Flask-RESTPlus Documentation

Release 0.3.0

Axel Haustant

December 14, 2014

Contents

Flask-RestPlus provide syntaxic suger, helpers and automatically generated Swagger documentation on top of Flask-Restful.

CHAPTER 1

Compatibility

flask-restplus requires Python 2.7+.

CHAPTER 2

Installation

You can install flask-restplus with pip:

```
$ pip install flask-restplus
```

or with easy_install:

```
$ easy_install flask-restplus
```

CHAPTER 3

Documentation

3.1 Quick start

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

```
from flask import Flask
from flask.ext.restplus import Api

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

of lazily with the factory pattern:

```
from flask import Flask
from flask.ext.restplus import Api

api = Api()

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

With Flask-Restplus, you only import the api instance to route and document your endpoints.

```
from flask import Flask
from flask.ext.restplus import Api, Resource, fields

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

@api.route('/somewhere')
class Somewhere(Resource):
    def get(self):
        return {}

    def post(self)
        api.abort(403)
```

3.2 Syntaxic sugar

One of the purpose of Flask-Restplus is to provide some syntaxic sugar of Flask-Restful.

3.2.1 Route with decorator

The Api class has a `route()` decorator used to route API's endpoint.

When with Flask-Restful you were writting :

```
class MyResource(Resource):
    def get(self, id):
        return {}

api.add_resource('/my-resource/<id>', MyResource.as_view('my-resource'))
```

With Flask-Restplus, you can write:

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

You can optionnaly provide class-wide documentation:

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

But it will be easier to read with two decorators for the same effect:

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

The namespace object provide the same feature:

```
ns = api.namespace('ns', 'Some namespace')

# Will ba available to /api/ns/my-resource/<id>
@ns.route('/my-resource/<id>', endpoint='my-resource')
class MyResource(Resource):
    def get(self, id):
        return {}
```

All routes within a namespace are prefixed with the namespace name.

3.2.2 `abort` shortcut

You can use the `Api.abort()` method to abort a request. This shortcut always serialize the response in the right format.

```
@api.route('/failure')
class MyResource(Resource):
    def get(self):
        api.abort(403)

    def post(self):
        api.abort(500, 'Some custom message')
```

3.2.3 `parser` shortcut

You can use the `Api.parser()` shortcut to obtain a RequestParser instance.

```
parser = api.parser()
parser.add_argument('param', type=str, help='Some parameter')
```

3.2.4 `marshal` shortcut

You can use the `Api.marshal()` shortcut to serialize your objects.

```
return api.marshal(todos, fields), 201
```

3.3 Documenting your API with Swagger

A Swagger API documentation is automatically generated and available on your API root but you need to provide some details with the `Api.doc()` decorator.

3.3.1 Documenting with the `Api.doc()` decorator

This decorator allows you specify some details about your API. They will be used in the Swagger API declarations.

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)
```

3.3.2 Documenting with the `Api.model()` decorator

The `Api.model` decorator allows you to declare the models that your API can serialize.

You can use it either on a fields dictionnary or a `field.Raw` subclass:

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

@api.model('MyField')
class MySpecialField(fields.Raw):
    pass

@api.model(type='integer', format='int64')
class MyIntField(fields.Raw):
    pass

@api.model(fields={'name': fields.String, 'age': fields.Integer})
```

```
class Person(fields.Raw):
    def format(self, value):
        return {'name': value.name, 'age': value.age}
```

3.3.3 Documenting with the `Api.marshal_with()` decorator

This decorator works like the Flask-Restful `marshal_with` decorator with the difference that it documents the methods. The optionnal parameter `as_list` allows you to specify wether 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)
    def post(self):
        return create_object()
```

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()
```

3.3.4 Documenting with the `Api.route()` decorator

You can provide class-wide documentation by using the `Api.route()`'s' `doc` parameter. It accept the same attribute/syntax than the `Api.doc()` decorator.

By example, these two declaration are equivalents:

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

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

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

3.3.5 Documenting the fields

Every Flask-Restplus fields accepts additional but 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`)

There is also field specific attributes.

The `String` field accept an optional `enum` argument to restrict the authorized values.

The `Integer`, `Float` and `Arbitrary` fields accept both `min` and `max` arguments to restrict the possible values.

```
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),
})
```

3.3.6 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 the Swagger unique `operationId` with the `id` documentation.

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

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

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

In the previous example, the default generated operationId will be `get_my_resource`

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

For each method, the path parameter are automatically extracted. You can provide additional parameters (from query parameters, body or form) or additionnal details on path parameters with the `params` documentation.

Input and output models

You can specify the serialized output model with the `model` documentation.

You can specify an input format for `POST` and `PUT` iwth the `body` documentation.

```
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 {}
```

You can't have body and form or file parameters at the same time, it will raise a SpecsError.

Models can 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(restplus.Resource):
    @api.doc(parser=parser)
    def get(self):
        return {}
```

3.3.7 Cascading

Documentation handling is done in cascade. Method documentation override class-wide documentation. Inherited documentation override parent one.

By example, these two declaration are equivalents:

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

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

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

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

3.3.8 Overriding the API root view

TODO

3.4 Full example

Here a full example extracted from Flask-Restful and ported to Flask-RestPlus.

```
from flask import Flask, redirect
from flask.ext.restplus import Api, Resource, fields

app = Flask(__name__)
api = Api(app, version='1.0', title='Todo API',
    description='A simple TODO API extracted from the original flask-restful example',
    prefix='/api'
)

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

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

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

listed_todo = api.model('ListedTodo', {
    'id': fields.String(required=True, description='The todo ID'),
    'todo': fields.Nested(todo, description='The Todo')
})


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

```
parser = api.parser()
parser.add_argument('task', type=str, required=True, help='The task details', location='form')


@ns.route('/<string:todo_id>')
@api.doc(responses={404: 'Todo not found'}, params={'todo_id': 'The Todo ID'})
class Todo(Resource):
    '''Show a single todo item and lets you delete them'''
    @api.doc(description='todo_id should be in {0}'.format(', '.join(TODOS.keys())))
    @api.marshal_with(todo)
    def get(self, todo_id):
        '''Fetch a given resource'''
        abort_if_todo_doesnt_exist(todo_id)
        return TODOS[todo_id]

    @api.doc(responses={204: 'Todo deleted'})
    def delete(self, todo_id):
        '''Delete a given resource'''
        abort_if_todo_doesnt_exist(todo_id)
        del TODOS[todo_id]
        return '', 204

    @api.doc(parser=parser)
    @api.marshal_with(todo)
    def put(self, todo_id):
        '''Update a given resource'''
        args = parser.parse_args()
        task = {'task': args['task']}
        TODOS[todo_id] = task
        return task


@ns.route('/')
class TodoList(Resource):
    '''Shows a list of all todos, and lets you POST to add new tasks'''
    @api.marshal_list_with(listed_todo)
    def get(self):
        '''List all todos'''
        return [{'id': id, 'todo': todo} for id, todo in TODOS.items()]

    @api.doc(parser=parser)
    @api.marshal_with(todo, code=201)
    def post(self):
        '''Create a todo'''
        args = parser.parse_args()
        todo_id = 'todo%d' % (len(TODOS) + 1)
        TODOS[todo_id] = {'task': args['task']}
        return TODOS[todo_id], 201


@app.route('/')
def redirect_to_api():
    '''
    Redirect on API until root API is supported by SwaggerJS/UI.

    See: https://github.com/swagger-api/swagger-js/issues/116
    '''
    return redirect(api.base_path)
```

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

You can find full examples in the github repository `examples` folder.

3.5 API

3.5.1 flask.ext.restplus

class flask_restplus.**Api**(*app=None*, *version=u'1.0'*, *title=None*, *description=None*, *terms_url=None*, *license=None*, *license_url=None*, *contact=None*, *contact_url=None*, *contact_email=None*, *endpoint=u'api'*, *prefix=None*, *authorizations=None*, *security=None*, *default=u'default'*, *default_label=u'Default namespace'*, ***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*) – the Flask application object
- **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)
- **prefix** (*str*) – Prefix all routes with a value, eg v1 or 2010-04-01
- **default_mediatype** (*str*) – The default media type to return
- **decorators** (*list*) – Decorators to attach to every resource
- **catch_all_404s** (*bool*) – Use `handle_error()` to handle 404 errors throughout your app

- **url_part_order** – A string that controls the order that the pieces of the url are concatenated when the full url is constructed. 'b' is the blueprint (or blueprint registration) prefix, 'a' is the api prefix, and 'e' is the path component the endpoint is added with
- **errors** (*dict*) – A dictionary to define a custom response for each exception or error raised during a request
- **authorizations** (*dict*) – A Swagger Authorizations declaration as dictionary

abort(*code=500*, *message=None*, ***kwargs*)
Properly abort the current request

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

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

doc(***kwargs*)
Add some api documentation to the decorated object

marshal(*data*, *fields*)
A shortcut to the `marshal` helper

marshal_list_with(*fields*, *code=200*)
A shortcut decorator for `marshal_with(as_list=True, code=code)`

marshal_with(*fields*, *as_list=False*, *code=200*, ***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** (*integer*) – Optionnaly give the expected HTTP response code if its different from 200

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

Model can be either a dictionnary or a fields.Raw subclass.

owns_endpoint(*endpoint*)
Override the default implementation as there is always a Blueprint

parser()
Instanciate a RequestParser

render_ui()
Override this method to customize the documentation page

flask_restplus.**marshal**(*data*, *fields*)
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

- **fields** – a dict of whose keys will make up the final serialized response output
- **data** – the actual object(s) from which the fields are taken from

```
>>> from flask.ext.restful import fields, marshal
>>> data = { 'a': 100, 'b': 'foo' }
>>> mfields = { 'a': fields.Raw }
```

```
>>> marshal(data, mfields)
OrderedDict([('a', 100)])
```

class flask_restplus.**marshal_with**(*fields*)
A decorator that apply marshalling to the return values of your methods.

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

see `flask.ext.restful.marshal()`

flask_restplus.**abort**(*http_status_code*, ***kwargs*)
Raise a HTTPException for the given http_status_code. Attach any keyword arguments to the exception for later processing.

exception flask_restplus.**RestException**(*msg*)
Base class for all Flask-Restplus Exceptions

exception flask_restplus.**SpecsError**(*msg*)
An helper class for incoherent specifications.

exception flask_restplus.**ValidationError**(*msg*)
An helper class for validation errors.

3.5.2 flask.ext.restplus.fields

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

3.5.3 flask.ext.restplus.reqparse

3.6 Changelog

3.6.1 0.3.0

- **Switch to Swagger 2.0 (Major breakage)**
  - `notes` documentation is now `description`
  - `nickname` documentation is now `id`
  - new responses declaration format
- Added missing `body` parameter to document `body` input
- Last release before Flask-Restful 0.3+ compatibility switch

3.6.2 0.2.4

- Handle `description` and `required` attributes on `fields.List`

3.6.3 0.2.3

- Fix custom fields registeration

3.6.4 0.2.2

- Fix model list in declaration

3.6.5 0.2.1

- Allow to type custom fields with `Api.model`
- Handle custom fields into `fieds.List`

3.6.6 0.2

- Upgraded to SwaggerUI 0.2.22
- Support additional field documentation attributes: `required`, `description`, `enum`, `min`, `max` and `default`
- Initial support for model in RequestParser

3.6.7 0.1.3

- Fix `Api.marshal()` shortcut

3.6.8 0.1.2

- Added `Api.marshal_with()` and `Api.marshal_list_with()` decorators
- Added `Api.marshal()` shortcut

3.6.9 0.1.1

- Use `zip_safe=False` for proper packaging.

3.6.10 0.1

- Initial release

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

f

A

D

F

M

O

P

R

S

V