

**План**

1. Application factory
2. Flask Restful Api
3. Argument parsing
4. Data formatting
5. Resource Method Decorators
6. Define custom errors
7. Use with Blueprints

**Application factory**

В зависимости от передаваемого Config приложение будет раниться с другими параметрами.

Это позволит безболезненно переключаться между dev, prod, и test конфигурациями.



**from flask import** Flask

**from flask\_restful import** Resource, Api

app = Flask( name ) api = Api(app)

**class HelloWorld**(Resource): **def** get(self):

**return** {'hello': 'world’} api.add\_resource(HelloWorld, '/’)

**if** name == ' main ’: app.run(debug=**True**)

**class Todo1**(Resource): **def** get(self):

*# Default to 200 OK*

**return** {'task': 'Hello world’}

**class Todo2**(Resource): **def** get(self): *#*

*Set the response code to 201*

**return** {'task': 'Hello world'}, 201

**class Todo3**(Resource): **def** get(self):

*# Set the response code to 201 and return custom headers*

**return** {'task': 'Hello world'}, 201, {'Etag': 'some-opaque-string'}

**from flask\_restful import** reqparse

parser = reqparse.RequestParser()

parser.add\_argument('rate', type=int, help='Rate to charge for this resource’) parser.add\_argument('name’)

args = parser.parse\_args()

Если передать в parce\_args значение strict=True, то оно будет проверять и вызывать ошибку если пользователь передаст аргумент который вы не ожидаете получить.

args = parser.parse\_args(strict=**True**)

Если вы укажете значение **help**, оно будет отображаться как сообщение об ошибке при возникновении ошибки. Если вы не укажете **help**, то оно вернет сатнартную ошибку из FLASK RESTFUL API.

По умолчанию аргументы не обязательны. Кроме того, аргументы, предоставленные в запросе, которые не являются частью RequestParser, будут игнорироваться.

Также обратите внимание: Аргументы, объявленные в вашем анализаторе запросов, но не заданные в самом запросе, по умолчанию будут равны None.

parser.add\_argument('name', required=**True**, help="Name cannot be blank!")

Для того что бы значения с одинаковыми именами записывались в список нужно добавить параметр action='append'

parser.add\_argument('name', action='append’)

curl [http://api.example.com](http://api.example.com/) -d "name=bob" -d "name=sue" -d "name=joe” args = parser.parse\_args() args['name'] *# ['bob', 'sue', 'joe']*

Для того что бы значения с одинаковыми именами записывались в список нужно добавить параметр action='append'

parser.add\_argument('name', action='append’)

curl [http://api.example.com](http://api.example.com/) -d "name=bob" -d "name=sue" -d "name=joe” args = parser.parse\_args() args['name'] *# ['bob', 'sue', 'joe']*

По дефолту RequestParser() ищет значения в request.args, для того что бы изменить или добавить дополнительную локацию поиска значений можно использовать параметр **location**, для передачи нескольких значений используется список **location=['headers', 'values'].**

*# 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')

Также вы можете наследовать один парсер от другого для того что бы избегать повторяющегося кода.

**from flask\_restful import** reqparse

parser = reqparse.RequestParser() parser.add\_argument('foo', type=int) parser\_copy = parser**.copy()**

parser\_copy.add\_argument('bar', type=int)

*# parser\_copy has both 'foo' and 'bar’*

parser\_copy.**replace\_argument**('foo', required=**True**, location='json’)

*# 'foo' is now a required str located in json, not an int as defined # by original parser*

parser\_copy.**remove\_argument**('foo’)

*# parser\_copy no longer has 'foo' argument*

# Argument parsing

По умолчанию RequstParser() свалится завершит приложение на первой же ошибке, и не выведет вам остальных даже если такие имеются. Для того что бы сразу отобразить все имеющиеся ошибки нужно передать параметр **bundle\_errors=True**

**from flask\_restful import** reqparse

parser = reqparse.RequestParser(bundle\_errors=**True**) parser.add\_argument('foo', type=int, required=**True**) parser.add\_argument('bar', type=int, required=**True**)

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

{

"message": {

"foo": "foo error message", "bar": "bar error message"

}

}

*# The default behavior would only return the first error*

parser = RequestParser()

parser.add\_argument('foo', type=int, required=**True**) parser.add\_argument('bar', type=int, required=**True**)

{

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

}

# Argument parsing

По умолчанию RequstParser() свалится завершит приложение на первой же ошибке, и не выведет вам остальных даже если такие имеются. Для того что бы сразу отобразить все имеющиеся ошибки нужно передать параметр **bundle\_errors=True**

**from flask\_restful import** reqparse

parser = reqparse.RequestParser(bundle\_errors=**True**) parser.add\_argument('foo', type=int, required=**True**) parser.add\_argument('bar', type=int, required=**True**)

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

{

"message": {

"foo": "foo error message", "bar": "bar error message"

}

}

*# The default behavior would only return the first error*

parser = RequestParser()

parser.add\_argument('foo', type=int, required=**True**) parser.add\_argument('bar', type=int, required=**True**)

{

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

}

По умолчанию вы можете возвращать в вашем методе стандартные Python структуры данных, но часто вы будете встречаться с потребностью вернуть объект. Чтобы решить эту проблему, Flask-RESTful предоставляет модуль fields и декоратор marshal\_with ().

**from flask\_restful import** fields, marshal\_with

resource\_fields = { 'task': fields.String, 'uri': fields.Url('todo\_ep’) }

**class TodoDao**(object):

**def** init (self, todo\_id, task): self.todo\_id = todo\_id self.task = task

*# This field will not be sent in the response*

self.status = 'active’

**class Todo**(Resource): **@marshal\_with**(resource\_fields) **def** get(self, \*\*kwargs):

**return** TodoDao(todo\_id='my\_todo', task='Remember the milk')

Часто бывает так что название публичного поля не совпадает с тем что у вас используется в обьекте, для того что бы их с мапить используем attribute.

model = {

'name': fields.String(attribute='private\_name’), 'address': fields.String,

}

Для того что бы задавать дефолтное значение нужно использовать default model = {

'name': fields.String(default='Anonymous User’),

'address': fields.String,

}

**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 ={

'name': fields.String,

'priority': UrgentItem(attribute='flags’), 'status': UnreadItem(attribute='flags’),

}

**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 ={

'name': fields.String,

'priority': UrgentItem(attribute='flags’), 'status': UnreadItem(attribute='flags’),

}

Вы можете создавать плоскую структуру которую marshall преобразует во вложенную

**>>> from flask\_restful import** fields, marshal

## >>> import json

>>>

**>>>** resource\_fields = {'name': fields.String}

**>>>** resource\_fields['address'] = {}

**>>>** resource\_fields['address']['line 1'] = fields.String(attribute='addr1’)

**>>>** resource\_fields['address']['line 2'] = fields.String(attribute='addr2’)

**>>>** resource\_fields['address']['city'] = fields.String

**>>>** resource\_fields['address']['state'] = fields.String

**>>>** resource\_fields['address']['zip'] = fields.String

**>>>** data = {'name': 'bob', 'addr1': '123 fake street', 'addr2': '', 'city': 'New York', 'state': 'NY', '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"}}'

Вы можете обрабатывать списки

**>>> from flask\_restful import** fields, marshal

## >>> import json

>>>

**>>>** resource\_fields = {'name': fields.String, 'first\_names': fields.List(fields.String)}

**>>>** data = {'name': 'Bougnazal', 'first\_names' : ['Emile', 'Raoul’]}

**>>>** json.dumps(marshal(data, resource\_fields))

**>>>** '{"first\_names": ["Emile", "Raoul"], "name": "Bougnazal"}'

# Data Formatting

Для правильного отображения вложенной структуры можно использовать **Nested()** field тип.

**>>> from flask\_restful import** fields, marshal

## >>> import json

>>>

**>>>** address\_fields = {}

**>>>** address\_fields['line 1'] = fields.String(attribute='addr1’)

**>>>** address\_fields['line 2'] = fields.String(attribute='addr2’)

**>>>** address\_fields['city'] = fields.String(attribute='city’)

**>>>** address\_fields['state'] = fields.String(attribute='state’)

**>>>** address\_fields['zip'] = fields.String(attribute='zip’)

>>>

**>>>** resource\_fields = {}

**>>>** resource\_fields['name'] = fields.String

**>>>** resource\_fields['billing\_address'] = fields.Nested(address\_fields)

**>>>** resource\_fields['shipping\_address'] = fields.Nested(address\_fields)

**>>>** address1 = {'addr1': '123 fake street', 'city': 'New York', 'state': 'NY', 'zip': '10468’}

**>>>** address2 = {'addr1': '555 nowhere', 'city': 'New York', 'state': 'NY', 'zip': '10468’}

**>>>** data = { 'name': 'bob', 'billing\_address': address1, 'shipping\_address': address2}

>>>

**>>>** json.dumps(marshal\_with(data, resource\_fields)) '{"billing\_address": {"line 1": "123 fake street", "line 2": null, "state": "NY", "zip": "10468", "city": "New York"},

"name": "bob", "shipping\_address": {"line 1": "555 nowhere", "line 2": null, "state":

"NY", "zip": "10468", "city": "New York"}}'

В классе Resource есть свойство с именем method\_decorators. Вы можете создать подкласс Resource и добавить свои собственные декораторы, которые будут добавлены ко всем функциям метода в ресурсе.

def cache(f): @wraps(f)

def cacher(\*args, \*\*kwargs): # caching stuff

return cacher

class MyResource(restful.Resource): method\_decorators = {'get': [cache]}

def get(self, \*args, \*\*kwargs):

return something\_interesting(\*args, \*\*kwargs)

def post(self, \*args, \*\*kwargs):

return create\_something(\*args, \*\*kwargs)

Вы можете задать свои ошибки которые будет возвращать api сервис по дефолту.

errors = { 'UserAlreadyExistsError': {

'message': "A user with that username already exists.", 'status': 409,

},

'ResourceDoesNotExist': {

'message': "A resource with that ID no longer exists.", 'status': 410,

'extra': "Any extra information you want.",

},

}

app = Flask( name )

api = flask\_restful.Api(app, errors=errors)

from **flask** import Flask, Blueprint

from **flask\_restful** import Api, Resource, url\_for

app = Flask( name )

api\_bp = Blueprint('api', name ) api = Api(api\_bp)

class **TodoItem**(Resource): def get(self, id):

return {'task': 'Say "Hello, World!"'}

api.add\_resource(TodoItem, '/todos/<int:id>') app.register\_blueprint(api\_bp)