
dRest Documentation

Release 0.9.6

BJ Dierkes

Sep 27, 2017

Contents

1	License	3
2	ChangeLog	5
3	Contributor Credits	7
4	Usage Documentation	9
5	API Documentation	17
6	Indices and tables	27
	Python Module Index	29

dRest is a configurable HTTP/REST client library for Python. It's goal is to make the creation of API clients dead simple, without lacking features.

Key Features:

- Light-weight API Client Library, implementing REST by default
- Native support for the Django TastyPie API Framework
- Only one external dependency on `httplib2`
- Key pieces of the library are customizable by defined handlers
- Interface definitions ensure handlers are properly implemented
- Support for Python 2.6+ and Python 3+
- 100% test coverage

Additional Links:

- RTFD: <http://drest.rtfid.org/>
- GITHUB: <http://github.com/derks/drest/>
- PYPI: <http://pypi.python.org/pypi/drest/>

Contents:

CHAPTER 1

License

Copyright (c) 2011-2012, BJ Dierkes All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of BJ Dierkes nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER 2

ChangeLog

All bugs/feature details can be found at:

<https://github.com/derks/drest/issues/XXXXXX>

Where XXXXXX is the 'Issue #' referenced below. Additionally, this change log is available online at:

<http://drest.readthedocs.org/en/latest/changelog.html>

0.9.6 - Mar 23, 2012

Bug Fixes:

- [Issue #9](#) - GET params incorrectly stored as permanent `_extra_url_params`.

Feature Enhancements:

- [Issue #4](#) - Better support for nested resource names.
- [Issue #5](#), [Issue #8](#) - Request object is now exposed publicly.
- [Issue #6](#) - Add capability to suppress final forward-slash
- [Issue #7](#) - Cache http object for improved performance.

Incompatible Changes:

- `api._request` is now `api.request`. `api.request` (old function) is now `api.make_request()`
- Lots of code refactoring.. numerous minor changes may break compatibility if using backend functions, but not likely if accessing the high level api functions.
- Every request now returns a `drest.response.ResponseHandler` object rather than a (response, data) tuple.

0.9.4 - Feb 16, 2012

Bug Fixes:

- [Issue #3](#) - TypeError: object.__init__() takes no parameters

Feature Enhancements:

- Improved test suite, now powered by Django TastyPie!
- Added support for Basic HTTP Auth.

Incompatible Changes:

- `drest.api.API.auth()` now implements Basic HTTP Auth by default rather than just appending the user/password/etc to the URL.

0.9.2 - Feb 01, 2012

- Initial Beta release. Future versions will detail bugs/features/etc.

CHAPTER 3

Contributor Credits

- Andrew Alcock

Usage Documentation

Contents:

Installation

The following outlines installation of dRest. It is recommended to work out of a [VirtualENV](#) for development, which is reference throughout this documentation. VirtualENV is easily installed on most platforms either with ‘easy_install’ or ‘pip’ or via your OS distributions packaging system (yum, apt, brew, etc).

Creating a Virtual Environment

```
$ virtualenv --no-site-packages ~/env/drest/  
$ source ~/env/drest/bin/activate
```

When installing drest, ensure that your development environment is active by sourcing the activate script (as seen above).

Installing Development Version From Git

```
(drest) $ git clone git://github.com/derks/drest.git  
(drest) $ cd src/drest/  
(drest) $ python setup.py install
```

To run tests, do the following from the ‘root’ directory:

```
(drest) $ pip install nose
(drest) $ python setup.py nosetests
```

Installing Stable Versions From PyPi

```
(drest) $ pip install drest
```

Running Unit Tests

Unit tests should be run to ensure dRest is completely functional on your system:

```
(drest) $ ./utils/run_tests.sh
```

For Python 3 testing, you will need to run ‘drest.mockapi’ manually via a separate virtualenv setup for Python 2.6+ (in a separate terminal), and then run the test suite with the option ‘–without-mockapi’:

Terminal 1:

```
$ virtualenv-2.7 ~/env/drest-py27/
$ source ~/env/drest-py27/bin/activate
(drest-py27) $ cd src/drest.mockapi/mockapi
(drest-py27) $ python manage.py testserver
```

Terminal 2:

```
$ virtualenv-3.2 ~/env/drest-py32/
$ source ~/env/drest-py32/bin/activate
(drest-py32) $ ./utils/run_tests.sh --without-mockapi
```

Quickstart Guide

A REST Client Example

Note that the following is all fictitious data. What is received from and sent to an API is unique to every API. Do not copy and paste these examples.

Connecting with an API

```
import drest
api = drest.API('http://localhost:8000/api/v1/')
```

Authentication

By default, `drest.api.API.auth()` implements HTTP Basic Authentication. This is generally overridden however by specific API's that subclass from `api.API()`.

```
api.auth('john.doe', 'my_password')
```

Note that authentication may not be necessary for your use case, or for read-only API's.

Making Requests

Requests can be made openly by specifying the method (GET, PUT, POST, DELETE, ...), as well as the path (after the baseurl).

```
# GET http://localhost:8000/api/v1/users/1/
response = api.make_request('GET', '/users/1/')
```

Additionally, you can add a resource which makes access to the API more native and programatic.

```
# Add a basic resource (assumes path='/users/')
api.add_resource('users')

# A list of available resources is available at:
api.resources

# GET http://localhost:8000/api/v1/users/
response = api.users.get()

# GET http://localhost:8000/api/v1/users/1/
response = api.users.get(1)
```

Creating a resource only requires a dictionary of 'parameters' passed to the resource:

```
user_data = dict(
    username='john.doe',
    password='oober-secure-password',
    first_name='John',
    last_name='Doe',
)

# POST http://localhost:8000/api/v1/users/
response = api.users.post(user_data)
```

Updating a resource is as easy as requesting data for it, modifying it, and sending it back

```
response = api.users.get(1)
updated_data = response.data.copy()
updated_data['first_name'] = 'John'
updated_data['last_name'] = 'Doe'

# PUT http://localhost:8000/api/v1/users/1/
response = api.users.put(1, updated_data)
```

Deleting a resource simply requires the primary key:

```
# DELETE http://localhost:8000/api/v1/users/1/
response = api.users.delete(1)
```

Working With Return Data

Every call to an API by default returns a `drest.response.ResponseHandler` object. The two most useful members of this object are:

- `response.status` (http status code)
- `response.data` (the data returned by the api)

If a serialization handler is used, then `response.data` will be the unserialized form (Python dict).

The Response Object

```
response = api.users.get()
response.status # 200
response.data # dict
```

Developers can base conditions on the status of the response (or other fields):

```
response = api.users.get()
if response.status != 200:
    print 'Uhoh.... we didn't get a good response.'
```

The data returned from a request is the data returned by the API. This is generally JSON, YAML, XML, etc... however if a Serialization handler is enabled, this will be a python dictionary. See [drest.serialization](#).

`response.data`:

```
{
  u'meta':
    {
      u'previous': None,
      u'total_count': 3,
      u'offset': 0,
      u'limit': 20,
      u'next':
        None
    },
  u'objects':
    [
      {
        u'username': u'john.doe',
        u'first_name': u'John',
        u'last_name': u'Doe',
        u'resource_pk': 2,
        u'last_login': u'2012-01-26T01:21:20',
        u'resource_uri': u'/api/v1/users/2/',
        u'id': u'2',
        u'date_joined': u'2008-09-04T14:25:29'
      }
    ]
}
```

The above is fictitious data returned from a TastyPie API. What is returned by an API is unique to that API therefore you should expect the 'data' to be different than the above.

Connecting Over SSL

Though this is documented elsewhere, it is a pretty common question. Often times API services are SSL enabled (over <https://>) but do not possess a valid or active SSL certificate. Anytime an API service has an invalid, or usually self-signed certificate, you will receive an SSL error similar to:

```
[Errno 1] _ssl.c:503: error:14090086:SSL routines:SSL3_GET_SERVER_
↪CERTIFICATE:certificate verify failed
```

In order to work around such situations, simply pass the following to your api:

```
api = drest.API('https://example.com/api/v1/', ignore_ssl_validation=True)
```

Working With Django TastyPie

dRest includes an API specifically built for the [Django TastyPie](#) API framework. It handles auto-detection of resources, and their schema, as well as other features to support the interface including getting resource data by 'resource_uri'.

API Reference

- `drest.api.TastyPieAPI`
- `drest.request.TastyPieRequestHandler`
- `drest.resource.TastyPieResourceHandler`

Customizing dRest

Every piece of dRest is completely customizable by way of 'handlers'.

API Reference:

- `drest.api`
- `drest.resource`
- `drest.request`
- `drest.response`
- `drest.serialization`

Example

The following is just a quick glance at what it would look to chain together a custom Serialization Handler, Request Handler, Resource Handler, and finally a custom API client object. This is not meant to be comprehensive by any means. In the real world, you will need to read the source code documentation listed above and determine a) what you need to customize, and b) what functionality you need to maintain.

```
import drest

class MySerializationHandler(drest.serialization.SerializationHandler):
    def serialize(self, data_dict):
```

```
    # do something to serialize data dictionary
    pass

    def deserialize(self, serialized_data):
        # do something to deserialize data
        pass

class MyResponseHandler(drest.response.ResponseHandler)
    def __init__(self, status, data, **kw):
        super(MyResponseHandler, self).__init__(status, data, **kw)
        # do something to customize the response handler

class MyRequestHandler(drest.request.RequestHandler):
    class Meta:
        serialization_handler = MySerializationHandler
        response_handler = MyResponseHandler

    def handle_response(self, response):
        # do something to wrape every response
        pass

class MyResourceHandler(drest.resource.ResourceHandler):
    class Meta:
        request_handler = MyRequestHandler

    def some_custom_function(self, params={}):
        # do some kind of custom api call
        return self.request('GET', '/users/some_custom_function', params)

class MyAPI(drest.api.API):
    class Meta:
        baseurl = 'http://example.com/api/v1/'
        resource_handler = MyResourceHandler
        request_handler = MyRequestHandler

    def auth(self, *args, **kw):
        # do something to customize authentication
        pass

api = MyAPI()

# Add resources
api.add_resource('users')
api.add_resource('projects')

# GET http://example.com/api/v1/users/
api.users.get()

# GET http://example.com/api/v1/users/133/
api.users.get(133)

# PUT http://example.com/api/v1/users/133/
api.users.put(133, data_dict)

# POST http://example.com/api/v1/users/
api.users.post(data_dict)

# DELETE http://example.com/api/v1/users/133/
```

```
api.users.delete(133)

# GET http://example.com/api/v1/users/some_custom_function/
api.users.some_custom_function()
```

Note that the id '133' above is the fictitious id of a user resource.

Debugging Requests

Often times in development, making calls to an API can be obscure and difficult to work with especially when receiving 500 Internal Server Errors with no idea what happens. In browser development, most frameworks like Django or the like provide some sort of debugging interface allowing developers to analyze tracebacks, and what not. Not so much when developing command line apps or similar.

Enabling Debug Output

In order to enable DEBUG output for every request, simply set the 'DREST_DEBUG' environment variable to 1:

```
$ set DREST_DEBUG=1

$ python test.py
DREST_DEBUG: method=POST url=http://localhost:8000/api/v0/systems/ params={} headers={
  'Content-Type': 'application/json', 'Authorization': 'ApiKey john.doe:XXXXXXXXXXXXX'}
```

In the above, test.py just made a simple api.system.post() call which triggered DREST_DEBUG output. In the output you have access to a number of things:

method This is the method used to make the request.

url The full url path of the request

params Any parameters passed with the request

headers Any headers passed with the request

Once done debugging, just disable DREST_DEBUG:

```
$ unset DREST_DEBUG
```

Viewing Upstream Tracebacks

If the error is happening server side, like a 500 Internal Server Error, you will likely receive a traceback in the return content (at least during development). This of course depends on the API you are developing against, however the following is common practice in development:

```
try:
    response = api.my_resource.get()
except drest.exc.dRestRequestError as e:
    print e.response.status
    print e.response.data
```

The above gives you the response object, as well as the content (data)... this is useful because the exception is triggered in drest code and not your own (therefore bringing the response object back down the stack where you can use it).

drest . api

dRest core API connection library.

class `drest . api . API` (*baseurl=None, **kw*)

The API class acts as a high level ‘wrapper’ around multiple lower level handlers. Most of the meta arguments are optionally passed to one or more handlers upon instantiation. All handler classes must be passed *un-instantiated*.

Arguments:

baseurl Translated to `self.baseurl` (for convenience).

Optional Arguments and Meta:

debug Boolean. Toggle debug console output. Default: False.

baseurl The base url to the API endpoint.

request_handler The Request Handler class that performs the actual HTTP (or other) requests.
Default: `drest.request.RequestHandler`.

resource_handler The Resource Handler class that is used when `api.add_resource` is called. Default: `drest.resource.ResourceHandler`.

response_handler An un-instantiated Response Handler class used to return responses to the caller.
Default: `drest.response.ResponseHandler`.

serialization_handler An un-instantiated Serialization Handler class used to serialize/deserialize data. Default: `drest.serialization.JsonSerializationHandler`.

ignore_ssl_validation Boolean. Whether or not to ignore ssl validation errors. Default: False

serialize Boolean. Whether or not to serialize data before sending requests. Default: False.

deserialize Boolean. Whether or not to deserialize data before returning the Response object. Default: True.

trailing_slash Boolean. Whether or not to append a trailing slash to the request url. Default: True.

extra_headers A dictionary of key value pairs that are added to the HTTP headers of *every* request. Passed to request_handler.add_header().

extra_params A dictionary of key value pairs that are added to the POST, or 'payload' data sent with *every* request. Passed to request_handler.add_param().

extra_url_params A dictionary of key value pairs that are added to the GET/URL parameters of *every* request. Passed to request_handler.add_extra_url_param().

Usage

```
import drest

# Create a generic client api object
api = drest.API('http://localhost:8000/api/v1/')

# Or something more customized:
api = drest.API(
    baseurl='http://localhost:8000/api/v1/',
    trailing_slash=False,
    ignore_ssl_validation=True,
)

# Or even more so:
class MyAPI(drest.API):
    class Meta:
        baseurl = 'http://localhost:8000/api/v1/'
        extra_headers = dict(MyKey='Some Value For Key')
        extra_params = dict(some_param='some_value')
        request_handler = MyCustomRequestHandler
api = MyAPI()

# By default, the API support HTTP Basic Auth with username/password.
api.auth('john.doe', 'password')

# Make calls openly
response = api.make_request('GET', '/users/1/')

# Or attach a resource
api.add_resource('users')

# Get available resources
api.resources

# Get all objects of a resource
response = api.users.get()

# Get a single resource with primary key '1'
response = api.users.get(1)

# Update a resource with primary key '1'
response = api.users.get(1)
updated_data = response.data.copy()
updated_data['first_name'] = 'John'
updated_data['last_name'] = 'Doe'

response = api.users.put(data['id'], updated_data)
```

```
# Create a resource
user_data = dict(
    username='john.doe',
    password='oober-secure-password',
    first_name='John',
    last_name='Doe',
)
response = api.users.post(user_data)

# Delete a resource with primary key '1'
response = api.users.delete(1)
```

add_resource (*name*, *resource_handler=None*, *path=None*)

Add a resource handler to the api object.

Required Arguments:

name The name of the resource. This is generally the basic name of the resource on the API. For example `/api/v0/users/` would likely be called `'users'` and will be accessible as `'api.users'` from which additional calls can be made. For example `'api.users.get()'`.

Optional Arguments:

resource_handler The resource handler class to use. Defaults to `self._meta.resource_handler`.

path The path to the resource on the API (after the base url). Defaults to `'/<name>/'`.

Nested Resources:

It is possible to attach resources in a 'nested' fashion. For example passing a name of `'my.nested.users'` would be accessible as `api.my.nested.users.get()`.

Usage:

```
api.add_resource('users')
response = api.users.get()

# Or for nested resources
api.add_resource('my.nested.users', path='/users/')
response = api.my.nested.users.get()
```

auth (*user*, *password*, ***kw*)

This authentication mechanism implements HTTP Basic Authentication.

Required Arguments:

user The API username.

password The password of that user.

class `drest.api.TastyPieAPI` (**args*, ***kw*)

This class implements an API client, specifically tailored for interfacing with [TastyPie](#).

Optional / Meta Arguments:

auth_mech The auth mechanism to use. One of `['basic', 'api_key']`. Default: `'api_key'`.

auto_detect_resources Boolean. Whether or not to auto detect, and add resource objects to the api. Default: `True`.

Authentication Mechanisms

Currently the only supported authentication mechanism are:

- ApiKeyAuthentication
- BasicAuthentication

Usage

Please note that the following example use fictitious resource data. What is returned, and sent to the API is unique to the API itself. Please do not copy and paste any of the following directly without modifying the request parameters per your use case.

Create the client object, and authenticate with a user/api_key pair by default:

```
import drest
api = drest.api.TastyPieAPI('http://localhost:8000/api/v0/')
api.auth('john.doe', '34547a497326dde80bc8f8bcee43e3d1b5f24cc9')
```

OR authenticate against HTTP Basic Auth:

```
import drest
api = drest.api.TastyPieAPI('http://localhost:8000/api/v0/',
                           auth_mech='basic')
api.auth('john.doe', 'my_password')
```

As drest auto-detects TastyPie resources, you can view those at:

```
api.resources
```

And access their schema:

```
api.users.schema
```

As well as make the usual calls such as:

```
api.users.get()
api.users.get(<pk>)
api.users.put(<pk>, data_dict)
api.users.post(data_dict)
api.users.delete(<pk>)
```

What about filtering? (these depend on how the [API is configured](#)):

```
api.users.get(params=dict(username='admin'))
api.users.get(params=dict(username__icontains='admin'))
...
```

See `drest.api.API` for more standard usage examples.

auth (*args, **kw)

Authenticate the request, determined by Meta.auth_mech. Arguments and Keyword arguments are just passed to the auth_mech function.

find_resources ()

Find available resources, and add them via add_resource().

drest.exc

exception drest.exc.dRestAPIError (msg)
dRest API Errors.

exception `drest.exc.dRestError(msg)`
Generic dRest Errors.

exception `drest.exc.dRestInterfaceError(msg)`
dRest Interface Errors.

exception `drest.exc.dRestRequestError(msg, response)`
dRest Request Errors.

exception `drest.exc.dRestResourceError(msg)`
dRest Resource Errors.

drest.interface

class `drest.interface.Attribute(description)`
Defines an Interface attribute.

Usage:

```
from drest import interface

class MyInterface(interface.Interface):
    my_attribute = interface.Attribute("A description of my_attribute.")
```

class `drest.interface.Interface`
This is an abstract base class that all interface classes should subclass from.

`drest.interface.validate(interface, obj, members, metas=[])`
A wrapper to validate interfaces.

Required Arguments:

- interface** The interface class to validate against
- obj** The object to validate.
- members** A list of object members that must exist.

Optional Arguments:

- metas** A list of meta parameters that must exist.

drest.meta

dRest core meta functionality. Barrowed from <http://slumber.in/>.

class `drest.meta.Meta(**kw)`
Model that acts as a container class for a meta attributes for a larger class. It stuffs any kwarg it gets in it's init as an attribute of itself.

class `drest.meta.MetaMixin(*args, **kw)`
Mixin that provides the Meta class support to add settings to instances of slumber objects. Meta settings cannot start with a `_`.

drest.request

class drest.request.IRequest

This class defines the Request Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

All implementations must provide sane ‘default’ functionality when instantiated with no arguments. Meaning, it can and should accept optional parameters that alter how it functions, but can not require any parameters.

Implementations do *not* subclass from interfaces.

add_header (*key*, *value*)

Add extra headers to pass along with *every* request.

Required Arguments:

key The key of the header to add.

value The value of the header to add.

add_param (*key*, *value*)

Add extra parameters to pass along with *every* request. These are passed with the request ‘payload’ (serialized if a serialization handler is enabled). With GET requests they are appended to the URL.

Required Arguments:

key The key of the parameter to add.

value The value of the parameter to add.

add_url_param (*key*, *value*)

Similar to ‘add_params’, however this function adds extra parameters to the url for *every* request. These are *not* passed with the request ‘payload’ (serialized if a serialization handler is enabled) except for GET requests.

Required Arguments:

key The key of the parameter to add.

value The value of the parameter to add.

handle_response (*response_object*)

Called after the request is made. This is a convenient place for developers to handle what happens during every request per their application needs.

Required Arguments:

response_object The response object created by the request.

make_request (*method*, *path*, *params*={}, *headers*={})

Make a request with the upstream API.

Required Arguments:

method The HTTP method to request as. I.e. ['GET', 'POST', 'PUT', 'DELETE', '...'].

path The of the request url *after* the baseurl.

Optional Arguments:

params Parameters to pass with the request. These will be serialized if configured to serialize.

headers Headers to pass to the request.

class `drest.request.RequestHandler` (***kw*)

Generic class that handles HTTP requests. Uses the Json Serialization handler by default, but only ‘deserializes’ response content.

Optional Arguments / Meta:

debug Boolean. Toggle debug console output. Default: False.

ignore_ssl_validation Boolean. Whether or not to ignore ssl validation errors. Default: False

response_handler An un-instantiated Response Handler class used to return responses to the caller. Default: `drest.response.ResponseHandler`.

serialization_handler An un-instantiated Serialization Handler class used to serialize/deserialize data. Default: `drest.serialization.JsonSerializationHandler`.

serialize Boolean. Whether or not to serialize data before sending requests. Default: False.

deserialize Boolean. Whether or not to deserialize data before returning the Response object. Default: True.

trailing_slash Boolean. Whether or not to append a trailing slash to the request url. Default: True.

add_header (*key, value*)

Adds a key/value to `self._extra_headers`, which is sent with every request.

Required Arguments:

key The key of the parameter.

value The value of ‘key’.

add_param (*key, value*)

Adds a key/value to `self._extra_params`, which is sent with every request.

Required Arguments:

key The key of the parameter.

value The value of ‘key’.

add_url_param (*key, value*)

Adds a key/value to `self._extra_url_params`, which is sent with every request (in the URL).

Required Arguments:

key The key of the parameter.

value The value of ‘key’.

handle_response (*response_object*)

A simple wrapper to handle the response. By default raises `exc.dRestRequestError` if the response code is within 400-499, or 500. Must return the original, or modified, response object.

Required Arguments:

response_object The response object created by the request.

make_request (*method, url, params={}, headers={}*)

Make a call to a resource based on path, and parameters.

Required Arguments:

method One of HEAD, GET, POST, PUT, PATCH, DELETE, etc.

url The full url of the request (without any parameters). Any params (with GET method) and `self.extra_url_params` will be added to this url.

Optional Arguments:

params Dictionary of additional (one-time) keyword arguments for the request.

headers Dictionary of additional (one-time) headers of the request.

set_auth_credentials (*user, password*)

Set the authentication user and password that will be used for HTTP Basic and Digest Authentication.

Required Arguments:

user The authentication username.

password That user's password.

class `drest.request.TastyPieRequestHandler` (***kw*)

This class implements the IRequest interface, specifically tailored for interfacing with TastyPie.

See `drest.request.RequestHandler` for Meta options and usage.

`drest.request.validate` (*obj*)

Validates a handler implementation against the IRequest interface.

drest.resource

class `drest.resource.IResource`

This class defines the Resource Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

All implementations must provide sane 'default' functionality when instantiated with no arguments. Meaning, it can and should accept optional parameters that alter how it functions, but can not require any parameters.

Implementations do *not* subclass from interfaces.

class `drest.resource.RESTResourceHandler` (*api_obj, name, path, **kw*)

This class implements the IResource interface, specifically for interacting with REST-like resources. It provides convenient functions that wrap around the typical GET, PUT, POST, DELETE actions.

Optional Arguments / Meta:

api_obj The api (parent) object that this resource is being attached to.

name The name of the resource on the API.

path The path to the resource (after api.baseurl).

Usage:

```
import drest

class MyAPI(drest.api.API):
    class Meta:
        resource_handler = drest.resource.RESTResourceHandler
    ...
```

create (*params={}*)

A synonym for self.post().

delete (*resource_id, params={}*)

Delete resource record.

Required Arguments:

resource_id The resource id

Optional Arguments:

params Some resource might allow additional parameters. For example, the user resource has a 'rdikwid' (really delete I know what I'm doing) option which causes a user to *really* be deleted (normally deletion only sets the status to 'Deleted').

filter (*params*)

Give the ability to alter params before sending the request.

Required Arguments:

params The list of params that will be passed to the endpoint.

get (*resource_id=None, params={}*)

Get all records for a resource, or a single resource record.

Optional Arguments:

resource_id The resource id (may also be a label in some environments).

params Additional request parameters to pass along.

post (*params={}*)

Create a new resource.

Required Arguments:

params A dictionary of parameters (different for every resource).

put (*resource_id, params={}*)

Update an existing resource.

Required Arguments:

resource_id The id of the resource to update.

params A dictionary of parameters (different for every resource).

update (*resource_id, params={}*)

A synonym for self.put().

class drest.resource.**ResourceHandler** (*api_obj, name, path, **kw*)

This class acts as a base class that other resource handler should subclass from.

class drest.resource.**TastyPieResourceHandler** (*api_obj, name, path, **kw*)

This class implements the IResource interface, specifically tailored for interfacing with [TastyPie](#).

get_by_uri (*resource_uri, params={}*)

A wrapper around self.get() that accepts a TastyPie 'resource_uri' rather than a 'pk' (primary key).

Required Arguments:

resource_uri The Resource URI to GET.

Optional Arguments

params All additional keyword arguments are passed as extra request parameters.

Usage

```
import drest
api = drest.api.TastyPieAPI('http://localhost:8000/api/v0/')
api.auth(user='john.doe', api_key='34547a497326dde80bc8f8bcee43e3d1b5f24cc9')
response = api.users.get_by_uri('/api/v1/users/234/')

```

schema

Returns the resources schema.

`drest.resource.validate(obj)`

Validates a handler implementation against the IResource interface.

drest.serialization

class `drest.serialization.ISerialization`

This class defines the Serialization Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

All implementations must provide sane ‘default’ functionality when instantiated with no arguments. Meaning, it can and should accept optional parameters that alter how it functions, but can not require any parameters.

Implementations do *not* subclass from interfaces.

deserialize()

Load a serialized string and return a dictionary of key/value pairs.

Required Arguments:

serialized_data A string of serialized data.

Returns: dict

get_headers()

Return a dictionary of additional headers to include in requests.

serialize()

Dump a dictionary of values from a serialized string.

Required Arguments:

data_dict A data dictionary to serialize.

Returns: string

class `drest.serialization.JsonSerializationHandler(**kw)`

This handler implements the ISerialization interface using the standard json library.

class `drest.serialization.SerializationHandler(**kw)`

Generic Serialization Handler. Should be used to subclass from.

`drest.serialization.validate(obj)`

Validates a handler implementation against the ISerialize interface.

CHAPTER 6

Indices and tables

- `genindex`
- `modindex`
- `search`

d

- `drest.api`, [17](#)
- `drest.exc`, [20](#)
- `drest.interface`, [21](#)
- `drest.meta`, [21](#)
- `drest.request`, [22](#)
- `drest.resource`, [24](#)
- `drest.serialization`, [26](#)

A

`add_header()` (drest.request.IRequest method), 22
`add_header()` (drest.request.RequestHandler method), 23
`add_param()` (drest.request.IRequest method), 22
`add_param()` (drest.request.RequestHandler method), 23
`add_resource()` (drest.api.API method), 19
`add_url_param()` (drest.request.IRequest method), 22
`add_url_param()` (drest.request.RequestHandler method), 23
API (class in drest.api), 17
Attribute (class in drest.interface), 21
`auth()` (drest.api.API method), 19
`auth()` (drest.api.TastyPieAPI method), 20

C

`create()` (drest.resource.RESTResourceHandler method), 24

D

`delete()` (drest.resource.RESTResourceHandler method), 24
`deserialize()` (drest.serialization.ISerialization method), 26
drest.api (module), 17
drest.exc (module), 20
drest.interface (module), 21
drest.meta (module), 21
drest.request (module), 22
drest.resource (module), 24
drest.serialization (module), 26
dRestAPIError, 20
dRestError, 20
dRestInterfaceError, 21
dRestRequestError, 21
dRestResourceError, 21

F

`filter()` (drest.resource.RESTResourceHandler method), 25

`find_resources()` (drest.api.TastyPieAPI method), 20

G

`get()` (drest.resource.RESTResourceHandler method), 25
`get_by_uri()` (drest.resource.TastyPieResourceHandler method), 25
`get_headers()` (drest.serialization.ISerialization method), 26

H

`handle_response()` (drest.request.IRequest method), 22
`handle_response()` (drest.request.RequestHandler method), 23

I

Interface (class in drest.interface), 21
IRequest (class in drest.request), 22
IResource (class in drest.resource), 24
ISerialization (class in drest.serialization), 26

J

JsonSerializationHandler (class in drest.serialization), 26

M

`make_request()` (drest.request.IRequest method), 22
`make_request()` (drest.request.RequestHandler method), 23
Meta (class in drest.meta), 21
MetaMixin (class in drest.meta), 21

P

`post()` (drest.resource.RESTResourceHandler method), 25
`put()` (drest.resource.RESTResourceHandler method), 25

R

RequestHandler (class in drest.request), 22
ResourceHandler (class in drest.resource), 25
RESTResourceHandler (class in drest.resource), 24

S

schema (drest.resource.TastyPieResourceHandler attribute), [25](#)
SerializationHandler (class in drest.serialization), [26](#)
serialize() (drest.serialization.ISerialization method), [26](#)
set_auth_credentials() (drest.request.RequestHandler method), [24](#)

T

TastyPieAPI (class in drest.api), [19](#)
TastyPieRequestHandler (class in drest.request), [24](#)
TastyPieResourceHandler (class in drest.resource), [25](#)

U

update() (drest.resource.RESTResourceHandler method), [25](#)

V

validate() (in module drest.interface), [21](#)
validate() (in module drest.request), [24](#)
validate() (in module drest.resource), [26](#)
validate() (in module drest.serialization), [26](#)