Flask-RESTful Documentation

0.2.1

Kyle Conroy, Ryan Horn, Frank Stratton

2015 09 21

Contents

Flask-RESTful is an extension for Flask that adds support for quickly building REST APIs. It is a lightweight abstraction that works with your existing ORM/libraries. Flask-RESTful encourages best practices with minimal setup. If you are familiar with Flask, Flask-RESTful should be easy to pick up.

CHAPTER 1

User's Guide

This part of the documentation will show you how to get started in using Flask-RESTful with Flask.

1.1 Installation

Install Flask-RESTful with `pip`

```
pip install flask-restful
```

The development version can be downloaded from its page at GitHub.

```
git clone https://github.com/flask-restful/flask-restful.git
cd flask-restful
python setup.py develop
```

Flask-RESTful has the following dependencies (which will be automatically installed if you use `pip`):

- Flask version 0.8 or greater

Flask-RESTful requires Python version 2.6, 2.7, 3.3, or 3.4.

1.2 Quickstart

It's time to write your first REST API. This guide assumes you have a working understanding of Flask, and that you have already installed both Flask and Flask-RESTful. If not, then follow the steps in the *Installation* section.

1.2.1 A Minimal API

A minimal Flask-RESTful API looks like this:

```
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        return {'hello': 'world'}
```

```
api.add_resource(HelloWorld, '/')

if __name__ == '__main__':
    app.run(debug=True)
```

Save this as api.py and run it using your Python interpreter. Note that we've enabled Flask debugging mode to provide code reloading and better error messages.

```
$ python api.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

: Debug mode should never be used in a production environment!

Now open up a new prompt to test out your API using curl

```
$ curl http://127.0.0.1:5000/
{"hello": "world"}
```

1.2.2 Resourceful Routing

The main building block provided by Flask-RESTful are resources. Resources are built on top of Flask pluggable views, giving you easy access to multiple HTTP methods just by defining methods on your resource. A basic CRUD resource for a todo application (of course) looks like this:

```
from flask import Flask, request
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

todos = {}

class TodoSimple(Resource):
    def get(self, todo_id):
        return {todo_id: todos[todo_id]}

    def put(self, todo_id):
        todos[todo_id] = request.form['data']
        return {todo_id: todos[todo_id]}

api.add_resource(TodoSimple, '/<string:todo_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

You can try it like this:

```
$ curl http://localhost:5000/todo1 -d "data=Remember the milk" -X PUT
{"todo1": "Remember the milk"}
$ curl http://localhost:5000/todo1
{"todo1": "Remember the milk"}
$ curl http://localhost:5000/todo2 -d "data=Change my brakepads" -X PUT
{"todo2": "Change my brakepads"}
$ curl http://localhost:5000/todo2
{"todo2": "Change my brakepads"}
```

Or from python if you have the `requests` library installed:

```
>>> from requests import put, get
>>> put('http://localhost:5000/todo1', data={'data': 'Remember the milk'}).json()
{u'todo1': u'Remember the milk'}
>>> get('http://localhost:5000/todo1').json()
{u'todo1': u'Remember the milk'}
>>> put('http://localhost:5000/todo2', data={'data': 'Change my brakepads'}).json()
{u'todo2': u'Change my brakepads'}
>>> get('http://localhost:5000/todo2').json()
{u'todo2': u'Change my brakepads'}
```

Flask-RESTful understands multiple kinds of return values from view methods. Similar to Flask, you can return any iterable and it will be converted into a response, including raw Flask response objects. Flask-RESTful also support setting the response code and response headers using multiple return values, as shown below:

```
class Todo1(Resource):
    def get(self):
        # Default to 200 OK
        return {'task': 'Hello world'}

class Todo2(Resource):
    def get(self):
        # Set the response code to 201
        return {'task': 'Hello world'}, 201

class Todo3(Resource):
    def get(self):
        # Set the response code to 201 and return custom headers
        return {'task': 'Hello world'}, 201, {'Etag': 'some-opaque-string'}
```

1.2.3 Endpoints

Many times in an API, your resource will have multiple URLs. You can pass multiple URLs to the `add_resource()` method on the *Api* object. Each one will be routed to your `Resource`

```
api.add_resource(HelloWorld,
    '/',
    '/hello')
```

You can also match parts of the path as variables to your resource methods.

```
api.add_resource(Todo,
    '/todo/<int:todo_id>', endpoint='todo_ep')
```

: If a request does not match any of your application's endpoints, Flask-RESTful will return a 404 error message with suggestions of other endpoints that closely match the requested endpoint. This can be disabled by setting `ERROR_404_HELP` to `False` in your application config.

1.2.4 Argument Parsing

While Flask provides easy access to request data (i.e. querystring or POST form encoded data), it's still a pain to validate form data. Flask-RESTful has built-in support for request data validation using a library similar to argparse.

```
from flask_restful import reqparse

parser = reqparse.RequestParser()
parser.add_argument('rate', type=int, help='Rate to charge for this resource')
args = parser.parse_args()
```

: Unlike the argparse module, `reqparse.RequestParser.parse_args()` returns a Python dictionary instead of a custom data structure.

Using the *reqparse* module also gives you sane error messages for free. If an argument fails to pass validation, Flask-RESTful will respond with a 400 Bad Request and a response highlighting the error.

```
$ curl -d 'rate=foo' http://127.0.0.1:5000/todos
{'status': 400, 'message': 'foo cannot be converted to int'}
```

The `inputs` module provides a number of included common conversion functions such as `inputs.date()` and `inputs.url()`.

Calling `parse_args` with `strict=True` ensures that an error is thrown if the request includes arguments your parser does not define.

args = parser.parse_args(strict=True)

1.2.5 Data Formatting

By default, all fields in your return iterable will be rendered as-is. While this works great when you're just dealing with Python data structures, it can become very frustrating when working with objects. To solve this problem, Flask-RESTful provides the `fields` module and the `marshal_with()` decorator. Similar to the Django ORM and WTForm, you use the `fields` module to describe the structure of your response.

```
from collections import OrderedDict
from flask_restful import fields, marshal_with

resource_fields = {
    'task':   fields.String,
    'uri':    fields.Url('todo_ep')
}

class TodoDao(object):
    def __init__(self, todo_id, task):
        self.todo_id = todo_id
        self.task = task

        # This field will not be sent in the response
        self.status = 'active'

class Todo(Resource):
    @marshal_with(resource_fields)
    def get(self, **kwargs):
        return TodoDao(todo_id='my_todo', task='Remember the milk')
```

The above example takes a python object and prepares it to be serialized. The `marshal_with()` decorator will apply the transformation described by `resource_fields`. The only field extracted from the object is `task`. The `fields.Url` field is a special field that takes an endpoint name and generates a URL for that endpoint in the response. Many of the field types you need are already included. See the `fields` guide for a complete list.

1.2.6 Full Example

Save this example in api.py

```
from flask import Flask
from flask_restful import reqparse, abort, Api, Resource

app = Flask(__name__)
api = Api(app)

TODOS = {
    'todo1': {'task': 'build an API'},
    'todo2': {'task': '?????'},
    'todo3': {'task': 'profit!'},
}


def abort_if_todo_doesnt_exist(todo_id):
    if todo_id not in TODOS:
        abort(404, message="Todo {} doesn't exist".format(todo_id))

parser = reqparse.RequestParser()
parser.add_argument('task')


# Todo
# shows a single todo item and lets you delete a todo item
class Todo(Resource):
    def get(self, todo_id):
        abort_if_todo_doesnt_exist(todo_id)
        return TODOS[todo_id]

    def delete(self, todo_id):
        abort_if_todo_doesnt_exist(todo_id)
        del TODOS[todo_id]
        return '', 204

    def put(self, todo_id):
        args = parser.parse_args()
        task = {'task': args['task']}
        TODOS[todo_id] = task
        return task, 201


# TodoList
# shows a list of all todos, and lets you POST to add new tasks
class TodoList(Resource):
    def get(self):
        return TODOS

    def post(self):
        args = parser.parse_args()
        todo_id = int(max(TODOS.keys()).lstrip('todo')) + 1
        todo_id = 'todo%i' % todo_id
        TODOS[todo_id] = {'task': args['task']}
        return TODOS[todo_id], 201

##
## Actually setup the Api resource routing here
```

```
##
api.add_resource(TodoList, '/todos')
api.add_resource(Todo, '/todos/<todo_id>')


if __name__ == '__main__':
    app.run(debug=True)
```

Example usage

```
$ python api.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

GET the list

```
$ curl http://localhost:5000/todos
{"todo1": {"task": "build an API"}, "todo3": {"task": "profit!"}, "todo2": {"task": "?????"}}
```

GET a single task

```
$ curl http://localhost:5000/todos/todo3
{"task": "profit!"}
```

DELETE a task

```
$ curl http://localhost:5000/todos/todo2 -X DELETE -v

> DELETE /todos/todo2 HTTP/1.1
> User-Agent: curl/7.19.7 (universal-apple-darwin10.0) libcurl/7.19.7 OpenSSL/0.9.8l zlib/1.2.3
> Host: localhost:5000
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 204 NO CONTENT
< Content-Type: application/json
< Content-Length: 0
< Server: Werkzeug/0.8.3 Python/2.7.2
< Date: Mon, 01 Oct 2012 22:10:32 GMT
```

Add a new task

```
$ curl http://localhost:5000/todos -d "task=something new" -X POST -v

> POST /todos HTTP/1.1
> User-Agent: curl/7.19.7 (universal-apple-darwin10.0) libcurl/7.19.7 OpenSSL/0.9.8l zlib/1.2.3
> Host: localhost:5000
> Accept: */*
> Content-Length: 18
> Content-Type: application/x-www-form-urlencoded
>
* HTTP 1.0, assume close after body
< HTTP/1.0 201 CREATED
< Content-Type: application/json
< Content-Length: 25
< Server: Werkzeug/0.8.3 Python/2.7.2
< Date: Mon, 01 Oct 2012 22:12:58 GMT
<
* Closing connection #0
{"task": "something new"}
```

Update a task

```
$ curl http://localhost:5000/todos/todo3 -d "task=something different" -X PUT -v

> PUT /todos/todo3 HTTP/1.1
> Host: localhost:5000
> Accept: */*
> Content-Length: 20
> Content-Type: application/x-www-form-urlencoded
>
* HTTP 1.0, assume close after body
< HTTP/1.0 201 CREATED
< Content-Type: application/json
< Content-Length: 27
< Server: Werkzeug/0.8.3 Python/2.7.3
< Date: Mon, 01 Oct 2012 22:13:00 GMT
<
* Closing connection #0
{"task": "something different"}
```

1.3 Request Parsing

: The whole request parser part of Flask-RESTful is slated for removal and will be replaced by documentation on how to integrate with other packages that do the input/output stuff better (such as marshmallow). This means that it will be maintained until 2.0 but consider it deprecated. Don't worry, if you have code using that now and wish to continue doing so, it's not going to go away any time too soon.

Flask-RESTful's request parsing interface, *reqparse*, is modeled after the argparse interface. It's designed to provide simple and uniform access to any variable on the `flask.request` object in Flask.

1.3.1 Basic Arguments

Here's a simple example of the request parser. It looks for two arguments in the `flask.Request.values` dict: an integer and a string

```
from flask_restful import reqparse

parser = reqparse.RequestParser()
parser.add_argument('rate', type=int, help='Rate cannot be converted')
parser.add_argument('name')
args = parser.parse_args()
```

: The default argument type is a unicode string. This will be `str` in python3 and `unicode` in python2.

If you specify the `help` value, it will be rendered as the error message when a type error is raised while parsing it. If you do not specify a help message, the default behavior is to return the message from the type error itself.

By default, arguments are **not** required. Also, arguments supplied in the request that are not part of the RequestParser will be ignored.

Also note: Arguments declared in your request parser but not set in the request itself will default to `None`.

1.3.2 Required Arguments

To require a value be passed for an argument, just add `required=True` to the call to `add_argument()`.

```
parser.add_argument('name', required=True,
help="Name cannot be blank!")
```

1.3.3 Multiple Values & Lists

If you want to accept multiple values for a key as a list, you can pass `action='append'`

```
parser.add_argument('name', action='append')
```

This will let you make queries like

```
curl http://api.example.com -d "name=bob" -d "name=sue" -d "name=joe"
```

And your args will look like this

```
args = parser.parse_args()
args['name']    # ['bob', 'sue', 'joe']
```

1.3.4 Other Destinations

If for some reason you'd like your argument stored under a different name once it's parsed, you can use the `dest` keyword argument.

```
parser.add_argument('name', dest='public_name')

args = parser.parse_args()
args['public_name']
```

1.3.5 Argument Locations

By default, the `RequestParser` tries to parse values from `flask.Request.values`, and `flask.Request.json`.

Use the `location` argument to `add_argument()` to specify alternate locations to pull the values from. Any variable on the `flask.Request` can be used. For example:

```
# Look only in the POST body
parser.add_argument('name', type=int, location='form')

# Look only in the querystring
parser.add_argument('PageSize', type=int, location='args')

# From the request headers
parser.add_argument('User-Agent', location='headers')

# From http cookies
parser.add_argument('session_id', location='cookies')

# From file uploads
parser.add_argument('picture', type=werkzeug.datastructures.FileStorage, location='files')
```

: Only use `type=list` when `location='json'`. See this issue for more details

1.3.6 Multiple Locations

Multiple argument locations can be specified by passing a list to `location`:

```
parser.add_argument('text', location=['headers', 'values'])
```

The last `location` listed takes precedence in the result set.

1.3.7 Parser Inheritance

Often you will make a different parser for each resource you write. The problem with this is if parsers have arguments in common. Instead of rewriting arguments you can write a parent parser containing all the shared arguments and then extend the parser with `copy()`. You can also overwrite any argument in the parent with `replace_argument()`, or remove it completely with `remove_argument()`. For example:

```
from flask_restful import reqparse

parser = reqparse.RequestParser()
parser.add_argument('foo', type=int)

parser_copy = parser.copy()
parser_copy.add_argument('bar', type=int)

# parser_copy has both 'foo' and 'bar'

parser_copy.replace_argument('foo', required=True, location='json')
# 'foo' is now a required str located in json, not an int as defined
#  by original parser

parser_copy.remove_argument('foo')
# parser_copy no longer has 'foo' argument
```

1.3.8 Error Handling

The default way errors are handled by the RequestParser is to abort on the first error that occurred. This can be beneficial when you have arguments that might take some time to process. However, often it is nice to have the errors bundled together and sent back to the client all at once. This behavior can be specified either at the Flask application level or on the specific RequestParser instance. To invoke a RequestParser with the bundling errors option, pass in the argument `bundle_errors`. For example

```
from flask_restful import reqparse

parser = reqparse.RequestParser(bundle_errors=True)
parser.add_argument('foo', type=int, required=True)
parser.add_argument('bar', type=int, required=True)

# If a request comes in not containing both 'foo' and 'bar', the error that
# will come back will look something like this.

{
    "message":  {
```

```
        "foo": "foo error message",
        "bar": "bar error message"
    }
}

# The default behavior would only return the first error

parser = RequestParser()
parser.add_argument('foo', type=int, required=True)
parser.add_argument('bar', type=int, required=True)

{
    "message":  {
        "foo": "foo error message"
    }
}
```

The application configuration key is "BUNDLE_ERRORS". For example

```
from flask import Flask

app = Flask(__name__)
app.config['BUNDLE_ERRORS'] = True
```

: `BUNDLE_ERRORS` is a global setting that overrides the `bundle_errors` option in individual `RequestParser` instances.

1.4 Output Fields

Flask-RESTful provides an easy way to control what data you actually render in your response. With the `fields` module, you can use whatever objects (ORM models/custom classes/etc.) you want in your resource. `fields` also lets you format and filter the response so you don't have to worry about exposing internal data structures.

It's also very clear when looking at your code what data will be rendered and how it will be formatted.

1.4.1 Basic Usage

You can define a dict or OrderedDict of fields whose keys are names of attributes or keys on the object to render, and whose values are a class that will format & return the value for that field. This example has three fields: two are `String` and one is a `DateTime`, formatted as an RFC 822 date string (ISO 8601 is supported as well)

```
from flask_restful import Resource, fields, marshal_with

resource_fields = {
    'name': fields.String,
    'address': fields.String,
    'date_updated': fields.DateTime(dt_format='rfc822'),
}

class Todo(Resource):
    @marshal_with(resource_fields, envelope='resource')
    def get(self, **kwargs):
        return db_get_todo()  # Some function that queries the db
```

This example assumes that you have a custom database object (`todo`) that has attributes `name`, `address`, and `date_updated`. Any additional attributes on the object are considered private and won't be rendered in the output. An optional `envelope` keyword argument is specified to wrap the resulting output.

The decorator `marshal_with` is what actually takes your data object and applies the field filtering. The marshalling can work on single objects, dicts, or lists of objects.

: `marshal_with` is a convenience decorator, that is functionally equivalent to

```
class Todo(Resource):
    def get(self, **kwargs):
        return marshal(db_get_todo(), resource_fields), 200
```

This explicit expression can be used to return HTTP status codes other than 200 along with a successful response (see `abort()` for errors).

1.4.2 Renaming Attributes

Often times your public facing field name is different from your internal field name. To configure this mapping, use the `attribute` keyword argument.

```
fields = {
    'name': fields.String(attribute='private_name'),
    'address': fields.String,
}
```

A lambda (or any callable) can also be specified as the `attribute`

```
fields = {
    'name': fields.String(attribute=lambda x: x._private_name),
    'address': fields.String,
}
```

Nested properties can also be accessed with `attribute`:

fields = { 'name': fields.String(attribute='people_list.0.person_dictionary.name'), 'address': fields.String,

}

1.4.3 Default Values

If for some reason your data object doesn't have an attribute in your fields list, you can specify a default value to return instead of `None`.

```
fields = {
    'name': fields.String(default='Anonymous User'),
    'address': fields.String,
}
```

1.4.4 Custom Fields & Multiple Values

Sometimes you have your own custom formatting needs. You can subclass the :class'fields.Raw' class and implement the format function. This is especially useful when an attribute stores multiple pieces of information. e.g. a bit-field

whose individual bits represent distinct values. You can use fields to multiplex a single attribute to multiple output values.

This example assumes that bit 1 in the `flags` attribute signifies a "Normal" or "Urgent" item, and bit 2 signifies "Read" or "Unread". These items might be easy to store in a bitfield, but for a human readable output it's nice to convert them to seperate string fields.

```
class UrgentItem(fields.Raw):
    def format(self, value):
        return "Urgent" if value & 0x01 else "Normal"

class UnreadItem(fields.Raw):
    def format(self, value):
        return "Unread" if value & 0x02 else "Read"

fields = {
    'name': fields.String,
    'priority': UrgentItem(attribute='flags'),
    'status': UnreadItem(attribute='flags'),
}
```

1.4.5 Url & Other Concrete Fields

Flask-RESTful includes a special field, `fields.Url`, that synthesizes a uri for the resource that's being requested. This is also a good example of how to add data to your response that's not actually present on your data object.:

```
class RandomNumber(fields.Raw):
    def output(self, key, obj):
        return random.random()

fields = {
    'name': fields.String,
    # todo_resource is the endpoint name when you called api.add_resource()
    'uri': fields.Url('todo_resource'),
    'random': RandomNumber,
}
```

By default `fields.Url` returns a relative uri. To generate an absolute uri that includes the scheme, hostname and port, pass the keyword argument `absolute=True` in the field declaration. To override the default scheme, pass the `scheme` keyword argument:

```
fields = {
    'uri': fields.Url('todo_resource', absolute=True)
    'https_uri': fields.Url('todo_resource', absolute=True, scheme='https')
}
```

1.4.6 Complex Structures

You can have a flat structure that `marshal()` will transform to a nested structure

```
>>> from flask_restful import fields, marshal
>>> import json
>>>
>>> resource_fields = {'name': fields.String}
>>> resource_fields['address'] = {}
>>> resource_fields['address']['line 1'] = fields.String(attribute='addr1')
```

```
>>> resource_fields['address']['line 2'] = fields.String(attribute='addr2')
>>> resource_fields['address']['city'] = fields.String
>>> resource_fields['address']['state'] = fields.String
>>> resource_fields['address']['zip'] = fields.String
>>> data = {'name': 'bob', 'addr1': '123 fake street', 'addr2': '', 'city': 'New York', 'state': 'NY
>>> json.dumps(marshal(data, resource_fields))
'{"name": "bob", "address": {"line 1": "123 fake street", "line 2": "", "state": "NY", "zip": "10468'
```

: The address field doesn't actually exist on the data object, but any of the sub-fields can access attributes directly from the object as if they were not nested.

1.4.7 List Field

You can also unmarshal fields as lists

```
>>> from flask_restful import fields, marshal
>>> import json
>>>
>>> resource_fields = {'name': fields.String, 'first_names': fields.List(fields.String)}
>>> data = {'name': 'Bougnazal', 'first_names' : ['Emile', 'Raoul']}
>>> json.dumps(marshal(data, resource_fields))
>>> '{"first_names": ["Emile", "Raoul"], "name": "Bougnazal"}'
```

1.4.8 Advanced : Nested Field

While nesting fields using dicts can turn a flat data object into a nested response, you can use `Nested` to unmarshal nested data structures and render them appropriately.

```
>>> from flask_restful import fields, marshal
>>> import json
>>>
>>> address_fields = {}
>>> address_fields['line 1'] = fields.String(attribute='addr1')
>>> address_fields['line 2'] = fields.String(attribute='addr2')
>>> address_fields['city'] = fields.String(attribute='city')
>>> address_fields['state'] = fields.String(attribute='state')
>>> address_fields['zip'] = fields.String(attribute='zip')
>>>
>>> resource_fields = {}
>>> resource_fields['name'] = fields.String
>>> resource_fields['billing_address'] = fields.Nested(address_fields)
>>> resource_fields['shipping_address'] = fields.Nested(address_fields)
>>> address1 = {'addr1': '123 fake street', 'city': 'New York', 'state': 'NY', 'zip': '10468'}
>>> address2 = {'addr1': '555 nowhere', 'city': 'New York', 'state': 'NY', 'zip': '10468'}
>>> data = { 'name': 'bob', 'billing_address': address1, 'shipping_address': address2}
>>>
>>> json.dumps(marshal_with(data, resource_fields))
'{"billing_address": {"line 1": "123 fake street", "line 2": null, "state": "NY", "zip": "10468", "c
```

This example uses two `Nested` fields. The `Nested` constructor takes a dict of fields to render as sub-fields.input The important difference between the `Nested` constructor and nested dicts (previous example), is the context for attributes. In this example, `billing_address` is a complex object that has its own fields and the context passed to the nested field is the sub-object instead of the original `data` object. In other words: `data.billing_address.addr1` is

in scope here, whereas in the previous example `data.addr1` was the location attribute. Remember: `Nested` and `List` objects create a new scope for attributes.

Use `Nested` with `List` to marshal lists of more complex objects:

```
user_fields = {
    'id': fields.Integer,
    'name': fields.String,
}

user_list_fields = {
    fields.List(fields.Nested(user_fields)),
}
```

1.5 Extending Flask-RESTful

We realize that everyone has different needs in a REST framework. Flask-RESTful tries to be as flexible as possible, but sometimes you might find that the builtin functionality is not enough to meet your needs. Flask-RESTful has a few different extension points that can help in that case.

1.5.1 Content Negotiation

Out of the box, Flask-RESTful is only configured to support JSON. We made this decision to give API maintainers full control of over API format support; so a year down the road you don't have to support people using the CSV representation of your API you didn't even know existed. To add additional mediatypes to your API, you'll need to declare your supported representations on the `Api` object.

```
app = Flask(__name__)
api = restful.Api(app)

@api.representation('application/json')
def output_json(data, code, headers=None):
    resp = make_response(json.dumps(data), code)
    resp.headers.extend(headers or {})
    return resp
```

These representation functions must return a Flask `Response` object.

: Flask-RESTful uses the `json` module from the Python standard library instead of `flask.json` because the Flask JSON serializer includes serializtion capabilities which are not in the JSON spec. If your application needs these customizations, you can replace the default JSON representation with one using the Flask JSON module as described above.

It is possible to configure how the default Flask-RESTful JSON representation will format JSON by providing a `RESTFUL_JSON` attribute on the application configuration. This setting is a dictionary with keys that correspond to the keyword arguments of `json.dumps()`.

```
class MyConfig(object):
    RESTFUL_JSON = {'separators': (', ', ': '),
                    'indent': 2,
                    'cls': MyCustomEncoder}
```

: If the application is running in debug mode (`app.debug = True`) and either `sort_keys` or `indent` are not

declared in the `RESTFUL_JSON` configuration setting, Flask-RESTful will provide defaults of `True` and `4` respectively.

1.5.2 Custom Fields & Inputs

One of the most common additions to Flask-RESTful is to define custom types or fields based on your own data types.

Fields

Custom output fields let you perform your own output formatting without having to modify your internal objects directly. All you have to do is subclass `Raw` and implement the `format()` method:

```
class AllCapsString(fields.Raw):
    def format(self, value):
        return value.upper()


# example usage
fields = {
    'name': fields.String,
    'all_caps_name': AllCapsString(attribute=name),
}
```

Inputs

For parsing arguments, you might want to perform custom validation. Creating your own input types lets you extend request parsing with ease.

```
def odd_number(value):
    if value % 2 == 0:
        raise ValueError("Value is not odd")

    return value
```

The request parser will also give you access to the name of the argument for cases where you want to reference the name in the error message.

```
def odd_number(value, name):
    if value % 2 == 0:
        raise ValueError("The parameter '{}' is not odd. You gave us the value: {}".format(name, valu

    return value
```

You can also convert public parameter values to internal representations:

```
# maps the strings to their internal integer representation
# 'init' => 0
# 'in-progress' => 1
# 'completed' => 2

def task_status(value):
    statuses = [u"init", u"in-progress", u"completed"]
    return statuses.index(value)
```

Then you can use these custom input types in your `RequestParser`:

```
parser = reqparse.RequestParser()
parser.add_argument('OddNumber', type=odd_number)
parser.add_argument('Status', type=task_status)
args = parser.parse_args()
```

1.5.3 Response Formats

To support other representations (xml, csv, html), you can use the `representation()` decorator. You need to have a reference to your API.

```
api = restful.Api(app)

@api.representation('text/csv')
def output_csv(data, code, headers=None):
    pass
    # implement csv output!
```

These output functions take three parameters, `data`, `code`, and `headers`

`data` is the object you return from your resource method, code is the HTTP status code that it expects, and headers are any HTTP headers to set in the response. Your output function should return a `flask.Response` object.

```
def output_json(data, code, headers=None):
    """Makes a Flask response with a JSON encoded body"""
    resp = make_response(json.dumps(data), code)
    resp.headers.extend(headers or {})
    return resp
```

Another way to accomplish this is to subclass the `Api` class and provide your own output functions.

```
class Api(restful.Api):
    def __init__(self, *args, **kwargs):
        super(Api, self).__init__(*args, **kwargs)
        self.representations = {
            'application/xml': output_xml,
            'text/html': output_html,
            'text/csv': output_csv,
            'application/json': output_json,
        }
```

1.5.4 Resource Method Decorators

There is a property on the `Resource` class called `method_decorators`. You can subclass the Resource and add your own decorators that will be added to all `method` functions in resource. For instance, if you want to build custom authentication into every request.

```
def authenticate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if not getattr(func, 'authenticated', True):
            return func(*args, **kwargs)

        acct = basic_authentication()  # custom account lookup function

        if acct:
            return func(*args, **kwargs)
```

```
        restful.abort(401)
    return wrapper


class Resource(restful.Resource):
    method_decorators = [authenticate]   # applies to all inherited resources
```

Since Flask-RESTful Resources are actually Flask view objects, you can also use standard flask view decorators.

1.5.5 Custom Error Handlers

Error handling is a tricky problem. Your Flask application may be wearing multiple hats, yet you want to handle all Flask-RESTful errors with the correct content type and error syntax as your 200-level requests.

Flask-RESTful will call the `handle_error()` function on any 400 or 500 error that happens on a Flask-RESTful route, and leave other routes alone. You may want your app to return an error message with the correct media type on 404 Not Found errors; in which case, use the *catch_all_404s* parameter of the `Api` constructor.

```
app = Flask(__name__)
api = flask_restful.Api(app, catch_all_404s=True)
```

Then Flask-RESTful will handle 404s in addition to errors on its own routes.

Sometimes you want to do something special when an error occurs - log to a file, send an email, etc. Use the `got_request_exception()` method to attach custom error handlers to an exception.

```
def log_exception(sender, exception, **extra):
    """ Log an exception to our logging framework """
    sender.logger.debug('Got exception during processing: %s', exception)

from flask import got_request_exception
got_request_exception.connect(log_exception, app)
```

Define Custom Error Messages

You may want to return a specific message and/or status code when certain errors are encountered during a request. You can tell Flask-RESTful how you want to handle each error/exception so you won't have to fill your API code with try/except blocks.

```
errors = {
    'UserAlreadyExistsError': {
        'message': "A user with that username already exists.",
        'status': 409,
    },
    'ResourceDoesNotExist': {
        'message': "A resource with that ID no longer exists.",
        'status': 410,
        'extra': "Any extra information you want.",
    },
}
```

Including the *'status'* key will set the Response's status code. If not specified it will default to 500.

Once your `errors` dictionary is defined, simply pass it to the `Api` constructor.

```
app = Flask(__name__)
api = flask_restful.Api(app, errors=errors)
```

1.6 Intermediate Usage

This page covers building a slightly more complex Flask-RESTful app that will cover out some best practices when setting up a real-world Flask-RESTful-based API. The *Quickstart* section is great for getting started with your first Flask-RESTful app, so if you're new to Flask-RESTful you'd be better off checking that out first.

1.6.1 Project Structure

There are many different ways to organize your Flask-RESTful app, but here we'll describe one that scales pretty well with larger apps and maintains a nice level organization.

The basic idea is to split your app into three main parts: the routes, the resources, and any common infrastructure.

Here's an example directory structure:

```
myapi/
    __init__.py
    app.py          # this file contains your app and routes
    resources/
        __init__.py
        foo.py      # contains logic for /Foo
        bar.py      # contains logic for /Bar
    common/
        __init__.py
        util.py     # just some common infrastructure
```

The common directory would probably just contain a set of helper functions to fulfill common needs across your application. It could also contain, for example, any custom input/output types your resources need to get the job done.

In the resource files, you just have your resource objects. So here's what `foo.py` might look like:

```
from flask_restful import Resource

class Foo(Resource):
    def get(self):
        pass
    def post(self):
        pass
```

The key to this setup lies in `app.py`:

```
from flask import Flask
from flask_restful import Api
from myapi.resources.foo import Foo
from myapi.resources.bar import Bar
from myapi.resources.baz import Baz

app = Flask(__name__)
api = Api(app)

api.add_resource(Foo, '/Foo', '/Foo/<str:id>')
api.add_resource(Bar, '/Bar', '/Bar/<str:id>')
api.add_resource(Baz, '/Baz', '/Baz/<str:id>')
```

As you can imagine with a particularly large or complex API, this file ends up being very valuable as a comprehensive list of all the routes and resources in your API. You would also use this file to set up any config values (`before_request()`, `after_request()`). Basically, this file configures your entire API.

The things in the common directory are just things you'd want to support your resource modules.

1.6.2 Use With Blueprints

See Modular Applications with Blueprints in the Flask documentation for what blueprints are and why you should use them. Here's an example of how to link an `Api` up to a `Blueprint`.

```
from flask import Flask, Blueprint
from flask_restful import Api, Resource, url_for

app = Flask(__name__)
api_bp = Blueprint('api', __name__)
api = Api(api_bp)

class TodoItem(Resource):
    def get(self, id):
        return {'task': 'Say "Hello, World!"'}

api.add_resource(TodoItem, '/todos/<int:id>')
app.register_blueprint(api_bp)
```

: Calling `Api.init_app()` is not required here because registering the blueprint with the app takes care of setting up the routing for the application.

1.6.3 Full Parameter Parsing Example

Elsewhere in the documentation, we've described how to use the reqparse example in detail. Here we'll set up a resource with multiple input parameters that exercise a larger amount of options. We'll define a resource named "User".

```
from flask_restful import fields, marshal_with, reqparse, Resource

def email(email_str):
    """Return email_str if valid, raise an exception in other case."""
    if valid_email(email_str):
        return email_str
    else:
        raise ValueError('{} is not a valid email'.format(email_str))

post_parser = reqparse.RequestParser()
post_parser.add_argument(
    'username', dest='username',
    location='form', equired=True,
    help='The user\'s username',
)
post_parser.add_argument(
    'email', dest='email',
    type=email, location='form',
    required=True, help='The user\'s email',
)
post_parser.add_argument(
```

```
    'user_priority', dest='user_priority',
    type=int, location='form',
    default=1, choices=range(5), help='The user\'s priority',
)

user_fields = {
    'id': fields.Integer,
    'username': fields.String,
    'email': fields.String,
    'user_priority': fields.Integer,
    'custom_greeting': fields.FormattedString('Hey there {username}!'),
    'date_created': fields.DateTime,
    'date_updated': fields.DateTime,
    'links': fields.Nested({
        'friends': fields.Url('user_friends'),
        'posts': fields.Url('user_posts'),
    }),
}

class User(Resource):

    @marshal_with(user_fields)
    def post(self):
        args = post_parser.parse_args()
        user = create_user(args.username, args.email, args.user_priority)
        return user

    @marshal_with(user_fields)
    def get(self, id):
        args = post_parser.parse_args()
        user = fetch_user(id)
        return user
```

As you can see, we create a `post_parser` specifically to handle the parsing of arguments provided on POST. Let's step through the definition of each argument.

```
post_parser.add_argument(
    'username', dest='username',
    location='form', required=True,
    help='The user\'s username',
)
```

The `username` field is the most normal out of all of them. It takes a string from the POST body and converts it to a string type. This argument is required (`required=True`), which means that if it isn't provided, Flask-RESTful will automatically return a 400 with a message along the lines of 'the username field is required'.

```
post_parser.add_argument(
    'email', dest='email',
    type=email, location='form',
    required=True, help='The user\'s email',
)
```

The `email` field has a custom type of `email`. A few lines earlier we defined an `email` function that takes a string and returns it if the type is valid, else it raises an exception, exclaiming that the email type was invalid.

```
post_parser.add_argument(
    'user_priority', dest='user_priority',
    type=int, location='form',
    default=1, choices=range(5), help='The user\'s priority',
```

```
)
```

The `user_priority` type takes advantage of the `choices` argument. This means that if the provided *user_priority* value doesn't fall in the range specified by the `choices` argument (in this case `[1, 2, 3, 4]`), Flask-RESTful will automatically respond with a 400 and a descriptive error message.

That covers the inputs. We also defined some interesting field types in the `user_fields` dictionary to showcase a couple of the more exotic types.

```
user_fields = {
    'id': fields.Integer,
    'username': fields.String,
    'email': fields.String,
    'user_priority': fields.Integer,
    'custom_greeting': fields.FormattedString('Hey there {username}!'),
    'date_created': fields.DateTime,
    'date_updated': fields.DateTime,
    'links': fields.Nested({
        'friends': fields.Url('user_friends', absolute=True),
        'posts': fields.Url('user_friends', absolute=True),
    }),
}
```

First up, there's `fields.FormattedString`.

```
'custom_greeting': fields.FormattedString('Hey there {username}!'),
```

This field is primarily used to interpolate values from the response into other values. In this instance, `custom_greeting` will always contain the value returned from the `username` field.

Next up, check out `fields.Nested`.

```
'links': fields.Nested({
    'friends': fields.Url('user_friends', absolute=True),
    'posts': fields.Url('user_posts', absolute=True),
}),
```

This field is used to create a sub-object in the response. In this case, we want to create a `links` sub-object to contain urls of related objects. Note that we passed *fields.Nested* another dict which is built in such a way that it would be an acceptable argument to `marshal()` by itself.

Finally, we used the `fields.Url` field type.

```
'friends': fields.Url('user_friends', absolute=True),
'posts': fields.Url('user_friends', absolute=True),
```

It takes as its first parameter the name of the endpoint associated with the urls of the objects in the `links` sub-object. Passing `absolute=True` ensures that the generated urls will have the hostname included.

1.6.4 Passing Constructor Parameters Into Resources

Your `Resource` implementation may require outside dependencies. Those dependencies are best passed-in through the constructor to loosely couple each other. The `Api.add_resource()` method has two keyword arguments: `resource_class_args` and `resource_class_kwargs`. Their values will be forwarded and passed into your Resource implementation's constructor.

So you could have a `Resource`:

```
from flask_restful import Resource

class TodoNext(Resource):
    def __init__(**kwargs):
        # smart_engine is a black box dependency
        self.smart_engine = kwargs['smart_engine']

    def get(self):
        return self.smart_engine.next_todo()
```

You can inject the required dependency into TodoNext like so:

```
smart_engine = SmartEngine()

api.add_resource(TodoNext, '/next',
    resource_class_kwargs={ 'smart_engine': smart_engine })
```

Same idea applies for forwarding *args*.

CHAPTER 2

API Reference

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

2.1 API Docs

2.1.1 Api

2.1.2 ReqParse

2.1.3 Fields

2.1.4 Inputs

CHAPTER 3

Additional Notes

See Flask's license for legal information governing this project.

3.1 Running the Tests

A `Makefile` is included to take care of setting up a virtualenv for running tests. All you need to do is run:

```
$ make test
```

To change the Python version used to run the tests (default is Python 2.7), change the `PYTHON_MAJOR` and `PYTHON_MINOR` variables at the top of the `Makefile`.

You can run on all supported versions with:

```
$ make test-all
```

Individual tests can be run using using a command with the format:

```
nosetests <filename>:ClassName.func_name
```

Example:

```
$ source env/bin/activate
$ nosetests tests/test_reqparse.py:ReqParseTestCase.test_parse_choices_insensitive
```

Alternately, if you push changes to your fork on Github, Travis will run the tests for your branch automatically.

A Tox config file is also provided so you can test against multiple python versions locally (2.6, 2.7, 3.3, and 3.4)

```
$ tox
```

f

r

F

R