Flask-RESTPlus Documentation

Release 0.8.1

Axel Haustant

November 27, 2015

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(app)
```

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)
```

3.1.1 Swagger UI

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

```
from flask import Flask, Blueprint
from flask.ext.restplus 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/'
```

If you need a custom UI, you can register a custom view function with the `@api.documentation` decorator:

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

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

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

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
```

3.2.5 Handle errors with `@api.errorhandler()` decorator

The `@api.errorhandler()` decorator allows you to register a specific handler for a given exception, in the same maner than you can do with Flask/Blueprint `@errorhandler` decorator.

```
@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}
```

It also allow to override the default error handler when used wihtout parameter:

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

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 also extend fields and use the `__schema_format__`, `__schema_type__` and `__schema_example__` to specify the produced types and examples:

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

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'
```

Duplicating with `api.extend`

The `api.extend` method allows you to register an augmented model. It saves you duplicating all fields.

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

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

Polymorphism with `api.inherit`

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

Will produce the following Swagger definitions:

```
"Parent": {
    "properties": {
        "name": {"type": "string"},
        "class": {"type": "string"}
    },
    "discriminator": "class",
    "required": ["class"]
},
"Child": {
    "allOf": [{
            "$ref": "#/definitions/Parent"
        }, {
            "properties": {
                "extra": {"type": "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)
})
```

3.3.3 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 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()
```

3.3.4 Documenting with the `@api.expect()` decorator

The `@api.expect()` decorator allows you to specify the expected input fields and is a shortcut for `@api.doc(body=<fields>)`. It accepts an optional boolean parameter `validate` defining wether or not the payload should be validated. The validation behavior can be customized globally by either setting the `RESTPLUS_VALIDATE` configuration to True or passing `validate=True` to the API constructor.

The following syntaxes are equivalents:

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

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

```
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
```

It allows you specify lists as expected input too:

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

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

An exemple of on-demand validation:

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

@api.route('/my-resource/<id>')
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 exemple of application-wide validation by config:

```
app.config['RESTPLUS_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 exemple 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
```

3.3.5 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 synatxes are equivalents:

```
@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 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
```

If you use the `@api.marshal_with()` decorator, it automatically document 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
```

At least, 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
```

3.3.6 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 {}
```

3.3.7 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`)

- `example`: an example to use when displaying (*default*: `None`)

There is also field specific attributes.

The `String` field accept the following optionnal arguments:

- `enum`: an array restricting the authorized values.
- `min_length`: the minimum length expected
- `max_length`: the maximum length expected
- `pattern`: a RegExp pattern the string need to validate

The `Integer`, `Float` and `Arbitrary` fields accept the following optionnal 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 also accept the `min`, `max`, ``exclusiveMin`` and `exclusiveMax` optionnal arguments but they should be date or datetime (either as 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),
})
```

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

```
@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 will be `get_my_resource`

You can override the default operationId generator by giving a callable as `default_id` parameter to your API. This callable will receive two positional arguments:

- the resource class name
- this lower cased HTTP method

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

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

In the previous example, the generated operationId will 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

For each method, the path parameter are automatically extracted. You can provide additional parameters (from query parameters, body or form) or additional 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` with 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 {}
```

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
```

Headers

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

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

3.3.9 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 {}
```

3.3.10 Marking as deprecated

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

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

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

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

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

3.3.11 Hiding from documentation

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

```
# 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):
    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 {}
```

3.3.12 Documenting autorizations

In order to document an authorization you can provide an authorization dictionary to the API constructor:

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

Next, you need to set the authorization documentation on each resource/method requiring it. You can use a decorator to make it easier:

```
def apikey(func):
    return api.doc(security='apikey')(func)

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

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

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

```
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',
        'scopes': {
            'read': 'Grant read-only access',
            'write': 'Grant read-write access',
        }
    }
}
api = Api(self.app, security=['apikey', {'oauth2': 'read'}], authorizations=authorizations)
```

And compose/override them at method level:

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

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

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

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

3.4 Fields masks

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

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

3.4.1 Syntax

The syntax is actually quite simple. You just provide a coma separated list of field names, optionaly 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}}'
```

3.4.2 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 permet to expose a global header once so it can make your Swagger specifications a lot more verbose. You can disable this behavior by setting `RESTPLUS_MASK_SWAGGER` to `False`.

3.5 Swagger UI documentation

By default `flask-restplus` provide a Swagger UI documentation on your API root.

```
from flask import Flask
from flask.ext.restplus 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>', 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)


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

If you run the code below and visit your API root URL (http://localhost:5000) you will have an automatically generated SwaggerUI documentation.

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

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

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

api = Api(app, version='1.0', title='Sample API',
```

```
    description='A sample API',
)

'...'

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

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.ext.restplus import Api, Resource, fields

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

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

'...'

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

You can totally disable the generated Swagger UI by setting `doc=False`:

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

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

'...'

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

You can also provide a custom UI by reusing the apidoc blueprint or rolling your own from scratch.

```
from flask import Flask, Blueprint, url_for
from flask.ext.restplus import API, apidoc

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

'...'

@api.documentation
def swagger_ui():
    return apidoc.ui_for(api)


app.register_blueprint(blueprint)
```

3.6 Exporting

Flask-restplus provide facilities to export your API.

3.6.1 Export as Swagger specifications

You can export the Swagger specifiifcations corresponding to your API.

```
from flask import json

from myapp import api

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

3.6.2 Export as Postman collection

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))
```

3.7 Full example

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

```
from flask import Flask
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',
)

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
```

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

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

3.8 API

3.8.1 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*, *authorizations=None*, *security=None*, *doc=u'/'*, *default_id=<function default_id>*, *default=u'default'*, *default_label=u'Default namespace'*, *validate=None*, *tags=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.
- **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

- **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

deprecated(*func*)
Mark a resource or a method as deprecated

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

errorhandler(*exception*)
Register an error handler for a given exception

expect(*body*, *validate=None*)
Specify the expected input model

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

header(*name*, *description=None*, ***kwargs*)
Specify one of the expected headers

inherit(*name*, *parent*, *fields*)
Inherit a modal (use the Swagger composition pattern aka. allOf)

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)

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

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

marshal_with(*fields*, *as_list=False*, *code=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** (*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 dictionary or a fields. Raw subclass.

parser()
Instanciate a RequestParser

render_doc()
Override this method to customize the documentation page

response(*code*, *description*, *model=None*, ***kwargs*)
Specify one of the expected responses

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

flask_restplus.**marshal**(*data*, *fields*, *envelope=None*)
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

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

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

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

class flask_restplus.**marshal_with**(*fields*, *envelope=None*)
A decorator that apply marshalling to the return values of your methods.

```
>>> from flask_restplus 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)]))])
```

see `flask_restful.marshal()`

class `flask_restplus.`**`marshal_with_field`**(*field*)
A decorator that formats the return values of your methods with a single field.

```
>>> from flask_restplus 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_restful.marshal_with()`

`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.

3.8.2 flask.ext.restplus.fields

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

3.8.3 flask.ext.restplus.reqparse

3.8.4 flask.ext.restplus.inputs

3.8.5 flask.ext.restplus.mask

class `flask_restplus.mask.`**`Nested`**(*name*, *fields*)

`fields`
Alias for field number 1

`name`
Alias for field number 0

exception `flask_restplus.mask.`**`ParseError`**
Raise when the mask parsing failed

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

flask_restplus.mask.**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.

3.9 Changelog

3.9.1 0.8.1 (2015-11-27)

- **Refactor Swagger UI handling:**
  - allow to register a custom view with `@api.documentation`
  - allow to register a custom URL with the `doc` parameter
  - allow to disable documentation with `doc=False`
- Added fields mask support through header (see: Fields Masks Documentation)
- Expose `flask_restful.inputs` module on `flask_restplus.inputs`
- **Added support for some missing fields and attributes:**
  - `host` root field (filed only if `SERVER_NAME` config is set)
  - custom `tags` root field
  - `exclusiveMinimum` and `exclusiveMaximum` number field attributes
  - `multipleOf` number field attribute
  - `minLength` and `maxLength` string field attributes
  - `pattern` string field attribute
  - `minItems` and `maxItems` list field attributes
  - `uniqueItems` list field attribute
- Allow to override the default error handler
- Fixes

3.9.2 0.8.0

- Added payload validation (initial implementation based on jsonschema)
- Added `@api.deprecated` to mark resources or methods as deprecated
- Added `@api.header` decorator shortcut to document headers
- Added Postman export
- Fix compatibility with flask-restful 0.3.4

- Allow to specify an exemple a custom fields with `__schema_example__`
- Added support for `PATCH` method in Swagger UI
- Upgraded to Swagger UI 2.1.2
- Handle enum as callable
- Allow to configure `docExpansion` with the `SWAGGER_UI_DOC_EXPANSION` parameter

3.9.3 0.7.2

- Compatibility with flask-restful 0.3.3
- Fix action=append handling in RequestParser
- Upgraded to SwaggerUI 2.1.8-M1
- Miscellaneous fixes

3.9.4 0.7.1

- Fix `@api.marshal_with_list()` keyword arguments handling.

3.9.5 0.7.0

- Expose models and fields schema through the `__schema__` attribute
- Drop support for model as class
- Added `@api.errorhandler()` to register custom error handlers
- Added `@api.response''` shortcut decorator
- Fix list nested models missing in definitions

3.9.6 0.6.0

- Python 2.6 support
- **Experimental polymorphism support (single inheritance only)**
  - Added `Polymorph` field
  - Added `discriminator` attribute support on `String` fields
  - Added `api.inherit()` method
- Added `ClassName` field

3.9.7 0.5.1

- Fix for parameter with schema (do not set type=string)

3.9.8 0.5.0

- Allow shorter syntax to set operation id: `@api.doc('my-operation')`
- Added a shortcut to specify the expected input model: `@api.expect(my_fields)`
- Added `title` attribute to fields
- Added `@api.extend()` to extend models
- Ensure coherence between `required` and `allow_null` for `NestedField`
- Support list of primitive types and list of models as body
- Upgraded to latest version of Swagger UI
- Fixes

3.9.9 0.4.2

- Rename apidoc blueprint into restplus_doc to avoid collisions

3.9.10 0.4.1

- Added `SWAGGER_VALIDATOR_URL` config parameter
- Added `readonly` field parameter
- Upgraded to latest version of Swagger UI

3.9.11 0.4.0

- Port to Flask-Restful 0.3+
- Use the default Blueprint/App mecanism
- Allow to hide some ressources or methods using `@api.doc(False)` or `@api.hide`
- Allow to globally customize the default operationId with the `default_id` callable parameter

3.9.12 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

3.9.13 0.2.4

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

3.9.14 0.2.3

- Fix custom fields registeration

3.9.15 0.2.2

- Fix model list in declaration

3.9.16 0.2.1

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

3.9.17 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

3.9.18 0.1.3

- Fix `Api.marshal()` shortcut

3.9.19 0.1.2

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

3.9.20 0.1.1

- Use `zip_safe=False` for proper packaging.

3.9.21 0.1

- Initial release

3.10 Contributing

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

3.10.1 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-restplus version used
- a stacktrace
- installed applications list
- a code sample to reproduce the issue
- ...

3.10.2 Submitting patches (bugfix, features, ...)

If you want to contribute some code:

1. fork the official flask-restplus repository
2. create a branch with an explicit name (like `my-new-feature` or `issue-XX`)
3. do your work in it
4. rebase it on the master branch from the official repository (cleanup your history by performing an interactive rebase)
5. submit your pull-request

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 mostly PEP8 compatible with a 120 characters line length
- your contribution should support both Python 2 and 3 (use `tox` to test)

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

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

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

```
$ inv -l
Available tasks:

  all     Run tests, reports and packaging
  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-restplus coding standards:

```
$ inv qa
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

f