
```

```
# accessing API-supported traits
>>> api.traits
[<TraitNode(name='secured')>, <TraitNode(name='paged')>]
>>> api.traits[0].query_params
[<QueryParameter(name='numPages')>]
>>> api.traits[0].query_params[0].description
The number of pages to return, not to exceed <<maxPages>>
```

```
# accessing a single resource
>>> books = api.resources[0]
>>> books
<ResourceNode(method='GET', path='/books')>
>>> books.type
{'searchableCollection': {'fallbackParamName': 'digest_all_fields', 'queryParamName': 'title'}}
>>> books.traits
[<TraitNode(name='secured')>, <TraitNode(name='paged')>]
>>> books.query_params
[<QueryParameter(name='title')>, <QueryParameter(name='digest_all_fields')>,
<QueryParameter(name='access_token')>, <QueryParameter(name='numPages')>]
>>> books.query_params[0].description
Return books that have their title matching the given value
>>> books.query_params[3].description
The number of pages to return, not to exceed 10
```

Check out API Definition for full definition of `traits` and `resources`, and its associated attributes and objects.

3.3.3 Resources

"Resources" are defined in the RAML Spec's Resource Section and is a relative URI (relative to the `base_uri` and, if nested, relative to its parent URI).

For example, *Foo API* defines `/foo/bar` as a resource (a "top-level resource" to be exact). It also defines `/{id}` under `/foo/bar`, making `/{id}` a nested resource, relative to `/foo/bar`. The relative path would be `/foo/bar/{id}`, and the absolute URI path would be `https://api.foo.com/v2/foo/bar/{id}`.

```
>>> api.resources
[<Resource(method='GET', path='/foo')>,..., <Resource(method='DELETE', path='/foo/bar/{id}')>]
>>>
>>> foo_bar = resources[-1]
>>> foo_bar.name
'/{id}'
>>> foo_bar.description
'[Delete a foo bar](https://developer.foo.com/api/delete-foo-bar/)\n'
>>> foo_bar.description.html
u'<p><a href="https://developer.foo.com/api/delete-foo-bar/">Delete a foo bar</a></p>\n'
>>> foo_bar.display_name
'foo bar'
>>> foo_bar.method
'delete'
>>> foo_bar.path
'/foo/bar/{id}'
>>> foo_bar.absolute_uri
'https://api.foo.com/v2/foo/bar/{id}'
>>> foo_bar.uri_params
[<URIParameter(name='id')>]
>>>
>>> uri_param = foo_bar.uri_params[0]
>>> uri_param.required
True
>>> uri_param.type
'string'
>>> uri_param.example
'f0ob@r1D'
>>> foo_bar.parent
<Resource(method='GET', path='/foo/bar/')>
```

Check out API Definition for full definition of what is available for a `resource` object, and its associated attributes and objects.

3.4 API Definition

3.4.1 Main functions

The following three functions are meant for you to use primarily when parsing a RAML file/string.

`ramlfications.`**`parse`**(*raml*, *config_file=None*)

Module helper function to parse a RAML File. First loads the RAML file with *`loader.RAMLLoader`* then parses with *`parser.parse_raml()`* to return a `raml.RAMLRoot` object.

Parameters

- **`raml`** – Either string path to the RAML file, a file object, or a string representation of RAML.

- **config_file** (*str*) – String path to desired config file, if any.

Returns parsed API

Return type RAMLRoot

Raises

- **LoadRAMLError** – If error occurred trying to load the RAML file (see *loader.RAMLLoader*)
- **RAMLParserError** – If error occurred during parsing of RAML file (see raml.RAMLRoot)
- **InvalidRamlFileError** – RAML file is invalid according to RAML specification.

ramlfications.**load**(*raml_file*)
Module helper function to load a RAML File using *loader.RAMLLoader*.

Parameters **raml_file** (*str*) – String path to RAML file

Returns loaded RAML

Return type dict

Raises LoadRAMLError If error occurred trying to load the RAML file

ramlfications.**loads**(*raml_string*)
Module helper function to load a RAML File using *loader.RAMLLoader*.

Parameters **raml_string** (*str*) – String of RAML data

Returns loaded RAML

Return type dict

Raises LoadRAMLError If error occurred trying to load the RAML file

ramlfications.**validate**(*raml*, *config_file=None*)
Module helper function to validate a RAML File. First loads the RAML file with *loader.RAMLLoader* then validates with validate.validate_raml().

Parameters

- **raml** (*str*) – Either string path to the RAML file, a file object, or a string representation of RAML.
- **config_file** (*str*) – String path to desired config file, if any.

Returns No return value if successful

Raises

- **LoadRAMLError** – If error occurred trying to load the RAML file (see *loader.RAMLLoader*)
- **InvalidRamlFileError** – If error occurred trying to validate the RAML file (see *validate*)

3.4.2 Core

Note: The following documentation is meant for understanding the underlying ramlfications API. No need to interact directly with the modules, classes, & functions below.

parser

ramlfications.parser.**parse_raml**(*loaded_raml*, *config*)
Parse loaded RAML file into RAML/Python objects.

Parameters **loaded_raml** (*RAMLDict*) – OrderedDict of loaded RAML file

Returns *raml.RootNode* object.

ramlfications.parser.**create_root**(*raml*, *config*)
Creates a Root Node based off of the RAML's root section.

Parameters **raml** (*RAMLDict*) – loaded RAML file

Returns *raml.RootNode* object with API root attributes set

ramlfications.parser.**create_traits**(*raml_data*, *root*)
Parse traits into `Trait` objects.

Parameters

- **raml_data** (*dict*) – Raw RAML data
- **root** (RootNode) – Root Node

Returns list of *raml.TraitNode* objects

ramlfications.parser.**create_resource_types**(*raml_data*, *root*)
Parse resourceTypes into `ResourceTypeNode` objects.

Parameters

- **raml_data** (*dict*) – Raw RAML data
- **root** (RootNode) – Root Node

Returns list of *raml.ResourceTypeNode* objects

ramlfications.parser.**create_resources**(*node*, *resources*, *root*, *parent*)
Recursively traverses the RAML file via DFS to find each resource endpoint.

Parameters

- **node** (*dict*) – Dictionary of node to traverse
- **resources** (*list*) – List of collected `ResourceNode` s
- **root** (RootNode) – The `RootNode` of the API
- **parent** (ResourceNode) – Parent `ResourceNode` of current `node`

Returns List of *raml.ResourceNode* objects.

ramlfications.parser.**create_node**(*name*, *raw_data*, *method*, *parent*, *root*)
Create a Resource Node object.

Parameters

- **name** (*str*) – Name of resource node
- **raw_data** (*dict*) – Raw RAML data associated with resource node
- **method** (*str*) – HTTP method associated with resource node
- **parent** (ResourceNode) – Parent node object of resource node, if any
- **api** (RootNode) – API `RootNode` that the resource node is attached to

Returns *raml.ResourceNode* object

raml

class ramlfications.raml.**RootNode**
API Root Node

raw
Ordered dict of all raw data from the RAML file/string.

version
str of API version.

base_uri
str of API's base URI.

base_uri_params
list of base *URIParameter* s for the base URI, or None.

uri_params
list of *URIParameter* s that can apply to all resources, or None.

protocols
list of str s of API-supported protocols. If not defined, is inferred by protocol from *RootNode.base_uri*.

title
str of API's title.

docs
list of *Documentation* objects, or None.

schemas
list of dict s, or None.

media_type
str of default accepted request/response media type, or None.

resource_types
list of *ResourceTypeNode* objects, or None.

traits
list of *TraitNode* objects, or None.

security_schemas
list of *SecurityScheme* objects, or None.

resources
list of *ResourceNode* objects, or None.

raml_obj
The loader.RAMLDict object.

Note: *TraitNode*, *ResourceTypeNode*, and *ResourceNode* all inherit the following *BaseNode* attributes:

---

class ramlfications.raml.**BaseNode**

name
str name of Node

raw
dict of the raw data of the Node from the RAML file/string

root
Back reference to the Node's API *RootNode* object.

headers
list of Node's *Header* objects, or None

body
list of Node's *Body* objects, or None

responses
list of Node's *Response* objects, or None

uri_params
list of Node's *URIParameter* objects, or None

base_uri_params
list of Node's base *URIParameter* objects, or None

query_params
list of Node's *QueryParameter* objects, or None

form_params
list of Node's *FormParameter* objects, or None.

media_type
str of supported request MIME media type. Defaults to *RootNode*'s media_type.

description
str description of Node.

protocols
list of str s of supported protocols. Defaults to *RootNode.protocols*.

class ramlfications.raml.**TraitNode**
Object representing a RAML Trait.

In addition to the *BaseNode*-defined attributes, TraitNode also has:

usage
Usage of trait

class ramlfications.raml.**ResourceTypeNode**
Object representing a RAML Resource Type.

In addition to the *BaseNode*-defined attributes, ResourceTypeNode also has:

display_name
str of user-friendly name of resource type, defaults to *BaseNode.name*

type
str name of inherited *ResourceTypeNode* object, or None.

method
str of supported method.

Removes the ? if present (see *optional*).

usage
str usage of resource type, or None

optional
bool resource type attributes inherited if applied resource defines method (i.e. there is a ? in resource type method).

is_
list of assigned trait names (str), or None

traits
list of assigned *TraitNode* objects, or None

secured_by
list of str s or dict s of assigned security scheme, or None.

If str, then the name of the security scheme.

If dict, the key is the name of the scheme, the values are the parameters assigned (e.g. relevant OAuth 2 scopes).

security_schemes
list of assigned parameters.SecurityScheme objects, or None.

class ramlfications.raml.**ResourceNode**

parent
parent *ResourceNode* object if any, or None.

method
str HTTP method for resource, or None.

display_name
str of user-friendly name of resource. Defaults to name.

path
str of relative path of resource.

absolute_uri
str concatenation of absolute URI of resource: *RootNode.base_uri* + *path*.

is_
list of str s or dict s of resource-assigned traits, or None.

traits
list of assigned *TraitNode* objects, or None.

type
str of the name of the assigned resource type, or None.

resource_type
The assigned *ResourceTypeNode* object from *ResourceNode.type*, or None

secured_by
list of str s or dict s of resource-assigned security schemes, or None.

If a str, the name of the security scheme.

If a dict, the key is the name of the scheme, the values are the parameters assigned (e.g. relevant OAuth 2 scopes).

security_schemes

Parameters

Note: The *URIParameter*, *QueryParameter*, *FormParameter*, and *Header* objects all share the same attributes.

class ramlfications.parameters.**URIParameter**
URI parameter with properties defined by the RAML specification's "Named Parameters" section, e.g.: /foo/{id} where id is the name of the URI parameter.

class ramlfications.parameters.**QueryParameter**
Query parameter with properties defined by the RAML specification's "Named Parameters" section, e.g. /foo/bar?baz=123 where baz is the name of the query parameter.

class ramlfications.parameters.**FormParameter**
Form parameter with properties defined by the RAML specification's "Named Parameters" section. Example:

```
curl -X POST https://api.com/foo/bar -d baz=123
```

where baz is the Form Parameter name.

class ramlfications.parameters.**Header**
Header with properties defined by the RAML spec's 'Named Parameters' section, e.g.:

```
curl -H 'X-Some-Header:  foobar' ...
```

where X-Some-Header is the Header name.

name
str of the name of parameter.

raw
dict of all raw data associated with the parameter from the RAML file/string.

description
str parameter description, or None.

display_name
str of a user-friendly name for display or documentation purposes.

If displayName is not specified in RAML data, defaults to name.

min_length
int of the parameter's minimum number of characters, or None.

Note: Only applies when the parameter's primative type is string.

max_length
int of the parameter's maximum number of characters, or None.

Note: Only applies when the parameter's primative type is string.

minimum
int of the parameter's minimum value, or None.

Note: Only applies when the parameter's primative type is integer or number.

maximum
int of the parmeter's maximum value, or None.

Note: Only applies when the parameter's primative type is integer or number.

example
Example value for parameter, or None.

Note: Attribute type of example will match that of type.

default
Default value for parameter, or `None`.

Note: Attribute type of `default` will match that of `type`.

repeat
`bool` if parameter can be repeated. Defaults to `False`.

pattern
`str` of a regular expression that parameter of type `string` must match, or `None` if not set.

enum
`list` of valid parameter values, or `None`.

Note: Only applies when parameter's primative `type` is `string`.

type
`str` representation of the primative type of parameter. Defaults to `string` if not set.

required
`bool` if parameter is required.

Note: Defaults to `True` if *URIParameter*, else defaults to `False`.

class `ramlfications.parameters.`**Body**
Body of the request/response.

mime_type
`str` of the accepted MIME media types for the body of the request/response.

raw
`dict` of all raw data associated with the `Body` from the RAML file/string

schema
`dict` of body schema definition, or `None` if not set.

Note: Can not be set if `mime_type` is `multipart/form-data` or `application/x-www-form-urlencoded`

example
`dict` of example of schema, or `None` if not set.

Note: Can not be set if `mime_type` is `multipart/form-data` or `application/x-www-form-urlencoded`

form_params
`list` of *FormParameter* objects accepted in the request body.

Note: Must be set if `mime_type` is `multipart/form-data` or `application/x-www-form-urlencoded`. Can not be used when schema and/or example is defined.

class `ramlfications.parameters.`**Response**
Expected response parameters.

code
`int` of HTTP response code.

raw
`dict` of all raw data associated with the `Response` from the RAML file/string

description
`str` of the response description, or `None`.

headers
`list` of *`Header`* objects, or `None`.

body
`list` of *`Body`* objects, or `None`.

method
`str` of HTTP request method associated with response.

class `ramlfications.parameters.`**`Documentation`**
User documentation for the API.

title
`str` title of documentation

content
`str` content of documentation

class `ramlfications.parameters.`**`SecurityScheme`**
Security scheme definition.

name
`str` name of security scheme.

raw
`dict` of all raw data associated with the `SecurityScheme` from the RAML file/string

type
`str` of type of authentication

described_by
`Header` s, *`Response`* s, *`QueryParameter`* s, etc. that is needed/can be expected when using security scheme.

description
`str` description of security scheme.

settings
`dict` of security schema-specific information

class `ramlfications.parameters.`**`Content`**(*data*)
Returns documentable content from the RAML file (e.g. Documentation content, description) in either raw or parsed form.

Parameters **`data`** (*str*) – The raw/marked up content data.

raw
Return raw Markdown/plain text written in the RAML file

html
Returns parsed Markdown into HTML

Loader

class ramlfications.loader.**RAMLLoader**
Extends YAML loader to load RAML files with `!include` tags.

load(*raml*)
Loads the desired RAML file and returns data.

Parameters **raml** – Either a string/unicode path to RAML file, a file object, or string-representation of RAML.

Returns Data from RAML file

Return type dict

Validate

Functions are used when instantiating the classes from `ramlfications.raml`.

ramlfications.validate.**root_version**(*inst*, *attr*, *value*)
Require an API Version (e.g. api.foo.com/v1).

ramlfications.validate.**root_base_uri**(*inst*, *attr*, *value*)
Require a Base URI.

ramlfications.validate.**root_base_uri_params**(*inst*, *attr*, *value*)
Require that Base URI parameters have a `default` parameter set.

ramlfications.validate.**root_uri_params**(*inst*, *attr*, *value*)
Assert that where is no `version` parameter in the regular URI parameters

ramlfications.validate.**root_protocols**(*inst*, *attr*, *value*)
Only support HTTP/S plus what is defined in user-config

ramlfications.validate.**root_title**(*inst*, *attr*, *value*)
Require a title for the defined API.

ramlfications.validate.**root_docs**(*inst*, *attr*, *value*)
Assert that if there is `documentation` defined in the root of the RAML file, that it contains a `title` and `content`.

ramlfications.validate.**root_media_type**(*inst*, *attr*, *value*)
Only support media types based on config and regex

ramlfications.validate.**assigned_traits**(*inst*, *attr*, *value*)
Assert assigned traits are defined in the RAML Root and correctly represented in the RAML.

ramlfications.validate.**assigned_res_type**(*inst*, *attr*, *value*)
Assert only one (or none) assigned resource type is defined in the RAML Root and correctly represented in the RAML.

ramlfications.validate.**header_type**(*inst*, *attr*, *value*)
Supported header type

ramlfications.validate.**body_mime_type**(*inst*, *attr*, *value*)
Supported MIME media type for request/response

ramlfications.validate.**body_schema**(*inst*, *attr*, *value*)
Assert no `schema` is defined if body as a form-related MIME media type

ramlfications.validate.**body_example**(*inst*, *attr*, *value*)
Assert no `example` is defined if body as a form-related MIME media type

ramlfications.validate.**body_form**(*inst*, *attr*, *value*)
Assert `formParameters` are defined if body has a form-related MIME type.

ramlfications.validate.**response_code**(*inst*, *attr*, *value*)
Assert a valid response code.

ramlfications.validate.**integer_number_type_parameter**(*inst*, *attr*, *value*)
Assert correct parameter attributes for `integer` or `number` primative parameter types.

ramlfications.validate.**string_type_parameter**(*inst*, *attr*, *value*)
Assert correct parameter attributes for `string` primative parameter types.

ramlfications.validate.**validate_mime_type**(*value*)
Assert a valid MIME media type for request/response body.

Tree

ramlfications.tree.**tree**(*load_obj*, *color*, *output*, *verbosity*, *validate*, *config*)
Create a tree visualization of given RAML file.

Parameters

- **load_obj** (*ramlfications.loader.RAMLDict*) – Loaded RAML File
- **color** (*str*) – `light`, `dark` or `None` (default) for the color output
- **output** (*str*) – Path to output file, if given
- **verbosity** (*str*) – Level of verbosity to print out

Returns ASCII Tree representation of API

Return type stdout to screen or given file name

Raises InvalidRamlFileError If error occured trying to validate the RAML file (see `validate.py`)

CHAPTER 4

Project Information

4.1 Changelog

4.1.1 0.1.0a1 (2015-04-18)

Initial alpha release of `ramlfications`!

4.2 License and Hall of Fame

`ramlfications` is licensed under the Apache 2.0 license. The full license text can be also found in the source code repository.

4.2.1 Credits

`ramlfications` is written and maintained by Lynn Root and thanks various contributors:

- Hynek Schlawack
- Matt Montag

4.3 How To Contribute

Every open source project lives from the generous help by contributors that sacrifice their time and `ramlfications` is no different.

To make participation as pleasant as possible, this project adheres to the Code of Conduct by the Python Software Foundation.

Here are a few hints and rules to get you started:

- Take a look at the `wishlist` for inspiration.
- Any GitHub issue that is not assigned is up for grabs.
- Add yourself to the AUTHORS.rst file in an alphabetical fashion. Every contribution is valuable and shall be credited.
- If your change is noteworthy, add an entry to the changelog.

- No contribution is too small; please submit as many fixes for typos and grammar bloopers as you can!
- *Always* add tests and docs for your code. This is a hard rule; patches with missing tests or documentation won't be merged – if a feature is not tested or documented, it doesn't exist.
- Obey PEP 8 and PEP 257.
- Write good commit messages.

Note: If you have something great but aren't sure whether it adheres – or even can adhere – to the rules above: **please submit a pull request anyway**!

In the best case, we can mold it into something, in the worst case the pull request gets politely closed. There's absolutely nothing to fear.

Thank you for considering to contribute to `ramlfications`!

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

r

Symbols

A

B

C

D

E

F

H

I

L

M

N

O

P

Q

R

S

T