Flask-RESTX Documentation

Release 0.2.0

Axel Haustant

Mar 22, 2020

Contents

Flask-RESTX is an extension for Flask that adds support for quickly building REST APIs. Flask-RESTX encourages best practices with minimal setup. If you are familiar with Flask, Flask-RESTX should be easy to pick up. It provides a coherent collection of decorators and tools to describe your API and expose its documentation properly (using Swagger).

CHAPTER 1

Compatibility

flask-restx requires Python 2.7+.

CHAPTER 2

Installation

You can install flask-restx with pip:

```
$ pip install flask-restx
```

or with easy_install:

```
$ easy_install flask-restx
```

CHAPTER 3

Documentation

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

3.1 Installation

Install Flask-RESTX with `pip`:

```
pip install flask-restx
```

The development version can be downloaded from GitHub.

```
git clone https://github.com/python-restx/flask-restx.git
cd flask-restx
pip install -e .[dev,test]
```

Flask-RESTX requires Python version 2.7, 3.3, 3.4 or 3.5. It's also working with PyPy and PyPy3.

3.2 Quick start

This guide assumes you have a working understanding of Flask, and that you have already installed both Flask and Flask-RESTX. If not, then follow the steps in the *Installation* section.

3.2.1 Initialization

As every other extension, you can initialize it with an application object:

```
from flask import Flask
from flask_restx import Api
```

(continues on next page)

(continued from previous page)

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

or lazily with the factory pattern:

```
from flask import Flask
from flask_restx import Api

api = Api()

app = Flask(__name__)
api.init_app(app)
```

3.2.2 A Minimal API

A minimal Flask-RESTX API looks like this:

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

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

@api.route('/hello')
class HelloWorld(Resource):
    def get(self):
        return {'hello': 'world'}

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
```

Warning: 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
{"hello": "world"}
```

You can also use the automatic documentation on you API root (by default). In this case: http://127.0.0.1:5000/. See *Swagger UI* for a complete documentation on the automatic documentation.

3.2.3 Resourceful Routing

The main building block provided by Flask-RESTX 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_restx import Resource, Api

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

todos = {}

@api.route('/<string:todo_id>')
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]}

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-RESTX 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-RESTX 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
```

(continues on next page)

(continued from previous page)

```
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'}
```

3.2.4 Endpoints

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

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

# or

@api.route('/hello', '/world')
class HelloWorld(Resource):
    pass
```

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')

# or

@api.route('/todo/<int:todo_id>', endpoint='todo_ep')
class HelloWorld(Resource):
    pass
```

Note: If a request does not match any of your application's endpoints, Flask-RESTX 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.

3.2.5 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-RESTX has built-in support for request data validation using a library similar to `argparse`.

```
from flask_restx import reqparse

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

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

Using the *RequestParser* class also gives you sane error messages for free. If an argument fails to pass validation, Flask-RESTX 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 *`date()`* and *`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)
```

3.2.6 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-RESTX 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 flask import Flask
from flask_restx import fields, Api, Resource

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

model = api.model('Model', {
    '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'

@api.route('/todo')
class Todo(Resource):
    @api.marshal_with(model)
    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 `model`. 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. Using the `marshal_with()` decorator also document the output in the swagger specifications. Many of the field types you need are already included. See the `fields` guide for a complete list.

Order Preservation

By default, fields order is not preserved as this have a performance drop effect. If you still require fields order preservation, you can pass a `ordered=True` parameter to some classes or function to force order preservation:

- globally on *`Api`*: `api = Api(ordered=True)`
- globally on *`Namespace`*: `ns = Namespace(ordered=True)`

- locally on *marshal()*: `return marshal(data, fields, ordered=True)`

3.2.7 Full example

See the *Full example* section for fully functional example.

3.3 Response marshalling

Flask-RESTX provides an easy way to control what data you actually render in your response or expect as in input payload. 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.

3.3.1 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 ISO 8601 datetime string (RFC 822 is supported as well):

```
from flask_restx import Resource, fields

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

@api.route('/todo')
class Todo(Resource):
    @api.marshal_with(model, 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.

Note: *marshal_with()* is a convenience decorator, that is functionally equivalent to:

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

The `@api.marshal_with` decorator add the swagger documentation ability.

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

3.3.2 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.

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

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

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

Nested properties can also be accessed with `attribute`:

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

3.3.3 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`.

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

3.3.4 Custom Fields & Multiple Values

Sometimes you have your own custom formatting needs. You can subclass the *`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 separate 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"

model = {
```

(continues on next page)

(continued from previous page)

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

3.3.5 Url & Other Concrete Fields

Flask-RESTX 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()

model = {
    'name': fields.String,
    # todo_resource is the endpoint name when you called api.route()
    '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:

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

3.3.6 Complex Structures

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

```
>>> from flask_restx 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', 'zip': '10468'}
>>> json.dumps(marshal(data, resource_fields))
'{"name": "bob", "address": {"line 1": "123 fake street", "line 2": "", "state": "NY",
↪ "zip": "10468", "city": "New York"}}'
```

Note: 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.

3.3.7 List Field

You can also unmarshal fields as lists

```
>>> from flask_restx 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"}'
```

3.3.8 Wildcard Field

If you don't know the name(s) of the field(s) you want to unmarshall, you can use *Wildcard*

```
>>> from flask_restx import fields, marshal
>>> import json
>>>
>>> wild = fields.Wildcard(fields.String)
>>> wildcard_fields = {'*': wild}
>>> data = {'John': 12, 'bob': 42, 'Jane': '68'}
>>> json.dumps(marshal(data, wildcard_fields))
>>> '{"Jane": "68", "bob": "42", "John": "12"}'
```

The name you give to your *Wildcard* acts as a real glob as shown bellow

```
>>> from flask_restx import fields, marshal
>>> import json
>>>
>>> wild = fields.Wildcard(fields.String)
>>> wildcard_fields = {'j*': wild}
>>> data = {'John': 12, 'bob': 42, 'Jane': '68'}
>>> json.dumps(marshal(data, wildcard_fields))
>>> '{"Jane": "68", "John": "12"}'
```

Note: It is important you define your *Wildcard* **outside** your model (ie. you **cannot** use it like this: `res_fields = {'*': fields.Wildcard(fields.String)}`) because it has to be stateful to keep a track of what fields it has already treated.

Note: The glob is not a regex, it can only treat simple wildcards like '*' or '?'.

In order to avoid unexpected behavior, when mixing *Wildcard* with other fields, you may want to use an `OrderedDict` and use the *Wildcard* as the last field

```
>>> from flask_restx import fields, marshal
>>> import json
>>>
>>> wild = fields.Wildcard(fields.Integer)
>>> # you can use it in api.model like this:
>>> # some_fields = api.model('MyModel', {'zoro': fields.String, '*': wild})
```

(continues on next page)

(continued from previous page)

```
>>>
>>> data = {'John': 12, 'bob': 42, 'Jane': '68', 'zoro': 72}
>>> json.dumps(marshal(data, mod))
>>> '{"zoro": "72", "Jane": 68, "bob": 42, "John": 12}'
```

3.3.9 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_restx 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(data, resource_fields))
'{"billing_address": {"line 1": "123 fake street", "line 2": null, "state": "NY", "zip
↪": "10468", "city": "New York"}, "name": "bob", "shipping_address": {"line 1": "555␣
↪nowhere", "line 2": null, "state": "NY", "zip": "10468", "city": "New York"}}'
```

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.

By default when the sub-object is *None*, an object with default values for the nested fields will be generated instead of *null*. This can be modified by passing the *allow_null* parameter, see the *Nested* constructor for more details.

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

```
user_fields = api.model('User', {
    'id': fields.Integer,
    'name': fields.String,
})

user_list_fields = api.model('UserList', {
    'users': fields.List(fields.Nested(user_fields)),
})
```

3.3.10 The `api.model()` factory

The *`model()`* factory allows you to instantiate and register models to your `API` or *`Namespace`*.

```
my_fields = api.model('MyModel', {
    'name': fields.String,
    'age': fields.Integer(min=0)
})

# Equivalent to
my_fields = Model('MyModel', {
    'name': fields.String,
    'age': fields.Integer(min=0)
})
api.models[my_fields.name] = my_fields
```

Duplicating with `clone`

The `Model.clone()` method allows you to instantiate an augmented model. It saves you duplicating all fields.

```
parent = Model('Parent', {
    'name': fields.String
})

child = parent.clone('Child', {
    'age': fields.Integer
})
```

The `Api/Namespace.clone` also register it on the API.

```
parent = api.model('Parent', {
    'name': fields.String
})

child = api.clone('Child', parent, {
    'age': fields.Integer
})
```

Polymorphism with `api.inherit`

The `Model.inherit()` method allows to extend a model in the "Swagger way" and to start handling polymorphism.

```
parent = api.model('Parent', {
    'name': fields.String,
    'class': fields.String(discriminator=True)
})

child = api.inherit('Child', parent, {
    'extra': fields.String
})
```

The `Api/Namespace.clone` will register both the parent and the child in the Swagger models definitions.

```
parent = Model('Parent', {
    'name': fields.String,
    'class': fields.String(discriminator=True)
})

child = parent.inherit('Child', {
    'extra': fields.String
})
```

The `class` field in this example will be populated with the serialized model name only if the property does not exists in the serialized object.

The *`Polymorph`* field allows you to specify a mapping between Python classes and fields specifications.

```
mapping = {
    Child1: child1_fields,
    Child2: child2_fields,
}

fields = api.model('Thing', {
    owner: fields.Polymorph(mapping)
})
```

3.3.11 Custom 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'),
}
```

You can also use the `__schema_format__`, `__schema_type__` and `__schema_example__` to specify the produced types and examples:

```
class MyIntField(fields.Integer):
    __schema_format__ = 'int64'

class MySpecialField(fields.Raw):
    __schema_type__ = 'some-type'
    __schema_format__ = 'some-format'

class MyVerySpecialField(fields.Raw):
    __schema_example__ = 'hello, world'
```

3.3.12 Skip fields which value is None

You can skip those fields which values is `None` instead of marshaling those fields with JSON value, null. This feature is useful to reduce the size of response when you have a lots of fields which value may be None, but which fields are `None` are unpredictable.

Let consider the following example with an optional `skip_none` keyword argument be set to True.

```
>>> from flask_restx import Model, fields, marshal_with
>>> import json
>>> model = Model('Model', {
...     'name': fields.String,
...     'address_1': fields.String,
...     'address_2': fields.String
... })
>>> @marshal_with(model, skip_none=True)
... def get():
...     return {'name': 'John', 'address_1': None}
...
>>> get()
OrderedDict([('name', 'John')])
```

You can see that `address_1` and `address_2` are skipped by *`marshal_with()`*. `address_1` be skipped because value is `None`. `address_2` be skipped because the dictionary return by `get()` have no key, `address_2`.

Skip none in Nested fields

If your module use *`fields.Nested`*, you need to pass `skip_none=True` keyword argument to *`fields.Nested`*.

```
>>> from flask_restx import Model, fields, marshal_with
>>> import json
>>> model = Model('Model', {
...     'name': fields.String,
...     'location': fields.Nested(location_model, skip_none=True)
... })
```

3.3.13 Define model using JSON Schema

You can define models using JSON Schema (Draft v4).

```
address = api.schema_model('Address', {
    'properties': {
        'road': {
            'type': 'string'
        },
    },
    'type': 'object'
})

person = address = api.schema_model('Person', {
    'required': ['address'],
    'properties': {
        'name': {
            'type': 'string'
```

(continues on next page)

(continued from previous page)

```
        },
        'age': {
            'type': 'integer'
        },
        'birthdate': {
            'type': 'string',
            'format': 'date-time'
        },
        'address': {
            '$ref': '#/definitions/Address',
        }
    },
    'type': 'object'
})
```

3.4 Request Parsing

Warning: The whole request parser part of Flask-RESTX 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-RESTX'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.

3.4.1 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_restx import reqparse

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

Note: 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. See *Error Messages* for more details.

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

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

3.4.2 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!")
```

3.4.3 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']
```

If you expect a coma separated list, use the `action='split'`:

```
parser.add_argument('fruits', action='split')
```

This will let you make queries like

```
curl http://api.example.com -d "fruits=apple,lemon,cherry"
```

And your args will look like this :

```
args = parser.parse_args()
args['fruits']    # ['apple', 'lemon', 'cherry']
```

3.4.4 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']
```

3.4.5 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')
```

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

Note: Using `location='form'` is way to both validate form data and document your form fields.

3.4.6 Multiple Locations

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

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

When multiple locations are specified, the arguments from all locations specified are combined into a single `MultiDict`. The last `location` listed takes precedence in the result set.

If the argument location list includes the `headers` location the argument names will no longer be case insensitive and must match their title case names (see `str.title()`). Specifying `location='headers'` (not as a list) will retain case insensitivity.

3.4.7 Advanced types handling

Sometimes, you need more than a primitive type to handle input validation. The *inputs* module provides some common type handling like:

- *boolean()* for wider boolean handling
- *ipv4()* and *ipv6()* for IP adresses
- *date_from_iso8601()* and *datetime_from_iso8601()* for ISO8601 date and datetime handling

You just have to use them as *type* argument:

```
parser.add_argument('flag', type=inputs.boolean)
```

See the *inputs* documentation for full list of available inputs.

You can also write your own:

```
def my_type(value):
    '''Parse my type'''
    if not condition:
        raise ValueError('This is not my type')
    return parse(value)

# Swagger documentation
my_type.__schema__ = {'type': 'string', 'format': 'my-custom-format'}
```

3.4.8 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_restx 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
```

3.4.9 File Upload

To handle file upload with the *RequestParser*, you need to use the *files* location and to set the type to FileStorage.

```
from werkzeug.datastructures import FileStorage

upload_parser = api.parser()
upload_parser.add_argument('file', location='files',
                           type=FileStorage, required=True)


@api.route('/upload/')
@api.expect(upload_parser)
class Upload(Resource):
    def post(self):
        uploaded_file = args['file']  # This is FileStorage instance
        url = do_something_with_file(uploaded_file)
        return {'url': url}, 201
```

See the dedicated Flask documentation section.

3.4.10 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_restx 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
```

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

3.4.11 Error Messages

Error messages for each field may be customized using the `help` parameter to `Argument` (and also `RequestParser.add_argument`).

If no help parameter is provided, the error message for the field will be the string representation of the type error itself. If `help` is provided, then the error message will be the value of `help`.

`help` may include an interpolation token, `{error_msg}`, that will be replaced with the string representation of the type error. This allows the message to be customized while preserving the original error:

```
from flask_restx import reqparse


parser = reqparse.RequestParser()
parser.add_argument(
    'foo',
    choices=('one', 'two'),
    help='Bad choice: {error_msg}'
)

# If a request comes in with a value of "three" for `foo`:

{
    "message":  {
        "foo": "Bad choice: three is not a valid choice",
    }
}
```

3.5 Error handling

3.5.1 HTTPException handling

Werkzeug HTTPException are automatically properly seriliazed reusing the description attribute.

```
from werkzeug.exceptions import BadRequest
raise BadRequest()
```

will return a 400 HTTP code and output

```
{
    "message": "The browser (or proxy) sent a request that this server could not␣
↪understand."
}
```

whereas this:

```
from werkzeug.exceptions import BadRequest
raise BadRequest('My custom message')
```

will output

```
{
    "message": "My custom message"
}
```

You can attach extras attributes to the output by providing a data attribute to your exception.

```
from werkzeug.exceptions import BadRequest
e = BadRequest('My custom message')
e.data = {'custom': 'value'}
raise e
```

will output

```
{
    "message": "My custom message",
    "custom": "value"
}
```

3.5.2 The Flask abort helper

The `abort` helper properly wraps errors into a `HTTPException` so it will have the same behavior.

```
from flask import abort
abort(400)
```

will return a 400 HTTP code and output

```
{
    "message": "The browser (or proxy) sent a request that this server could not␣
↪understand."
}
```

whereas this:

```
from flask import abort
abort(400, 'My custom message')
```

will output

```
{
    "message": "My custom message"
}
```

3.5.3 The Flask-RESTX abort helper

The *`errors.abort()`* and the *`Namespace.abort()`* helpers works like the original Flask `flask.abort()` but it will also add the keyword arguments to the response.

```
from flask_restx import abort
abort(400, custom='value')
```

will return a 400 HTTP code and output

```
{
    "message": "The browser (or proxy) sent a request that this server could not␣
↪understand.",
    "custom": "value"
}
```

whereas this:

```
from flask import abort
abort(400, 'My custom message', custom='value')
```

will output

```
{
    "message": "My custom message",
    "custom": "value"
}
```

3.5.4 The `@api.errorhandler` decorator

The *`@api.errorhandler`* decorator allows you to register a specific handler for a given exception (or any exceptions inherited from it), in the same manner that you can do with Flask/Blueprint `@errorhandler` decorator.

```
@api.errorhandler(RootException)
def handle_root_exception(error):
    '''Return a custom message and 400 status code'''
    return {'message': 'What you want'}, 400


@api.errorhandler(CustomException)
def handle_custom_exception(error):
    '''Return a custom message and 400 status code'''
    return {'message': 'What you want'}, 400


@api.errorhandler(AnotherException)
def handle_another_exception(error):
    '''Return a custom message and 500 status code'''
    return {'message': error.specific}


@api.errorhandler(FakeException)
def handle_fake_exception_with_header(error):
    '''Return a custom message and 400 status code'''
    return {'message': error.message}, 400, {'My-Header': 'Value'}


@api.errorhandler(NoResultFound)
def handle_no_result_exception(error):
    '''Return a custom not found error message and 404 status code'''
    return {'message': error.specific}, 404
```

Note: A "NoResultFound" error with description is required by the OpenAPI 2.0 spec. The docstring in the error handle function is output in the swagger.json as the description.

You can also document the error:

```
@api.errorhandler(FakeException)
@api.marshal_with(error_fields, code=400)
@api.header('My-Header',  'Some description')
def handle_fake_exception_with_header(error):
    '''This is a custom error'''
    return {'message': error.message}, 400, {'My-Header': 'Value'}


@api.route('/test/')
class TestResource(Resource):
```

(continues on next page)

(continued from previous page)

```
    def get(self):
        '''
        Do something

        :raises CustomException: In case of something
        '''
        pass
```

In this example, the `:raise:` docstring will be automatically extracted and the response 400 will be documented properly.

It also allows for overriding the default error handler when used without parameter:

```
@api.errorhandler
def default_error_handler(error):
    '''Default error handler'''
    return {'message': str(error)}, getattr(error, 'code', 500)
```

Note: Flask-RESTX will return a message in the error response by default. If a custom response is required as an error and the message field is not needed, it can be disabled by setting `ERROR_INCLUDE_MESSAGE` to `False` in your application config.

Error handlers can also be registered on namespaces. An error handler registered on a namespace will override one registered on the api.

```
ns = Namespace('cats', description='Cats related operations')

@ns.errorhandler
def specific_namespace_error_handler(error):
    '''Namespace error handler'''
    return {'message': str(error)}, getattr(error, 'code', 500)
```

3.6 Fields masks

Flask-RESTX support partial object fetching (aka. fields mask) by supplying a custom header in the request.

By default the header is `X-Fields` but it can be changed with the `RESTX_MASK_HEADER` parameter.

3.6.1 Syntax

The syntax is actually quite simple. You just provide a coma separated list of field names, optionally wrapped in brackets.

```
# These two mask are equivalents
mask = '{name,age}'
# or
mask = 'name,age'
data = requests.get('/some/url/', headers={'X-Fields': mask})
assert len(data) == 2
assert 'name' in data
assert 'age' in data
```

To specify a nested fields mask, simply provide it in bracket following the field name:

```
mask = '{name, age, pet{name}}'
```

Nesting specification works with nested object or list of objects:

```
# Will apply the mask {name} to each pet
# in the pets list.
mask = '{name, age, pets{name}}'
```

There is a special star token meaning "all remaining fields". It allows to only specify nested filtering:

```
# Will apply the mask {name} to each pet
# in the pets list and take all other root fields
# without filtering.
mask = '{pets{name},*}'

# Will not filter anything
mask = '*'
```

3.6.2 Usage

By default, each time you use `api.marshal` or `@api.marshal_with`, the mask will be automatically applied if the header is present.

The header will be exposed as a Swagger parameter each time you use the `@api.marshal_with` decorator.

As Swagger does not permit exposing a global header once it can make your Swagger specifications a lot more verbose. You can disable this behavior by setting `RESTX_MASK_SWAGGER` to `False`.

You can also specify a default mask that will be applied if no header mask is found.

```
class MyResource(Resource):
    @api.marshal_with(my_model, mask='name,age')
    def get(self):
        pass
```

Default mask can also be handled at model level:

```
model = api.model('Person', {
    'name': fields.String,
    'age': fields.Integer,
    'boolean': fields.Boolean,
}, mask='{name,age}')
```

It will be exposed into the model *x-mask* vendor field:

```
{"definitions": {
    "Test": {
        "properties": {
            "age": {"type": "integer"},
            "boolean": {"type": "boolean"},
            "name": {"type": "string"}
        },
        "x-mask": "{name,age}"
    }
}}
```

To override default masks, you need to give another mask or pass * as mask.

3.7 Swagger documentation

Swagger API documentation is automatically generated and available from your API's root URL. You can configure the documentation using the `@api.doc()` decorator.

3.7.1 Documenting with the `@api.doc()` decorator

The `api.doc()` decorator allows you to include additional information in the documentation.

You can document a class or a method:

```
@api.route('/my-resource/<id>', endpoint='my-resource')
@api.doc(params={'id': 'An ID'})
class MyResource(Resource):
    def get(self, id):
        return {}

    @api.doc(responses={403: 'Not Authorized'})
    def post(self, id):
        api.abort(403)
```

3.7.2 Automatically documented models

All models instantiated with *`model()`*, *`clone()`* and *`inherit()`* will be automatically documented in your Swagger specifications.

The *`inherit()`* method will register both the parent and the child in the Swagger models definitions:

```
parent = api.model('Parent', {
    'name': fields.String,
    'class': fields.String(discriminator=True)
})

child = api.inherit('Child', parent, {
    'extra': fields.String
})
```

The above configuration will produce these Swagger definitions:

```
{
    "Parent": {
        "properties": {
            "name": {"type": "string"},
            "class": {"type": "string"}
        },
        "discriminator": "class",
        "required": ["class"]
    },
    "Child": {
        "allOf": [
            {
```

(continues on next page)

(continued from previous page)

```
                "$ref": "#/definitions/Parent"
            }, {
                "properties": {
                    "extra": {"type": "string"}
                }
            }
        ]
    }
}
```

3.7.3 The `@api.marshal_with()` decorator

This decorator works like the raw *`marshal_with()`* decorator with the difference that it documents the methods. The optional parameter `code` allows you to specify the expected HTTP status code (200 by default). The optional parameter `as_list` allows you to specify whether or not the objects are returned as a list.

```
resource_fields = api.model('Resource', {
    'name': fields.String,
})

@api.route('/my-resource/<id>', endpoint='my-resource')
class MyResource(Resource):
    @api.marshal_with(resource_fields, as_list=True)
    def get(self):
        return get_objects()

    @api.marshal_with(resource_fields, code=201)
    def post(self):
        return create_object(), 201
```

The `Api.marshal_list_with()` decorator is strictly equivalent to `Api.marshal_with(fields, as_list=True)()`.

```
resource_fields = api.model('Resource', {
    'name': fields.String,
})

@api.route('/my-resource/<id>', endpoint='my-resource')
class MyResource(Resource):
    @api.marshal_list_with(resource_fields)
    def get(self):
        return get_objects()

    @api.marshal_with(resource_fields)
    def post(self):
        return create_object()
```

3.7.4 The `@api.expect()` decorator

The `@api.expect()` decorator allows you to specify the expected input fields. It accepts an optional boolean parameter `validate` indicating whether the payload should be validated. The validation behavior can be customized globally either by setting the `RESTX_VALIDATE` configuration to `True` or passing `validate=True` to the API constructor.

The following examples are equivalent:

- Using the `@api.expect()` decorator:

```
resource_fields = api.model('Resource', {
    'name': fields.String,
})

@api.route('/my-resource/<id>')
class MyResource(Resource):
    @api.expect(resource_fields)
    def get(self):
        pass
```

- Using the `api.doc()` decorator:

```
resource_fields = api.model('Resource', {
    'name': fields.String,
})

@api.route('/my-resource/<id>')
class MyResource(Resource):
    @api.doc(body=resource_fields)
    def get(self):
        pass
```

You can specify lists as the expected input:

```
resource_fields = api.model('Resource', {
    'name': fields.String,
})

@api.route('/my-resource/<id>')
class MyResource(Resource):
    @api.expect([resource_fields])
    def get(self):
        pass
```

You can use *RequestParser* to define the expected input:

```
parser = api.parser()
parser.add_argument('param', type=int, help='Some param', location='form')
parser.add_argument('in_files', type=FileStorage, location='files')


@api.route('/with-parser/', endpoint='with-parser')
class WithParserResource(restx.Resource):
    @api.expect(parser)
    def get(self):
        return {}
```

Validation can be enabled or disabled on a particular endpoint:

```
resource_fields = api.model('Resource', {
    'name': fields.String,
})

@api.route('/my-resource/<id>')
```

(continues on next page)

(continued from previous page)

```
class MyResource(Resource):
    # Payload validation disabled
    @api.expect(resource_fields)
    def post(self):
        pass

    # Payload validation enabled
    @api.expect(resource_fields, validate=True)
    def post(self):
        pass
```

An example of application-wide validation by config:

```
app.config['RESTX_VALIDATE'] = True

api = Api(app)

resource_fields = api.model('Resource', {
    'name': fields.String,
})

@api.route('/my-resource/<id>')
class MyResource(Resource):
    # Payload validation enabled
    @api.expect(resource_fields)
    def post(self):
        pass

    # Payload validation disabled
    @api.expect(resource_fields, validate=False)
    def post(self):
        pass
```

An example of application-wide validation by constructor:

```
api = Api(app, validate=True)

resource_fields = api.model('Resource', {
    'name': fields.String,
})

@api.route('/my-resource/<id>')
class MyResource(Resource):
    # Payload validation enabled
    @api.expect(resource_fields)
    def post(self):
        pass

    # Payload validation disabled
    @api.expect(resource_fields, validate=False)
    def post(self):
        pass
```

3.7.5 Documenting with the `@api.response()` decorator

The `@api.response()` decorator allows you to document the known responses and is a shortcut for `@api.doc(responses='...')`.

The following two definitions are equivalent:

```
@api.route('/my-resource/')
class MyResource(Resource):
    @api.response(200, 'Success')
    @api.response(400, 'Validation Error')
    def get(self):
        pass


@api.route('/my-resource/')
class MyResource(Resource):
    @api.doc(responses={
        200: 'Success',
        400: 'Validation Error'
    })
    def get(self):
        pass
```

You can optionally specify a response model as the third argument:

```
model = api.model('Model', {
    'name': fields.String,
})

@api.route('/my-resource/')
class MyResource(Resource):
    @api.response(200, 'Success', model)
    def get(self):
        pass
```

The `@api.marshal_with()` decorator automatically documents the response:

```
model = api.model('Model', {
    'name': fields.String,
})

@api.route('/my-resource/')
class MyResource(Resource):
    @api.response(400, 'Validation error')
    @api.marshal_with(model, code=201, description='Object created')
    def post(self):
        pass
```

You can specify a default response sent without knowing the response code:

```
@api.route('/my-resource/')
class MyResource(Resource):
    @api.response('default', 'Error')
    def get(self):
        pass
```

3.7.6 The `@api.route()` decorator

You can provide class-wide documentation using the `doc` parameter of `Api.route()`. This parameter accepts the same values as the `Api.doc()` decorator.

For example, these two declarations are equivalent:

- Using `@api.doc()`:

```
@api.route('/my-resource/<id>', endpoint='my-resource')
@api.doc(params={'id': 'An ID'})
class MyResource(Resource):
    def get(self, id):
        return {}
```

- Using `@api.route()`:

```
@api.route('/my-resource/<id>', endpoint='my-resource', doc={'params':{'id': 'An ID'}}
↪)
class MyResource(Resource):
    def get(self, id):
        return {}
```

Multiple Routes per Resource

Multiple `Api.route()` decorators can be used to add multiple routes for a `Resource`. The `doc` parameter provides documentation **per route**.

For example, here the `description` is applied only to the `/also-my-resource/<id>` route:

```
@api.route("/my-resource/<id>")
@api.route(
    "/also-my-resource/<id>",
    doc={"description": "Alias for /my-resource/<id>"},
)
class MyResource(Resource):
    def get(self, id):
        return {}
```

Here, the `/also-my-resource/<id>` route is marked as deprecated:

```
@api.route("/my-resource/<id>")
@api.route(
    "/also-my-resource/<id>",
    doc={
        "description": "Alias for /my-resource/<id>, this route is being phased out␣
↪in V2",
        "deprecated": True,
    },
)
class MyResource(Resource):
    def get(self, id):
        return {}
```

Documentation applied to the `Resource` using `Api.doc()` is *shared* amongst all routes unless explicitly overridden:

```
@api.route("/my-resource/<id>")
@api.route(
"/also-my-resource/<id>",
doc={"description": "Alias for /my-resource/<id>"},
)
@api.doc(params={"id": "An ID", description="My resource"})
class MyResource(Resource):
def get(self, id):
    return {}
```

Here, the `id` documentation from the `@api.doc()` decorator is present in both routes, `/my-resource/<id>` inherits the `My resource` description from the `@api.doc()` decorator and `/also-my-resource/<id>` overrides the description with `Alias for /my-resource/<id>`.

Routes with a `doc` parameter are given a *unique* Swagger `operationId`. Routes without `doc` parameter have the same Swagger `operationId` as they are deemed the same operation.

3.7.7 Documenting the fields

Every Flask-RESTX field accepts optional arguments used to document the field:

- `required`: a boolean indicating if the field is always set (*default*: `False`)
- `description`: some details about the field (*default*: `None`)
- `example`: an example to use when displaying (*default*: `None`)

There are also field-specific attributes:

- **The `String` field accepts the following optional arguments:**
  - `enum`: an array restricting the authorized values.
  - `min_length`: the minimum length expected.
  - `max_length`: the maximum length expected.
  - `pattern`: a RegExp pattern used to validate the string.
- **The `Integer`, `Float` and `Arbitrary` fields accept the following optional arguments:**
  - `min`: restrict the minimum accepted value.
  - `max`: restrict the maximum accepted value.
  - `exclusiveMin`: if `True`, minimum value is not in allowed interval.
  - `exclusiveMax`: if `True`, maximum value is not in allowed interval.
  - `multiple`: specify that the number must be a multiple of this value.
- The `DateTime` field accepts the `min`, `max`, `exclusiveMin` and `exclusiveMax` optional arguments. These should be dates or datetimes (either ISO strings or native objects).

```
my_fields = api.model('MyModel', {
    'name': fields.String(description='The name', required=True),
    'type': fields.String(description='The object type', enum=['A', 'B']),
    'age': fields.Integer(min=0),
})
```

3.7.8 Documenting the methods

Each resource will be documented as a Swagger path.

Each resource method (`get`, `post`, `put`, `delete`, `path`, `options`, `head`) will be documented as a Swagger operation.

You can specify a unique Swagger `operationId` with the `id` keyword argument:

```
@api.route('/my-resource/')
class MyResource(Resource):
    @api.doc(id='get_something')
    def get(self):
        return {}
```

You can also use the first argument for the same purpose:

```
@api.route('/my-resource/')
class MyResource(Resource):
    @api.doc('get_something')
    def get(self):
        return {}
```

If not specified, a default `operationId` is provided with the following pattern:

```
{{verb}}_{{resource class name | camelCase2dashes }}
```

In the previous example, the default generated `operationId` would be `get_my_resource`.

You can override the default `operationId` generator by providing a callable for the `default_id` parameter. This callable accepts two positional arguments:

- The resource class name
- The HTTP method (lower-case)

```
def default_id(resource, method):
    return ''.join((method, resource))

api = Api(app, default_id=default_id)
```

In the previous example, the generated `operationId` would be `getMyResource`.

Each operation will automatically receive the namespace tag. If the resource is attached to the root API, it will receive the default namespace tag.

Method parameters

Parameters from the URL path are documented automatically. You can provide additional information using the `params` keyword argument of the `api.doc()` decorator:

```
@api.route('/my-resource/<id>', endpoint='my-resource')
@api.doc(params={'id': 'An ID'})
class MyResource(Resource):
    pass
```

or by using the `api.param` shortcut decorator:

```
@api.route('/my-resource/<id>', endpoint='my-resource')
@api.param('id', 'An ID')
class MyResource(Resource):
    pass
```

Input and output models

You can specify the serialized output model using the `model` keyword argument of the `api.doc()` decorator.

For `POST` and `PUT` methods, use the `body` keyword argument to specify the input model.

```
fields = api.model('MyModel', {
    'name': fields.String(description='The name', required=True),
    'type': fields.String(description='The object type', enum=['A', 'B']),
    'age': fields.Integer(min=0),
})


@api.model(fields={'name': fields.String, 'age': fields.Integer})
class Person(fields.Raw):
    def format(self, value):
        return {'name': value.name, 'age': value.age}


@api.route('/my-resource/<id>', endpoint='my-resource')
@api.doc(params={'id': 'An ID'})
class MyResource(Resource):
    @api.doc(model=fields)
    def get(self, id):
        return {}

    @api.doc(model='MyModel', body=Person)
    def post(self, id):
        return {}
```

If both `body` and `formData` parameters are used, a *SpecsError* will be raised.

Models can also be specified with a *RequestParser*.

```
parser = api.parser()
parser.add_argument('param', type=int, help='Some param', location='form')
parser.add_argument('in_files', type=FileStorage, location='files')

@api.route('/with-parser/', endpoint='with-parser')
class WithParserResource(restx.Resource):
    @api.expect(parser)
    def get(self):
        return {}
```

Note: The decoded payload will be available as a dictionary in the payload attribute in the request context.

```
@api.route('/my-resource/')
class MyResource(Resource):
    def get(self):
        data = api.payload
```

Note: Using *RequestParser* is preferred over the `api.param()` decorator to document form fields as it also perform validation.

Headers

You can document response headers with the `@api.header()` decorator shortcut.

```
@api.route('/with-headers/')
@api.header('X-Header', 'Some class header')
class WithHeaderResource(restx.Resource):
    @api.header('X-Collection', type=[str], collectionType='csv')
    def get(self):
        pass
```

If you need to specify an header that appear only on a given response, just use the *@api.response headers* parameter.

```
@api.route('/response-headers/')
class WithHeaderResource(restx.Resource):
    @api.response(200, 'Success', headers={'X-Header': 'Some header'})
    def get(self):
        pass
```

Documenting expected/request headers is done through the *@api.expect* decorator

```
parser = api.parser()
parser.add_argument('Some-Header', location='headers')

@api.route('/expect-headers/')
@api.expect(parser)
class ExpectHeaderResource(restx.Resource):
    def get(self):
        pass
```

3.7.9 Cascading

Method documentation takes precedence over class documentation, and inherited documentation takes precedence over parent documentation.

For example, these two declarations are equivalent:

- Class documentation is inherited by methods:

```
@api.route('/my-resource/<id>', endpoint='my-resource')
@api.params('id', 'An ID')
class MyResource(Resource):
    def get(self, id):
        return {}
```

- Class documentation is overridden by method-specific documentation:

```
@api.route('/my-resource/<id>', endpoint='my-resource')
@api.param('id', 'Class-wide description')
class MyResource(Resource):
    @api.param('id', 'An ID')
    def get(self, id):
        return {}
```

You can also provide method-specific documentation from a class decorator. The following example will produce the same documentation as the two previous examples:

```
@api.route('/my-resource/<id>', endpoint='my-resource')
@api.params('id', 'Class-wide description')
@api.doc(get={'params': {'id': 'An ID'}})
class MyResource(Resource):
    def get(self, id):
        return {}
```

3.7.10 Marking as deprecated

You can mark resources or methods as deprecated with the `@api.deprecated` decorator:

```
# Deprecate the full resource
@api.deprecated
@api.route('/resource1/')
class Resource1(Resource):
    def get(self):
        return {}

# Deprecate methods
@api.route('/resource4/')
class Resource4(Resource):
    def get(self):
        return {}

    @api.deprecated
    def post(self):
        return {}

    def put(self):
        return {}
```

3.7.11 Hiding from documentation

You can hide some resources or methods from documentation using any of the following:

```
# Hide the full resource
@api.route('/resource1/', doc=False)
class Resource1(Resource):
    def get(self):
        return {}

@api.route('/resource2/')
@api.doc(False)
class Resource2(Resource):
```

(continues on next page)

(continued from previous page)

```
    def get(self):
        return {}

@api.route('/resource3/')
@api.hide
class Resource3(Resource):
    def get(self):
        return {}

# Hide methods
@api.route('/resource4/')
@api.doc(delete=False)
class Resource4(Resource):
    def get(self):
        return {}

    @api.doc(False)
    def post(self):
        return {}

    @api.hide
    def put(self):
        return {}

    def delete(self):
        return {}
```

Note: Namespace tags without attached resources will be hidden automatically from the documentation.

3.7.12 Documenting authorizations

You can use the `authorizations` keyword argument to document authorization information. See Swagger Authentication documentation for configuration details.

- `authorizations` is a Python dictionary representation of the Swagger `securityDefinitions` configuration.

```
authorizations = {
    'apikey': {
        'type': 'apiKey',
        'in': 'header',
        'name': 'X-API-KEY'
    }
}
api = Api(app, authorizations=authorizations)
```

Then decorate each resource and method that requires authorization:

```
@api.route('/resource/')
class Resource1(Resource):
    @api.doc(security='apikey')
    def get(self):
        pass
```

(continues on next page)

(continued from previous page)

```
    @api.doc(security='apikey')
    def post(self):
        pass
```

You can apply this requirement globally with the `security` parameter on the `Api` constructor:

```
authorizations = {
    'apikey': {
        'type': 'apiKey',
        'in': 'header',
        'name': 'X-API-KEY'
    }
}
api = Api(app, authorizations=authorizations, security='apikey')
```

You can have multiple security schemes:

```
authorizations = {
    'apikey': {
        'type': 'apiKey',
        'in': 'header',
        'name': 'X-API'
    },
    'oauth2': {
        'type': 'oauth2',
        'flow': 'accessCode',
        'tokenUrl': 'https://somewhere.com/token',
        'authorizationUrl': 'https://somewhere.com/auth',
        'scopes': {
            'read': 'Grant read-only access',
            'write': 'Grant read-write access',
        }
    }
}
api = Api(self.app, security=['apikey', {'oauth2': 'read'}],␣
↪authorizations=authorizations)
```

Security schemes can be overridden for a particular method:

```
@api.route('/authorizations/')
class Authorized(Resource):
    @api.doc(security=[{'oauth2': ['read', 'write']}])
    def get(self):
        return {}
```

You can disable security on a given resource or method by passing `None` or an empty list as the `security` parameter:

```
@api.route('/without-authorization/')
class WithoutAuthorization(Resource):
    @api.doc(security=[])
    def get(self):
        return {}

    @api.doc(security=None)
    def post(self):
        return {}
```

3.7.13 Expose vendor Extensions

Swaggers allows you to expose custom vendor extensions and you can use them in Flask-RESTX with the *@api.vendor* decorator.

It supports both extensions as *dict* or *kwargs* and perform automatique *x-* prefix:

```
@api.route('/vendor/')
@api.vendor(extension1='any authorized value')
class Vendor(Resource):
    @api.vendor({
        'extension-1': {'works': 'with complex values'},
        'x-extension-3': 'x- prefix is optional',
    })
    def get(self):
        return {}
```

3.7.14 Export Swagger specifications

You can export the Swagger specifications for your API:

```
from flask import json

from myapp import api

print(json.dumps(api.__schema__))
```

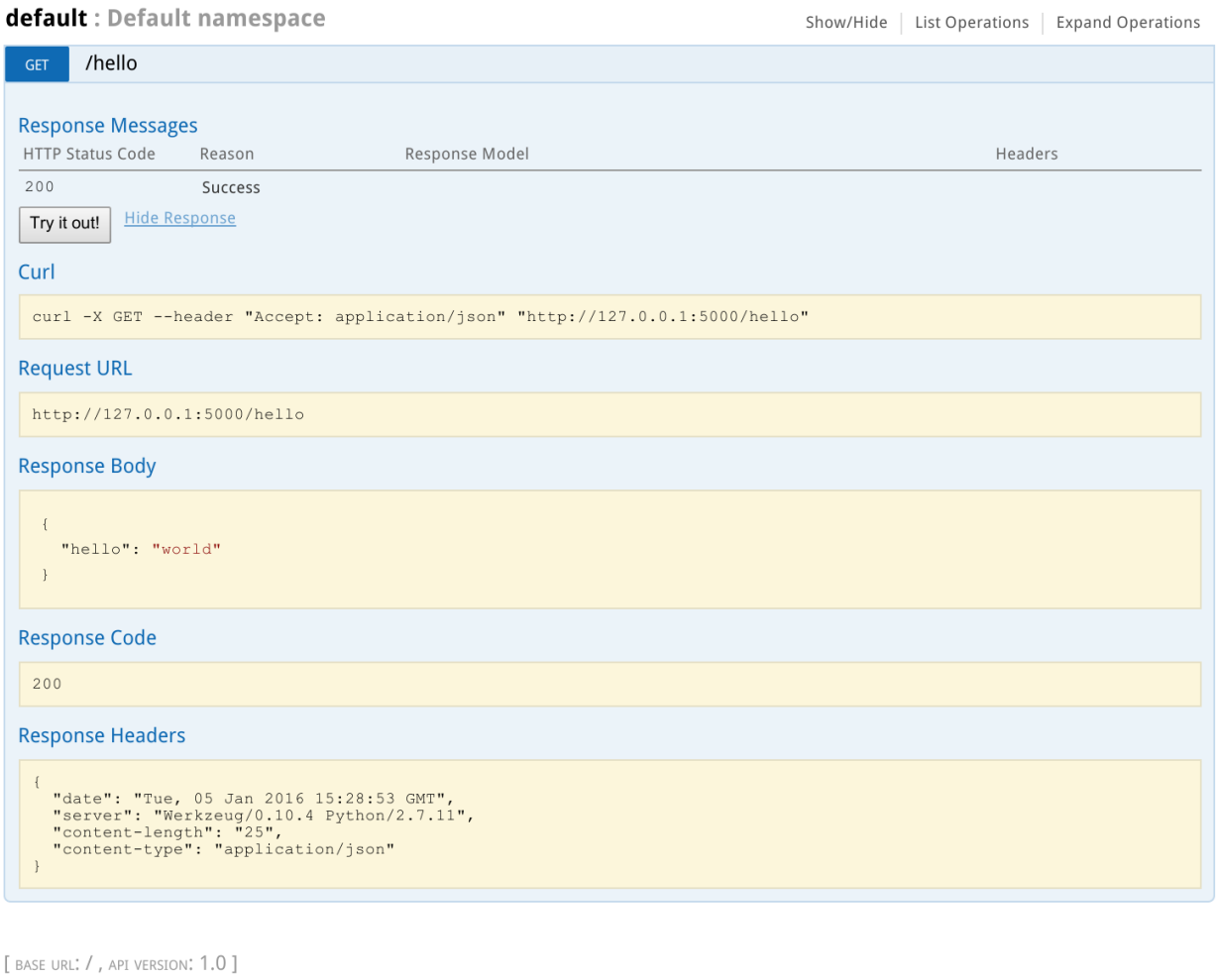
3.7.15 Swagger UI

By default `flask-restx` provides Swagger UI documentation, served from the root URL of the API.

```
from flask import Flask
from flask_restx import Api, Resource, fields

app = Flask(__name__)
api = Api(app, version='1.0', title='Sample API',
    description='A sample API',
)

@api.route('/my-resource/<id>')
@api.doc(params={'id': 'An ID'})
class MyResource(Resource):
    def get(self, id):
        return {}

    @api.response(403, 'Not Authorized')
    def post(self, id):
        api.abort(403)


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

If you run the code below and visit your API's root URL (http://localhost:5000) you can view the automatically-generated Swagger UI documentation.

API



[BASE URL: / , API VERSION: 1.0]

Customization

You can control the Swagger UI path with the `doc` parameter (defaults to the API root):

```
from flask import Flask, Blueprint
from flask_restx import Api

app = Flask(__name__)
blueprint = Blueprint('api', __name__, url_prefix='/api')
api = Api(blueprint, doc='/doc/')

app.register_blueprint(blueprint)

assert url_for('api.doc') == '/api/doc/'
```

You can specify a custom validator URL by setting `config.SWAGGER_VALIDATOR_URL`:

```
from flask import Flask
from flask_restx import Api
```

(continues on next page)

(continued from previous page)

```
app = Flask(__name__)
app.config.SWAGGER_VALIDATOR_URL = 'http://domain.com/validator'

api = Api(app)
```

You can enable [OAuth2 Implicit Flow](https://oauth.net/2/grant-types/implicit/) for retrieving an authorization token for testing api endpoints interactively within Swagger UI. The `config.SWAGGER_UI_OAUTH_CLIENT_ID` and `authorizationUrl` and `scopes` will be specific to your OAuth2 IDP configuration. The realm string is added as a query parameter to authorizationUrl and tokenUrl. These values are all public knowledge. No *client secret* is specified here. .. Using PKCE instead of Implicit Flow depends on https://github.com/swagger-api/swagger-ui/issues/5348

```
from flask import Flask
from flask_restx import Api

app = Flask(__name__)

app.config.SWAGGER_UI_OAUTH_CLIENT_ID = 'MyClientId'
app.config.SWAGGER_UI_OAUTH_REALM = '-'
app.config.SWAGGER_UI_OAUTH_APP_NAME = 'Demo'

api = Api(
    app,
    title=app.config.SWAGGER_UI_OAUTH_APP_NAME,
    security={'OAuth2': ['read', 'write']},
    authorizations={
        'OAuth2': {
            'type': 'oauth2',
            'flow': 'implicit',
            'authorizationUrl': 'https://idp.example.com/authorize?audience=https://
↪app.example.com',
            'clientId': app.config.SWAGGER_UI_OAUTH_CLIENT_ID,
            'scopes': {
                'openid': 'Get ID token',
                'profile': 'Get identity',
            }
        }
    }
)
```

You can also specify the initial expansion state with the `config.SWAGGER_UI_DOC_EXPANSION` setting (`'none'`, `'list'` or `'full'`):

```
from flask import Flask
from flask_restx import Api

app = Flask(__name__)
app.config.SWAGGER_UI_DOC_EXPANSION = 'list'

api = Api(app)
```

By default, operation ID is hidden as well as request duration, you can enable them respectively with:

```
from flask import Flask
from flask_restx import Api
```

(continues on next page)

(continued from previous page)

```
app = Flask(__name__)
app.config.SWAGGER_UI_OPERATION_ID = True
app.config.SWAGGER_UI_REQUEST_DURATION = True

api = Api(app)
```

If you need a custom UI, you can register a custom view function with the *documentation()* decorator:

```
from flask import Flask
from flask_restx import Api, apidoc

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

@api.documentation
def custom_ui():
    return apidoc.ui_for(api)
```

Configuring "Try it Out"

By default, all paths and methods have a "Try it Out" button for performing API requests in the browser. These can be disable **per method** with the `SWAGGER_SUPPORTED_SUBMIT_METHODS` configuration option, supporting the same values as the `supportedSubmitMethods` Swagger UI parameter.

```
from flask import Flask
from flask_restx import Api

app = Flask(__name__)

# disable Try it Out for all methods
app.config.SWAGGER_SUPPORTED_SUBMIT_METHODS = []

# enable Try it Out for specific methods
app.config.SWAGGER_SUPPORTED_SUBMIT_METHODS = ["get", "post"]

api = Api(app)
```

Disabling the documentation

To disable Swagger UI entirely, set `doc=False`:

```
from flask import Flask
from flask_restx import Api

app = Flask(__name__)
api = Api(app, doc=False)
```

3.8 Logging

Flask-RESTX extends Flask's logging by providing each `API` and `Namespace` it's own standard Python `logging.Logger` instance. This allows separation of logging on a per namespace basis to allow more fine-grained detail and configuration.

By default, these loggers inherit configuration from the Flask application object logger.

```
import logging

import flask

from flask_restx import Api, Resource

# configure root logger
logging.basicConfig(level=logging.INFO)

app = flask.Flask(__name__)

api = Api(app)


# each of these loggers uses configuration from app.logger
ns1 = api.namespace('api/v1', description='test')
ns2 = api.namespace('api/v2', description='test')


@ns1.route('/my-resource')
class MyResource(Resource):
    def get(self):
        # will log
        ns1.logger.info("hello from ns1")
        return {"message": "hello"}


@ns2.route('/my-resource')
class MyNewResource(Resource):
    def get(self):
        # won't log due to INFO log level from app.logger
        ns2.logger.debug("hello from ns2")
        return {"message": "hello"}
```

Loggers can be configured individually to override the configuration from the Flask application object logger. In the above example, `ns2` log level can be set to `DEBUG` individually:

```
# ns1 will have log level INFO from app.logger
ns1 = api.namespace('api/v1', description='test')

# ns2 will have log level DEBUG
ns2 = api.namespace('api/v2', description='test')
ns2.logger.setLevel(logging.DEBUG)


@ns1.route('/my-resource')
class MyResource(Resource):
    def get(self):
        # will log
        ns1.logger.info("hello from ns1")
        return {"message": "hello"}


@ns2.route('/my-resource')
class MyNewResource(Resource):
    def get(self):
```

(continues on next page)

(continued from previous page)

```
        # will log
        ns2.logger.debug("hello from ns2")
        return {"message": "hello"}
```

Adding additional handlers:

```
# configure a file handler for ns1 only
ns1 = api.namespace('api/v1')
fh = logging.FileHandler("v1.log")
ns1.logger.addHandler(fh)

ns2 = api.namespace('api/v2')


@ns1.route('/my-resource')
class MyResource(Resource):
    def get(self):
        # will log to *both* v1.log file and app.logger handlers
        ns1.logger.info("hello from ns1")
        return {"message": "hello"}


@ns2.route('/my-resource')
class MyNewResource(Resource):
    def get(self):
        # will log to *only* app.logger handlers
        ns2.logger.info("hello from ns2")
        return {"message": "hello"}
```

3.9 Postman

To help you testing, you can export your API as a Postman collection.

```
from flask import json

from myapp import api

urlvars = False  # Build query strings in URLs
swagger = True  # Export Swagger specifications
data = api.as_postman(urlvars=urlvars, swagger=swagger)
print(json.dumps(data))
```

3.10 Scaling your project

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

3.10.1 Multiple namespaces

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

Flask-RESTX provides a way to use almost the same pattern as Flask's *blueprint*. The main idea is to split your app into reusable namespaces.

Here's an example directory structure:

```
project/
├── app.py
├── core
│   ├── __init__.py
│   ├── utils.py
│   └── ...
└── apis
    ├── __init__.py
    ├── namespace1.py
    ├── namespace2.py
    ├── ...
    └── namespaceX.py
```

The *app* module will serve as a main application entry point following one of the classic Flask patterns (See Larger Applications and Application Factories).

The *core* module is an example, it contains the business logic. In fact, you call it whatever you want, and there can be many packages.

The *apis* package will be your main API entry point that you need to import and register on the application, whereas the namespaces modules are reusable namespaces designed like you would do with Flask's Blueprint.

A namespace module contains models and resources declarations. For example:

```python
from flask_restx import Namespace, Resource, fields

api = Namespace('cats', description='Cats related operations')

cat = api.model('Cat', {
    'id': fields.String(required=True, description='The cat identifier'),
    'name': fields.String(required=True, description='The cat name'),
})

CATS = [
    {'id': 'felix', 'name': 'Felix'},
]

@api.route('/')
class CatList(Resource):
    @api.doc('list_cats')
    @api.marshal_list_with(cat)
    def get(self):
        '''List all cats'''
        return CATS

@api.route('/<id>')
@api.param('id', 'The cat identifier')
@api.response(404, 'Cat not found')
class Cat(Resource):
```

(continues on next page)

(continued from previous page)

```
    @api.doc('get_cat')
    @api.marshal_with(cat)
    def get(self, id):
        '''Fetch a cat given its identifier'''
        for cat in CATS:
            if cat['id'] == id:
                return cat
        api.abort(404)
```

The *apis.__init__* module should aggregate them:

```
from flask_restx import Api

from .namespace1 import api as ns1
from .namespace2 import api as ns2
# ...
from .namespaceX import api as nsX

api = Api(
    title='My Title',
    version='1.0',
    description='A description',
    # All API metadatas
)

api.add_namespace(ns1)
api.add_namespace(ns2)
# ...
api.add_namespace(nsX)
```

You can define custom url-prefixes for namespaces during registering them in your API. You don't have to bind url-prefix while declaration of Namespace object.

```
from flask_restx import Api

from .namespace1 import api as ns1
from .namespace2 import api as ns2
# ...
from .namespaceX import api as nsX

api = Api(
    title='My Title',
    version='1.0',
    description='A description',
    # All API metadatas
)

api.add_namespace(ns1, path='/prefix/of/ns1')
api.add_namespace(ns2, path='/prefix/of/ns2')
# ...
api.add_namespace(nsX, path='/prefix/of/nsX')
```

Using this pattern, you simply have to register your API in *app.py* like that:

```
from flask import Flask
from apis import api
```

(continues on next page)

(continued from previous page)

```
app = Flask(__name__)
api.init_app(app)

app.run(debug=True)
```

3.10.2 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 Blueprint
from flask_restx import Api

blueprint = Blueprint('api', __name__)
api = Api(blueprint)
# ...
```

Using a *blueprint* will allow you to mount your API on any url prefix and/or subdomain in you application:

```
from flask import Flask
from apis import blueprint as api

app = Flask(__name__)
app.register_blueprint(api, url_prefix='/api/1')
app.run(debug=True)
```

Note: 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.

Note: When using blueprints, remember to use the blueprint name with `url_for()`:

```
# without blueprint
url_for('my_api_endpoint')

# with blueprint
url_for('api.my_api_endpoint')
```

3.10.3 Multiple APIs with reusable namespaces

Sometimes you need to maintain multiple versions of an API. If you built your API using namespaces composition, it's quite simple to scale it to multiple APIs.

Given the previous layout, we can migrate it to the following directory structure:

```
project/
├── app.py
├── apiv1.py
├── apiv2.py
```

(continues on next page)

(continued from previous page)

```
└── apis
    ├── __init__.py
    ├── namespace1.py
    ├── namespace2.py
    ├── ...
    └── namespaceX.py
```

Each *apivX* module will have the following pattern:

```
from flask import Blueprint
from flask_restx import Api

api = Api(blueprint)

from .apis.namespace1 import api as ns1
from .apis.namespace2 import api as ns2
# ...
from .apis.namespaceX import api as nsX

blueprint = Blueprint('api', __name__, url_prefix='/api/1')
api = Api(blueprint
    title='My Title',
    version='1.0',
    description='A description',
    # All API metadatas
)

api.add_namespace(ns1)
api.add_namespace(ns2)
# ...
api.add_namespace(nsX)
```

And the app will simply mount them:

```
from flask import Flask
from api1 import blueprint as api1
from apiX import blueprint as apiX

app = Flask(__name__)
app.register_blueprint(api1)
app.register_blueprint(apiX)
app.run(debug=True)
```

These are only proposals and you can do whatever suits your needs. Look at the github repository examples folder for more complete examples.

3.11 Full example

Here is a full example of a TodoMVC API.

```
from flask import Flask
from flask_restx import Api, Resource, fields
from werkzeug.contrib.fixers import ProxyFix
```

(continues on next page)

(continued from previous page)

```
app = Flask(__name__)
app.wsgi_app = ProxyFix(app.wsgi_app)
api = Api(app, version='1.0', title='TodoMVC API',
    description='A simple TodoMVC API',
)

ns = api.namespace('todos', description='TODO operations')

todo = api.model('Todo', {
    'id': fields.Integer(readonly=True, description='The task unique identifier'),
    'task': fields.String(required=True, description='The task details')
})


class TodoDAO(object):
    def __init__(self):
        self.counter = 0
        self.todos = []

    def get(self, id):
        for todo in self.todos:
            if todo['id'] == id:
                return todo
        api.abort(404, "Todo {} doesn't exist".format(id))

    def create(self, data):
        todo = data
        todo['id'] = self.counter = self.counter + 1
        self.todos.append(todo)
        return todo

    def update(self, id, data):
        todo = self.get(id)
        todo.update(data)
        return todo

    def delete(self, id):
        todo = self.get(id)
        self.todos.remove(todo)


DAO = TodoDAO()
DAO.create({'task': 'Build an API'})
DAO.create({'task': '?????'})
DAO.create({'task': 'profit!'})


@ns.route('/')
class TodoList(Resource):
    '''Shows a list of all todos, and lets you POST to add new tasks'''
    @ns.doc('list_todos')
    @ns.marshal_list_with(todo)
    def get(self):
        '''List all tasks'''
        return DAO.todos

    @ns.doc('create_todo')
```

(continues on next page)

(continued from previous page)

```
    @ns.expect(todo)
    @ns.marshal_with(todo, code=201)
    def post(self):
        '''Create a new task'''
        return DAO.create(api.payload), 201


@ns.route('/<int:id>')
@ns.response(404, 'Todo not found')
@ns.param('id', 'The task identifier')
class Todo(Resource):
    '''Show a single todo item and lets you delete them'''
    @ns.doc('get_todo')
    @ns.marshal_with(todo)
    def get(self, id):
        '''Fetch a given resource'''
        return DAO.get(id)

    @ns.doc('delete_todo')
    @ns.response(204, 'Todo deleted')
    def delete(self, id):
        '''Delete a task given its identifier'''
        DAO.delete(id)
        return '', 204

    @ns.expect(todo)
    @ns.marshal_with(todo)
    def put(self, id):
        '''Update a task given its identifier'''
        return DAO.update(id, api.payload)


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

You can find other examples in the github repository examples folder.

3.12 API Reference

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

3.12.1 API

Core

class flask_restx.**Api**(*app=None*, *version='1.0'*, *title=None*, *description=None*, *terms_url=None*, *license=None*, *license_url=None*, *contact=None*, *contact_url=None*, *contact_email=None*, *authorizations=None*, *security=None*, *doc='/'*, *default_id=<function default_id>*, *default='default'*, *default_label='Default namespace'*, *validate=None*, *tags=None*, *prefix=''*, *ordered=False*, *default_mediatype='application/json'*, *decorators=None*, *catch_all_404s=False*, *serve_challenge_on_401=False*, *format_checker=None*, ***kwargs*)

The main entry point for the application. You need to initialize it with a Flask Application:

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

Alternatively, you can use *init_app()* to set the Flask application after it has been constructed.

The endpoint parameter prefix all views and resources:

- The API root/documentation will be `{endpoint}.root`
- A resource registered as 'resource' will be available as `{endpoint}.resource`

Parameters

- **app** (*flask.Flask|flask.Blueprint*) – the Flask application object or a Blueprint
- **version** (*str*) – The API version (used in Swagger documentation)
- **title** (*str*) – The API title (used in Swagger documentation)
- **description** (*str*) – The API description (used in Swagger documentation)
- **terms_url** (*str*) – The API terms page URL (used in Swagger documentation)
- **contact** (*str*) – A contact email for the API (used in Swagger documentation)
- **license** (*str*) – The license associated to the API (used in Swagger documentation)
- **license_url** (*str*) – The license page URL (used in Swagger documentation)
- **endpoint** (*str*) – The API base endpoint (default to 'api).
- **default** (*str*) – The default namespace base name (default to 'default')
- **default_label** (*str*) – The default namespace label (used in Swagger documentation)
- **default_mediatype** (*str*) – The default media type to return
- **validate** (*bool*) – Whether or not the API should perform input payload validation.
- **ordered** (*bool*) – Whether or not preserve order models and marshalling.
- **doc** (*str*) – The documentation path. If set to a false value, documentation is disabled. (Default to '/')
- **decorators** (*list*) – Decorators to attach to every resource
- **catch_all_404s** (*bool*) – Use *handle_error()* to handle 404 errors throughout your app
- **authorizations** (*dict*) – A Swagger Authorizations declaration as dictionary
- **serve_challenge_on_401** (*bool*) – Serve basic authentication challenge with 401 responses (default 'False')

- **format_checker** (*FormatChecker*) – A jsonschema.FormatChecker object that is hooked into the Model validator. A default or a custom FormatChecker can be provided (e.g., with custom checkers), otherwise the default action is to not enforce any format validation.

add_namespace(*ns*, *path=None*)
This method registers resources from namespace for current instance of api. You can use argument path for definition custom prefix url for namespace.

Parameters

- **ns** (Namespace) – the namespace
- **path** – registration prefix of namespace

as_postman(*urlvars=False*, *swagger=False*)
Serialize the API as Postman collection (v1)

Parameters

- **urlvars** (*bool*) – whether to include or not placeholders for query strings
- **swagger** (*bool*) – whether to include or not the swagger.json specifications

base_path
The API path

Return type str

base_url
The API base absolute url

Return type str

default_endpoint(*resource*, *namespace*)
Provide a default endpoint for a resource on a given namespace.

Endpoints are ensured not to collide.

Override this method specify a custom algorithm for default endpoint.

Parameters

- **resource** (Resource) – the resource for which we want an endpoint
- **namespace** (Namespace) – the namespace holding the resource

Returns str An endpoint name

documentation(*func*)
A decorator to specify a view function for the documentation

error_router(*original_handler*, *e*)
This function decides whether the error occurred in a flask-restx endpoint or not. If it happened in a flask-restx endpoint, our handler will be dispatched. If it happened in an unrelated view, the app's original error handler will be dispatched. In the event that the error occurred in a flask-restx endpoint but the local handler can't resolve the situation, the router will fall back onto the original_handler as last resort.

Parameters

- **original_handler** (*function*) – the original Flask error handler for the app
- **e** (*Exception*) – the exception raised while handling the request

errorhandler(*exception*)
A decorator to register an error handler for a given exception

handle_error(*e*)
Error handler for the API transforms a raised exception into a Flask response, with the appropriate HTTP status code and body.

Parameters **e** (*Exception*) – the raised Exception object

init_app(*app*, ***kwargs*)
Allow to lazy register the API on a Flask application:

```
>>> app = Flask(__name__)
>>> api = Api()
>>> api.init_app(app)
```

Parameters

- **app** (*flask.Flask*) – the Flask application object
- **title** (*str*) – The API title (used in Swagger documentation)
- **description** (*str*) – The API description (used in Swagger documentation)
- **terms_url** (*str*) – The API terms page URL (used in Swagger documentation)
- **contact** (*str*) – A contact email for the API (used in Swagger documentation)
- **license** (*str*) – The license associated to the API (used in Swagger documentation)
- **license_url** (*str*) – The license page URL (used in Swagger documentation)

make_response(*data*, **args*, ***kwargs*)
Looks up the representation transformer for the requested media type, invoking the transformer to create a response object. This defaults to default_mediatype if no transformer is found for the requested mediatype. If default_mediatype is None, a 406 Not Acceptable response will be sent as per RFC 2616 section 14.1

Parameters **data** – Python object containing response data to be transformed

mediatypes()
Returns a list of requested mediatypes sent in the Accept header

mediatypes_method()
Return a method that returns a list of mediatypes

namespace(**args*, ***kwargs*)
A namespace factory.

Returns Namespace a new namespace instance

output(*resource*)
Wraps a resource (as a flask view function), for cases where the resource does not directly return a response object

Parameters **resource** – The resource as a flask view function

owns_endpoint(*endpoint*)
Tests if an endpoint name (not path) belongs to this Api. Takes into account the Blueprint name part of the endpoint name.

Parameters **endpoint** (*str*) – The name of the endpoint being checked

Returns bool

payload
Store the input payload in the current request context

render_doc()
Override this method to customize the documentation page

representation(*mediatype*)
Allows additional representation transformers to be declared for the api. Transformers are functions that must be decorated with this method, passing the mediatype the transformer represents. Three arguments are passed to the transformer:

- The data to be represented in the response body
- The http status code
- A dictionary of headers

The transformer should convert the data appropriately for the mediatype and return a Flask response object.

Ex:

```
@api.representation('application/xml')
def xml(data, code, headers):
    resp = make_response(convert_data_to_xml(data), code)
    resp.headers.extend(headers)
    return resp
```

specs_url
The Swagger specifications absolute url (ie. *swagger.json*)

Return type str

unauthorized(*response*)
Given a response, change it to ask for credentials

url_for(*resource*, ***values*)
Generates a URL to the given resource.

Works like `flask.url_for()`.

class flask_restx.**Namespace**(*name*, *description=None*, *path=None*, *decorators=None*, *validate=None*, *authorizations=None*, *ordered=False*, ***kwargs*)
Group resources together.

Namespace is to API what `flask.Blueprint` is for `flask.Flask`.

Parameters

- **name** (*str*) – The namespace name
- **description** (*str*) – An optional short description
- **path** (*str*) – An optional prefix path. If not provided, prefix is `/+name`
- **decorators** (*list*) – A list of decorators to apply to each resources
- **validate** (*bool*) – Whether or not to perform validation on this namespace
- **ordered** (*bool*) – Whether or not to preserve order on models and marshalling
- **api** (`Api`) – an optional API to attache to the namespace

abort(**args*, ***kwargs*)
Properly abort the current request

See: *abort()*

add_resource(*resource*, **urls*, ***kwargs*)
Register a Resource for a given API Namespace

Parameters

- **resource** (Resource) – the resource ro register
- **urls** (*str*) – one or more url routes to match for the resource, standard flask routing rules apply. Any url variables will be passed to the resource method as args.
- **endpoint** (*str*) – endpoint name (defaults to `Resource.__name__.lower()` Can be used to reference this route in *fields.Url* fields
- **resource_class_args** (*list|tuple*) – args to be forwarded to the constructor of the resource.
- **resource_class_kwargs** (*dict*) – kwargs to be forwarded to the constructor of the resource.

Additional keyword arguments not specified above will be passed as-is to `flask.Flask.add_url_rule()`.

Examples:

```
namespace.add_resource(HelloWorld, '/', '/hello')
namespace.add_resource(Foo, '/foo', endpoint="foo")
namespace.add_resource(FooSpecial, '/special/foo', endpoint="foo")
```

as_list (*field*)
Allow to specify nested lists for documentation

clone (*name*, **specs*)
Clone a model (Duplicate all fields)

Parameters

- **name** (*str*) – the resulting model name
- **specs** – a list of models from which to clone the fields

See also:

`Model.clone()`

deprecated (*func*)
A decorator to mark a resource or a method as deprecated

doc (*shortcut=None*, ***kwargs*)
A decorator to add some api documentation to the decorated object

errorhandler (*exception*)
A decorator to register an error handler for a given exception

expect (**inputs*, ***kwargs*)
A decorator to Specify the expected input model

Parameters

- **inputs** (*ModelBase|Parse*) – An expect model or request parser
- **validate** (*bool*) – whether to perform validation or not

extend (*name*, *parent*, *fields*)
Extend a model (Duplicate all fields)

Deprecated since 0.9. Use *clone()* instead

header (*name*, *description=None*, ***kwargs*)
A decorator to specify one of the expected headers

Parameters

- **name** (*str*) – the HTTP header name
- **description** (*str*) – a description about the header

hide(*func*)
A decorator to hide a resource or a method from specifications

inherit(*name*, **specs*)
Inherit a model (use the Swagger composition pattern aka. allOf)

See also:

`Model.inherit()`

marshal(**args*, ***kwargs*)
A shortcut to the *marshal()* helper

marshal_list_with(*fields*, ***kwargs*)
A shortcut decorator for `marshal_with()` with `as_list=True`

marshal_with(*fields*, *as_list=False*, *code=<HTTPStatus.OK: 200>*, *description=None*, ***kwargs*)
A decorator specifying the fields to use for serialization.

Parameters

- **as_list** (*bool*) – Indicate that the return type is a list (for the documentation)
- **code** (*int*) – Optionally give the expected HTTP response code if its different from 200

model(*name=None*, *model=None*, *mask=None*, ***kwargs*)
Register a model

See also:

Model

param(*name*, *description=None*, *_in='query'*, ***kwargs*)
A decorator to specify one of the expected parameters

Parameters

- **name** (*str*) – the parameter name
- **description** (*str*) – a small description
- **_in** (*str*) – the parameter location *(query|header|formData|body|cookie)*

parser()
Instanciate a `RequestParser`

payload
Store the input payload in the current request context

produces(*mimetypes*)
A decorator to specify the MIME types the API can produce

response(*code*, *description*, *model=None*, ***kwargs*)
A decorator to specify one of the expected responses

Parameters

- **code** (*int*) – the HTTP status code
- **description** (*str*) – a small description about the response
- **model** (*ModelBase*) – an optional response model

route(**urls*, ***kwargs*)
A decorator to route resources.

schema_model(*name=None*, *schema=None*)
Register a model

See also:

Model

vendor(**args*, ***kwargs*)
A decorator to expose vendor extensions.

Extensions can be submitted as dict or kwargs. The `x-` prefix is optionnal and will be added if missing.

See: http://swagger.io/specification/#specification-extensions-128

class flask_restx.**Resource**(*api=None*, **args*, ***kwargs*)
Represents an abstract RESTX resource.

Concrete resources should extend from this class and expose methods for each supported HTTP method. If a resource is invoked with an unsupported HTTP method, the API will return a response with status 405 Method Not Allowed. Otherwise the appropriate method is called and passed all arguments from the url rule used when adding the resource to an Api instance. See `add_resource()` for details.

classmethod as_view(*name*, **class_args*, ***class_kwargs*)
Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the `View` on each request and call the *dispatch_request()* method on it.

The arguments passed to *as_view()* are forwarded to the constructor of the class.

dispatch_request(**args*, ***kwargs*)
Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

validate_payload(*func*)
Perform a payload validation on expected model if necessary

Models

class flask_restx.**Model**(*name*, **args*, ***kwargs*)
A thin wrapper on fields dict to store API doc metadata. Can also be used for response marshalling.

Parameters

- **name** (*str*) – The model public name
- **mask** (*str*) – an optional default model mask

All fields accept a `required` boolean and a `description` string in `kwargs`.

class flask_restx.fields.**Raw**(*default=None*, *attribute=None*, *title=None*, *description=None*, *required=None*, *readonly=None*, *example=None*, *mask=None*, ***kwargs*)
Raw provides a base field class from which others should extend. It applies no formatting by default, and should only be used in cases where data does not need to be formatted before being serialized. Fields should throw a *MarshallingError* in case of parsing problem.

Parameters

- **default** – The default value for the field, if no value is specified.

- **attribute** – If the public facing value differs from the internal value, use this to retrieve a different attribute from the response than the publicly named value.
- **title** (*str*) – The field title (for documentation purpose)
- **description** (*str*) – The field description (for documentation purpose)
- **required** (*bool*) – Is the field required ?
- **readonly** (*bool*) – Is the field read only ? (for documentation purpose)
- **example** – An optional data example (for documentation purpose)
- **mask** (*callable*) – An optional mask function to be applied to output

format(*value*)

Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

Parameters **value** – The value to format

Raises ***MarshallingError*** – In case of formatting problem

Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

output(*key*, *obj*, ***kwargs*)

Pulls the value for the given key from the object, applies the field's formatting and returns the result. If the key is not found in the object, returns the default value. Field classes that create values which do not require the existence of the key in the object should override this and return the desired value.

Raises ***MarshallingError*** – In case of formatting problem

class flask_restx.fields.**String**(**args*, ***kwargs*)

Marshal a value as a string. Uses `six.text_type` so values will be converted to `unicode` in python2 and `str` in python3.

format(*value*)

Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

Parameters **value** – The value to format

Raises ***MarshallingError*** – In case of formatting problem

Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

class flask_restx.fields.**FormattedString**(*src_str*, ***kwargs*)

FormattedString is used to interpolate other values from the response into this field. The syntax for the source string is the same as the string `format()` method from the python stdlib.

Ex:

```
fields = {
    'name': fields.String,
    'greeting': fields.FormattedString("Hello {name}")
```

(continues on next page)

(continued from previous page)

```
}
data = {
    'name': 'Doug',
}
marshal(data, fields)
```

Parameters **src_str** (*str*) – the string to format with the other values from the response.

output (*key*, *obj*, ***kwargs*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result. If the key is not found in the object, returns the default value. Field classes that create values which do not require the existence of the key in the object should override this and return the desired value.

Raises ***MarshallingError*** – In case of formatting problem

class flask_restx.fields.**Url**(*endpoint=None*, *absolute=False*, *scheme=None*, ***kwargs*)
A string representation of a Url

Parameters

- **endpoint** (*str*) – Endpoint name. If endpoint is `None`, `request.endpoint` is used instead
- **absolute** (*bool*) – If `True`, ensures that the generated urls will have the hostname included
- **scheme** (*str*) – URL scheme specifier (e.g. `http`, `https`)

output (*key*, *obj*, ***kwargs*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result. If the key is not found in the object, returns the default value. Field classes that create values which do not require the existence of the key in the object should override this and return the desired value.

Raises ***MarshallingError*** – In case of formatting problem

class flask_restx.fields.**DateTime**(*dt_format='iso8601'*, ***kwargs*)
Return a formatted datetime string in UTC. Supported formats are RFC 822 and ISO 8601.

See `email.utils.formatdate()` for more info on the RFC 822 format.

See `datetime.datetime.isoformat()` for more info on the ISO 8601 format.

Parameters **dt_format** (*str*) – `rfc822` or `iso8601`

format(*value*)
Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

Parameters **value** – The value to format

Raises ***MarshallingError*** – In case of formatting problem

Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

format_iso8601(*dt*)
Turn a datetime object into an ISO8601 formatted date.

Parameters **dt** (*datetime*) – The datetime to transform

Returns A ISO 8601 formatted date string

format_rfc822(*dt*)
Turn a datetime object into a formatted date.

Parameters **dt** (*datetime*) – The datetime to transform

Returns A RFC 822 formatted date string

class flask_restx.fields.**Date**(***kwargs*)
Return a formatted date string in UTC in ISO 8601.

See datetime.date.isoformat() for more info on the ISO 8601 format.

class flask_restx.fields.**Boolean**(*default=None*, *attribute=None*, *title=None*, *description=None*, *required=None*, *readonly=None*, *example=None*, *mask=None*, ***kwargs*)
Field for outputting a boolean value.

Empty collections such as `""`, `{}`, `[]`, etc. will be converted to `False`.

format(*value*)
Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

Parameters **value** – The value to format

Raises ***MarshallingError*** – In case of formatting problem

Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

class flask_restx.fields.**Integer**(**args*, ***kwargs*)
Field for outputting an integer value.

Parameters **default** (*int*) – The default value for the field, if no value is specified.

format(*value*)
Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

Parameters **value** – The value to format

Raises ***MarshallingError*** – In case of formatting problem

Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

class flask_restx.fields.**Float**(**args*, ***kwargs*)
A double as IEEE-754 double precision.

ex : 3.141592653589793 3.1415926535897933e-06 3.141592653589793e+24 nan inf -inf

format(*value*)
Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

Parameters **value** – The value to format

Raises ***MarshallingError*** – In case of formatting problem

Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

class flask_restx.fields.**Arbitrary**(**args*, ***kwargs*)

A floating point number with an arbitrary precision.

ex: 634271127864378216478362784632784678324.23432

format(*value*)

Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

Parameters **value** – The value to format

Raises ***MarshallingError*** – In case of formatting problem

Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

class flask_restx.fields.**Fixed**(*decimals=5*, ***kwargs*)

A decimal number with a fixed precision.

format(*value*)

Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

Parameters **value** – The value to format

Raises ***MarshallingError*** – In case of formatting problem

Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

class flask_restx.fields.**Nested**(*model*, *allow_null=False*, *skip_none=False*, *as_list=False*, ***kwargs*)

Allows you to nest one set of fields inside another. See *Nested Field* for more information

Parameters

- **model** (*dict*) – The model dictionary to nest
- **allow_null** (*bool*) – Whether to return None instead of a dictionary with null keys, if a nested dictionary has all-null keys
- **skip_none** (*bool*) – Optional key will be used to eliminate inner fields which value is None or the inner field's key not exist in data
- **kwargs** – If `default` keyword argument is present, a nested dictionary will be marshaled as its value if nested dictionary is all-null keys (e.g. lets you return an empty JSON object instead of null)

output (*key*, *obj*, *ordered=False*, ***kwargs*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result. If the key is not found in the object, returns the default value. Field classes that create values which do not require the existence of the key in the object should override this and return the desired value.

Raises ***MarshallingError*** – In case of formatting problem

class flask_restx.fields.**List**(*cls_or_instance*, ***kwargs*)
Field for marshalling lists of other fields.

See *List Field* for more information.

Parameters **cls_or_instance** – The field type the list will contain.

format (*value*)
Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

Parameters **value** – The value to format

Raises ***MarshallingError*** – In case of formatting problem

Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

output (*key*, *data*, *ordered=False*, ***kwargs*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result. If the key is not found in the object, returns the default value. Field classes that create values which do not require the existence of the key in the object should override this and return the desired value.

Raises ***MarshallingError*** – In case of formatting problem

class flask_restx.fields.**ClassName**(*dash=False*, ***kwargs*)
Return the serialized object class name as string.

Parameters **dash** (*bool*) – If *True*, transform CamelCase to kebab_case.

output (*key*, *obj*, ***kwargs*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result. If the key is not found in the object, returns the default value. Field classes that create values which do not require the existence of the key in the object should override this and return the desired value.

Raises ***MarshallingError*** – In case of formatting problem

class flask_restx.fields.**Polymorph**(*mapping*, *required=False*, ***kwargs*)
A Nested field handling inheritance.

Allows you to specify a mapping between Python classes and fields specifications.

```
mapping = {
    Child1: child1_fields,
    Child2: child2_fields,
}

fields = api.model('Thing', {
    owner: fields.Polymorph(mapping)
})
```

Parameters **mapping** (*dict*) – Maps classes to their model/fields representation

output(*key*, *obj*, *ordered=False*, ***kwargs*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result. If the key is not found in the object, returns the default value. Field classes that create values which do not require the existence of the key in the object should override this and return the desired value.

Raises ***MarshallingError*** – In case of formatting problem

resolve_ancestor(*models*)
Resolve the common ancestor for all models.

Assume there is only one common ancestor.

class flask_restx.fields.**Wildcard**(*cls_or_instance*, ***kwargs*)
Field for marshalling list of "unkown" fields.

Parameters **cls_or_instance** – The field type the list will contain.

output(*key*, *obj*, *ordered=False*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result. If the key is not found in the object, returns the default value. Field classes that create values which do not require the existence of the key in the object should override this and return the desired value.

Raises ***MarshallingError*** – In case of formatting problem

exception flask_restx.fields.**MarshallingError**(*underlying_exception*)
This is an encapsulating Exception in case of marshalling error.

Serialization

flask_restx.**marshal**(*data*, *fields*, *envelope=None*, *skip_none=False*, *mask=None*, *ordered=False*)
Takes raw data (in the form of a dict, list, object) and a dict of fields to output and filters the data based on those fields.

Parameters

- **data** – the actual object(s) from which the fields are taken from
- **fields** – a dict of whose keys will make up the final serialized response output
- **envelope** – optional key that will be used to envelop the serialized response
- **skip_none** (*bool*) – optional key will be used to eliminate fields which value is None or the field's key not exist in data
- **ordered** (*bool*) – Wether or not to preserve order

```
>>> from flask_restx import fields, marshal
>>> data = { 'a': 100, 'b': 'foo', 'c': None }
>>> mfields = { 'a': fields.Raw, 'c': fields.Raw, 'd': fields.Raw }
```

```
>>> marshal(data, mfields)
{'a': 100, 'c': None, 'd': None}
```

```
>>> marshal(data, mfields, envelope='data')
{'data': {'a': 100, 'c': None, 'd': None}}
```

```
>>> marshal(data, mfields, skip_none=True)
{'a': 100}
```

```
>>> marshal(data, mfields, ordered=True)
OrderedDict([('a', 100), ('c', None), ('d', None)])
```

```
>>> marshal(data, mfields, envelope='data', ordered=True)
OrderedDict([('data', OrderedDict([('a', 100), ('c', None), ('d', None)]))])
```

```
>>> marshal(data, mfields, skip_none=True, ordered=True)
OrderedDict([('a', 100)])
```

flask_restx.**marshal_with**(*fields*, *envelope=None*, *skip_none=False*, *mask=None*, *ordered=False*)

A decorator that apply marshalling to the return values of your methods.

```
>>> from flask_restx import fields, marshal_with
>>> mfields = { 'a': fields.Raw }
>>> @marshal_with(mfields)
... def get():
...     return { 'a': 100, 'b': 'foo' }
...
...
>>> get()
OrderedDict([('a', 100)])
```

```
>>> @marshal_with(mfields, envelope='data')
... def get():
...     return { 'a': 100, 'b': 'foo' }
...
...
>>> get()
OrderedDict([('data', OrderedDict([('a', 100)]))])
```

```
>>> mfields = { 'a': fields.Raw, 'c': fields.Raw, 'd': fields.Raw }
>>> @marshal_with(mfields, skip_none=True)
... def get():
...     return { 'a': 100, 'b': 'foo', 'c': None }
...
...
>>> get()
OrderedDict([('a', 100)])
```

see *flask_restx.marshal()*

flask_restx.**marshal_with_field**(*field*)

A decorator that formats the return values of your methods with a single field.

```
>>> from flask_restx import marshal_with_field, fields
>>> @marshal_with_field(fields.List(fields.Integer))
... def get():
...     return ['1', 2, 3.0]
...
>>> get()
[1, 2, 3]
```

see *flask_restx.marshal_with()*

class flask_restx.mask.**Mask**(*mask=None*, *skip=False*, ***kwargs*)

Hold a parsed mask.

Parameters

- **mask** (*str|dict|Mask*) – A mask, parsed or not
- **skip** (*bool*) – If `True`, missing fields won't appear in result

apply(*data*)
Apply a fields mask to the data.

Parameters **data** – The data or model to apply mask on

Raises ***MaskError*** – when unable to apply the mask

clean(*mask*)
Remove unnecessary characters

filter_data(*data*)
Handle the data filtering given a parsed mask

Parameters

- **data** (*dict*) – the raw data to filter
- **mask** (*list*) – a parsed mask to filter against
- **skip** (*bool*) – whether or not to skip missing fields

parse(*mask*)
Parse a fields mask. Expect something in the form:

```
{field,nested{nested_field,another},last}
```

External brackets are optionals so it can also be written:

```
field,nested{nested_field,another},last
```

All extras characters will be ignored.

Parameters **mask** (*str*) – the mask string to parse

Raises ***ParseError*** – when a mask is unparseable/invalid

flask_restx.mask.**apply**(*data*, *mask*, *skip=False*)
Apply a fields mask to the data.

Parameters

- **data** – The data or model to apply mask on
- **mask** (*str|Mask*) – the mask (parsed or not) to apply on data
- **skip** (*bool*) – If rue, missing field won't appear in result

Raises ***MaskError*** – when unable to apply the mask

Request parsing

class flask_restx.reqparse.**Argument**(*name*, *default=None*, *dest=None*, *required=False*, *ignore=False*, *type=<function <lambda>>*, *location=('json', 'values')*, *choices=()*, *action='store'*, *help=None*, *operators=('=',)*, *case_sensitive=True*, *store_missing=True*, *trim=False*, *nullable=True*)

Parameters

- **name** – Either a name or a list of option strings, e.g. foo or -f, –foo.
- **default** – The value produced if the argument is absent from the request.
- **dest** – The name of the attribute to be added to the object returned by `parse_args()`.
- **required** (*bool*) – Whether or not the argument may be omitted (optionals only).
- **action** (*string*) – The basic type of action to be taken when this argument is encountered in the request. Valid options are "store" and "append".
- **ignore** (*bool*) – Whether to ignore cases where the argument fails type conversion
- **type** – The type to which the request argument should be converted. If a type raises an exception, the message in the error will be returned in the response. Defaults to `unicode` in python2 and `str` in python3.
- **location** – The attributes of the `flask.Request` object to source the arguments from (ex: headers, args, etc.), can be an iterator. The last item listed takes precedence in the result set.
- **choices** – A container of the allowable values for the argument.
- **help** – A brief description of the argument, returned in the response when the argument is invalid. May optionally contain an "{error_msg}" interpolation token, which will be replaced with the text of the error raised by the type converter.
- **case_sensitive** (*bool*) – Whether argument values in the request are case sensitive or not (this will convert all values to lowercase)
- **store_missing** (*bool*) – Whether the arguments default value should be stored if the argument is missing from the request.
- **trim** (*bool*) – If enabled, trims whitespace around the argument.
- **nullable** (*bool*) – If enabled, allows null value in argument.

handle_validation_error(*error*, *bundle_errors*)

Called when an error is raised while parsing. Aborts the request with a 400 status and an error message

Parameters

- **error** – the error that was raised
- **bundle_errors** (*bool*) – do not abort when first error occurs, return a dict with the name of the argument and the error message to be bundled

parse(*request*, *bundle_errors=False*)

Parses argument value(s) from the request, converting according to the argument's type.

Parameters

- **request** – The flask request object to parse arguments from
- **bundle_errors** (*bool*) – do not abort when first error occurs, return a dict with the name of the argument and the error message to be bundled

source(*request*)

Pulls values off the request in the provided location :param request: The flask request object to parse arguments from

flask_restx.reqparse.**LOCATIONS = {'args': 'query', 'files': 'formData', 'form': 'formDat**

Maps Flask-RESTX RequestParser locations to Swagger ones

flask_restx.reqparse.**PY_TYPES = {<class 'int'>: 'integer', <class 'str'>: 'string', <clas**

Maps Python primitives types to Swagger ones

class flask_restx.reqparse.**ParseResult**
The default result container as an Object dict.

class flask_restx.reqparse.**RequestParser**(*argument_class=<class 'flask_restx.reqparse.Argument'>*, *result_class=<class 'flask_restx.reqparse.ParseResult'>*, *trim=False*, *bundle_errors=False*)
Enables adding and parsing of multiple arguments in the context of a single request. Ex:

```
from flask_restx import RequestParser

parser = RequestParser()
parser.add_argument('foo')
parser.add_argument('int_bar', type=int)
args = parser.parse_args()
```

Parameters

- **trim** (*bool*) – If enabled, trims whitespace on all arguments in this parser
- **bundle_errors** (*bool*) – If enabled, do not abort when first error occurs, return a dict with the name of the argument and the error message to be bundled and return all validation errors

add_argument(**args*, ***kwargs*)
Adds an argument to be parsed.

Accepts either a single instance of Argument or arguments to be passed into *Argument*'s constructor.

See *Argument*'s constructor for documentation on the available options.

copy()
Creates a copy of this RequestParser with the same set of arguments

parse_args(*req=None*, *strict=False*)
Parse all arguments from the provided request and return the results as a ParseResult

Parameters **strict** (*bool*) – if req includes args not in parser, throw 400 BadRequest exception

Returns the parsed results as *ParseResult* (or any class defined as `result_class`)

Return type *ParseResult*

remove_argument(*name*)
Remove the argument matching the given name.

replace_argument(*name*, **args*, ***kwargs*)
Replace the argument matching the given name with a new version.

Inputs

This module provide some helpers for advanced types parsing.

You can define you own parser using the same pattern:

```
def my_type(value):
    if not condition:
        raise ValueError('This is not my type')
    return parse(value)

# Swagger documentation
my_type.__schema__ = {'type': 'string', 'format': 'my-custom-format'}
```

The last line allows you to document properly the type in the Swagger documentation.

class flask_restx.inputs.**URL**(*check=False*, *ip=False*, *local=False*, *port=False*, *auth=False*, *schemes=None*, *domains=None*, *exclude=None*)

Validate an URL.

Example:

```
parser = reqparse.RequestParser()
parser.add_argument('url', type=inputs.URL(schemes=['http', 'https']))
```

Input to the `URL` argument will be rejected if it does not match an URL with specified constraints. If `check` is True it will also be rejected if the domain does not exists.

Parameters

- **check** (*bool*) – Check the domain exists (perform a DNS resolution)
- **ip** (*bool*) – Allow IP (both ipv4/ipv6) as domain
- **local** (*bool*) – Allow localhost (both string or ip) as domain
- **port** (*bool*) – Allow a port to be present
- **auth** (*bool*) – Allow authentication to be present
- **schemes** (*list|tuple*) – Restrict valid schemes to this list
- **domains** (*list|tuple*) – Restrict valid domains to this list
- **exclude** (*list|tuple*) – Exclude some domains

flask_restx.inputs.**boolean**(*value*)

Parse the string `"true"` or `"false"` as a boolean (case insensitive).

Also accepts `"1"` and `"0"` as `True`/`False` (respectively).

If the input is from the request JSON body, the type is already a native python boolean, and will be passed through without further parsing.

Raises **ValueError** – if the boolean value is invalid

flask_restx.inputs.**date**(*value*)

Parse a valid looking date in the format YYYY-mm-dd

flask_restx.inputs.**date_from_iso8601**(*value*)

Turns an ISO8601 formatted date into a date object.

Example:

```
inputs.date_from_iso8601("2012-01-01")
```

Parameters **value** (*str*) – The ISO8601-complying string to transform

Returns A date

Return type date

Raises **ValueError** – if value is an invalid date literal

flask_restx.inputs.**datetime_from_iso8601**(*value*)
Turns an ISO8601 formatted date into a datetime object.

Example:

```
inputs.datetime_from_iso8601("2012-01-01T23:30:00+02:00")
```

Parameters **value** (*str*) – The ISO8601-complying string to transform

Returns A datetime

Return type datetime

Raises **ValueError** – if value is an invalid date literal

flask_restx.inputs.**datetime_from_rfc822**(*value*)
Turns an RFC822 formatted date into a datetime object.

Example:

```
inputs.datetime_from_rfc822('Wed, 02 Oct 2002 08:00:00 EST')
```

Parameters **value** (*str*) – The RFC822-complying string to transform

Returns The parsed datetime

Return type datetime

Raises **ValueError** – if value is an invalid date literal

class flask_restx.inputs.**email**(*check=False*, *ip=False*, *local=False*, *domains=None*, *exclude=None*)
Validate an email.

Example:

```
parser = reqparse.RequestParser()
parser.add_argument('email', type=inputs.email(dns=True))
```

Input to the `email` argument will be rejected if it does not match an email and if domain does not exists.

Parameters

- **check** (*bool*) – Check the domain exists (perform a DNS resolution)
- **ip** (*bool*) – Allow IP (both ipv4/ipv6) as domain
- **local** (*bool*) – Allow localhost (both string or ip) as domain
- **domains** (*list|tuple*) – Restrict valid domains to this list
- **exclude** (*list|tuple*) – Exclude some domains

class flask_restx.inputs.**int_range**(*low*, *high*, *argument='argument'*)
Restrict input to an integer in a range (inclusive)

flask_restx.inputs.**ip**(*value*)
Validate an IP address (both IPv4 and IPv6)

flask_restx.inputs.**ipv4**(*value*)
Validate an IPv4 address

flask_restx.inputs.**ipv6**(*value*)
Validate an IPv6 address

flask_restx.inputs.**iso8601interval**(*value*, *argument='argument'*)
Parses ISO 8601-formatted datetime intervals into tuples of datetimes.

Accepts both a single date(time) or a full interval using either start/end or start/duration notation, with the following behavior:

- Intervals are defined as inclusive start, exclusive end
- Single datetimes are translated into the interval spanning the largest resolution not specified in the input value, up to the day.
- The smallest accepted resolution is 1 second.
- All timezones are accepted as values; returned datetimes are localized to UTC. Naive inputs and date inputs will are assumed UTC.

Examples:

```
"2013-01-01" -> datetime(2013, 1, 1), datetime(2013, 1, 2)
"2013-01-01T12" -> datetime(2013, 1, 1, 12), datetime(2013, 1, 1, 13)
"2013-01-01/2013-02-28" -> datetime(2013, 1, 1), datetime(2013, 2, 28)
"2013-01-01/P3D" -> datetime(2013, 1, 1), datetime(2013, 1, 4)
"2013-01-01T12:00/PT30M" -> datetime(2013, 1, 1, 12), datetime(2013, 1, 1, 12, 30)
"2013-01-01T06:00/2013-01-01T12:00" -> datetime(2013, 1, 1, 6), datetime(2013, 1,␣
↪1, 12)
```

Parameters **value** (*str*) – The ISO8601 date time as a string

Returns Two UTC datetimes, the start and the end of the specified interval

Return type A tuple (datetime, datetime)

Raises **ValueError** – if the interval is invalid.

flask_restx.inputs.**natural**(*value*, *argument='argument'*)
Restrict input type to the natural numbers (0, 1, 2, 3...)

flask_restx.inputs.**positive**(*value*, *argument='argument'*)
Restrict input type to the positive integers (1, 2, 3...)

class flask_restx.inputs.**regex**(*pattern*)
Validate a string based on a regular expression.

Example:

```
parser = reqparse.RequestParser()
parser.add_argument('example', type=inputs.regex('^[0-9]+$'))
```

Input to the `example` argument will be rejected if it contains anything but numbers.

Parameters **pattern** (*str*) – The regular expression the input must match

flask_restx.inputs.**url = <flask_restx.inputs.URL object>**
Validate an URL

Legacy validator, allows, auth, port, ip and local Only allows schemes 'http', 'https', 'ftp' and 'ftps'

Errors

flask_restx.errors.**abort**(*code=<HTTPStatus.INTERNAL_SERVER_ERROR: 500>*, *message=None*, ***kwargs*)

Properly abort the current request.

Raise a *HTTPException* for the given status *code*. Attach any keyword arguments to the exception for later processing.

Parameters

- **code** (*int*) – The associated HTTP status code
- **message** (*str*) – An optional details message
- **kwargs** – Any additional data to pass to the error payload

Raises **HTTPException** –

exception flask_restx.errors.**RestError**(*msg*)

Base class for all Flask-RESTX Errors

exception flask_restx.errors.**ValidationError**(*msg*)

A helper class for validation errors.

exception flask_restx.errors.**SpecsError**(*msg*)

A helper class for incoherent specifications.

exception flask_restx.fields.**MarshallingError**(*underlying_exception*)

This is an encapsulating Exception in case of marshalling error.

exception flask_restx.mask.**MaskError**(*msg*)

Raised when an error occurs on mask

exception flask_restx.mask.**ParseError**(*msg*)

Raised when the mask parsing failed

Schemas

This module give access to OpenAPI specifications schemas and allows to validate specs against them.

New in version 0.12.1.

class flask_restx.schemas.**LazySchema**(*filename*, *validator=<class 'jsonschema.validators.create.<locals>.Validator'>*)

A thin wrapper around schema file lazy loading the data on first access

Parameters

- **str** (*filename*) – The package relative json schema filename
- **validator** – The jsonschema validator class version

New in version 0.12.1.

validator

The jsonschema validator to validate against

flask_restx.schemas.**OAS_20 = <flask_restx.schemas.LazySchema object>**

OpenAPI 2.0 specification schema

exception flask_restx.schemas.**SchemaValidationError**(*msg*, *errors=None*)

Raised when specification is not valid

New in version 0.12.1.

flask_restx.schemas.**VERSIONS = {'2.0': <flask_restx.schemas.LazySchema object>}**
Map supported OpenAPI versions to their JSON schema

flask_restx.schemas.**validate**(*data*)
Validate an OpenAPI specification.

Supported OpenAPI versions: 2.0

Parameters **dict** (*data*) – The specification to validate

Returns boolean True if the specification is valid

Raises

- ***SchemaValidationError*** – when the specification is invalid
- ***flask_restx.errors.SpecsError*** – when it's not possible to determinate the schema to validate against

New in version 0.12.1.

Internals

These are internal classes or helpers. Most of the time you shouldn't have to deal directly with them.

class flask_restx.api.**SwaggerView**(*api=None*, **args*, ***kwargs*)
Render the Swagger specifications as JSON

class flask_restx.swagger.**Swagger**(*api*)
A Swagger documentation wrapper for an API instance.

class flask_restx.postman.**PostmanCollectionV1**(*api*, *swagger=False*)
Postman Collection (V1 format) serializer

flask_restx.utils.**merge**(*first*, *second*)
Recursively merges two dictionaries.

Second dictionary values will take precedence over those from the first one. Nested dictionaries are merged too.

Parameters

- **first** (*dict*) – The first dictionary
- **second** (*dict*) – The second dictionary

Returns the resulting merged dictionary

Return type dict

flask_restx.utils.**camel_to_dash**(*value*)
Transform a CamelCase string into a low_dashed one

Parameters **value** (*str*) – a CamelCase string to transform

Returns the low_dashed string

Return type str

flask_restx.utils.**default_id**(*resource*, *method*)
Default operation ID generator

flask_restx.utils.**not_none**(*data*)
Remove all keys where value is None

Parameters **data** (*dict*) – A dictionary with potentially some values set to None

Returns The same dictionary without the keys with values to `None`

Return type dict

`flask_restx.utils.`**`not_none_sorted`**(*data*)
Remove all keys where value is None

Parameters **`data`** (*`OrderedDict`*) – A dictionary with potentially some values set to None

Returns The same dictionary without the keys with values to `None`

Return type OrderedDict

`flask_restx.utils.`**`unpack`**(*response*, *default_code=<HTTPStatus.OK: 200>*)
Unpack a Flask standard response.

Flask response can be: - a single value - a 2-tuple `(value, code)` - a 3-tuple `(value, code, headers)`

Warning: When using this function, you must ensure that the tuple is not the response data. To do so, prefer returning list instead of tuple for listings.

Parameters

- **`response`** – A Flask style response
- **`default_code`** (*`int`*) – The HTTP code to use as default if none is provided

Returns a 3-tuple `(data, code, headers)`

Return type tuple

Raises **`ValueError`** – if the response does not have one of the expected format

3.13 Additional Notes

3.13.1 Contributing

flask-restx is open-source and very open to contributions.

If you're part of a corporation with an NDA, and you may require updating the license. See Updating Copyright below

Submitting issues

Issues are contributions in a way so don't hesitate to submit reports on the official bugtracker.

Provide as much informations as possible to specify the issues:

- the flask-restx version used
- a stacktrace
- installed applications list
- a code sample to reproduce the issue
- ...

Submitting patches (bugfix, features, . . .)

If you want to contribute some code:

1. fork the official flask-restx repository
2. Ensure an issue is opened for your feature or bug
3. create a branch with an explicit name (like `my-new-feature` or `issue-XX`)
4. do your work in it
5. Commit your changes. Ensure the commit message includes the issue. Also, if contributing from a corporation, be sure to add a comment with the Copyright information
6. rebase it on the master branch from the official repository (cleanup your history by performing an interactive rebase)
7. add your change to the changelog
8. submit your pull-request
9. 2 Maintainers should review the code for bugfix and features. 1 maintainer for minor changes (such as docs)
10. After review, a maintainer a will merge the PR. Maintainers should not merge their own PRs

There are some rules to follow:

- your contribution should be documented (if needed)
- your contribution should be tested and the test suite should pass successfully
- your code should be properly formatted (use `black .` to format)
- your contribution should support both Python 2 and 3 (use `tox` to test)

You need to install some dependencies to develop on flask-restx:

```
$ pip install -e .[dev]
```

An Invoke `tasks.py` is provided to simplify the common tasks:

```
$ inv -l
Available tasks:

  all      Run tests, reports and packaging
  assets   Fetch web assets -- Swagger. Requires NPM (see below)
  clean    Cleanup all build artifacts
  cover    Run tests suite with coverage
  demo     Run the demo
  dist     Package for distribution
  doc      Build the documentation
  qa       Run a quality report
  test     Run tests suite
  tox      Run tests against Python versions
```

To ensure everything is fine before submission, use `tox`. It will run the test suite on all the supported Python version and ensure the documentation is generating.

```
$ tox
```

You also need to ensure your code is compliant with the flask-restx coding standards:

```
$ inv qa
```

To ensure everything is fine before committing, you can launch the all in one command:

```
$ inv qa tox
```

It will ensure the code meet the coding conventions, runs on every version on python and the documentation is properly generating.

Running a local Swagger Server

For local development, you may wish to run a local server. running the following will install a swagger server

```
$ inv assets
```

NOTE: You'll need NPM installed to do this. If you're new to NPM, also check out nvm

Release process

The new releases are pushed on Pypi.org automatically from GitHub Actions when we add a new tag (unless the tests are failing).

In order to prepare a new release, you can use bumpr which automates a few things. You first need to install it, then run the `bumpr` command. You can then refer to the documentation for further details. For instance, you would run `bumpr -m` (replace `-m` with `-p` or `-M` depending the expected version).

Updating Copyright

If you're a part of a corporation with an NDA, you may be required to update the LICENSE file. This should be discussed and agreed upon by the project maintainers.

1. Check with your legal department first.
2. Add an appropriate line to the LICENSE file.
3. When making a commit, add the specific copyright notice.

Double check with your legal department about their regulations. Not all changes constitute new or unique work.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

U

V

W