

**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.



Разработчику, тестировщику и при развертывании приложения на продакшин один и тот же код требует использовать разные конфики.

Для того, чтобы каждый раз не менять код используется ТН Application factory.

1. Cоздаем новый проект Flask

По умолчанию pycharm создает файл app.py

и 2 папки static and templates

2. Cоздаем модуль main.

Создаем python\_package with name - main. ( в папке по умолчанию создается файл \_\_init\_\_.py).

Создаем в файле \_\_init\_\_.py blueprint.

from flask import Blueprint

main = Blueprint("main", \_\_name\_\_)

@main.route("/")

def hello\_from\_main():

return 'Hello from main'

3. Регистрация нашего blueprint in app.py:

from flask import Flask

from main import main

app = Flask(\_\_name\_\_)

app.register\_blueprint(main)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

4. Задаем конфиг в файле app.py для подключения к базе данных.

app.config("TEST\_DB\_connection") = "psql..//"

В этом случае при каждом изменении конфигов - для девелоперов, тестировщиков и т. д. Нужно менять код в app.py, что создает неудобства.

Во избежание этого используюется application factory.

5. Создаем файл config.py

class Config:

TEST\_DB\_connection = "psql..//"

В нем создаем класс Config - в котором подписуем конфиг подключения к данной базе.

В файле app.py:

from flask import Flask

from main import main

from config import Config импортируем класс с конфигом

app = Flask(\_\_name\_\_)

app.config.from\_object(Config) указываем, что к данному приложению идет конфиг от объекта Config

app.register\_blueprint(main)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

6. Создаем несколько конфигурационных классов в файле config.py.

6.1. Создание нескольких конфигурационных классов, которые наследуются от одного класса.

6.2. Создание функции, которая в зависимости от значения глобальной переменной возвращает тот или иной кофигурационный класс. При этом код нигде не меняется. А меняется только глобальная переменная

Значение глобально переменной задается — в консоли через export или либо в конфигурации pycharm можно задавать ее значение в поле FLASK\_ENV

import os

class Config:

TEST\_DB\_CONNECTION = "psql..//"

DEFAULT = "default value"

class DevConfig(Config):

TEST\_DB\_CONNECTION = "DEVpsql..//"

class TestConfig(Config):

TEST\_DB\_CONNECTION = "TESTpsql..//"

def run\_config():

env = os.environ.get("ENV"):

if env == "DEV":

return DevConfig

elif env == "TEST":

return TestConfig

else:

return Config

os.environ - Значение **os.environ** известно как объект мэппинга (сопоставления), который работает со словарем переменных пользовательской среды. Возможно вы не знали, но каждый раз, когда вы пользуетесь своим компьютером, некоторые переменные среды уже установлены.

7. Передаем значение функции run\_config to app.py.

from flask import Flask

from main import main

from config import run\_config

app = Flask(\_\_name\_\_)

app.config.from\_object(run\_config())

app.register\_blueprint(main)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

8. Передаем в blueprint main значение

импортируме объект current\_app, который позволяет получить доступ к конфигурациям данного приложения.

from flask import Blueprint, current\_app

main = Blueprint("main", \_\_name\_\_)

@main.route("/")

def hello\_from\_main():

print(current\_app.config)

return current\_app.config.get("TEST\_DB\_CONNECTION")

теперь домашняя страница возвращает значение соответствующего конфига при любом значении глобалной переменной env.

<http://jsonviewer.stack.hu/>

?????В консоли приложение не запускается flask run,

export ENV=DEV - не задается глобальная переменная

pip list ,

**Flask-RESTful API**

Архитектура rest использует стандарты HTTP

Обмен идет в формате json ( от сервиса к сервису). Данные подаются на клиентское приложение (angular, react …) для отображения web pages.

CRUD — create, read, update, delete.

1) Импорт библиотеки restful.

Для создания приложения с помощью rest используется библиотека flask\_restful — pip install flask\_restful

2) Cоздание 1го endopoint in rest:

**from flask\_restful import** Api

API — объект, который оборачивает наше приложение.

api = Api(app)

Теперь вместо декораторов @app.route будем использовать классы, которые наследуются от класса Resource:

Класс Resource нужно импортировать также из flask\_restful и он уже содержит в себе все методы ( get, delete, post..)

Использование @app.route также возможно, но это усложняет чтение, поддержку кода.

Исползование rest делает код более читаемым и лаконичным.

Render\_templates - обычно в rest не используется.

2.1 Оборачиваем наше приложение:

api = Api(app)

2.2.Создаем класс, кот наследуется от Resource

2.3.Указываем метод get, post, delete - что он должен возвращать.

class HelloRest(Resource):

def get(self):

return {"key": "value"}

2.4. Прописываем url для данного класса.

api.add\_resource(HelloRest, "/")

from flask import Flask

from flask\_restful import Api, Resource

from config import run\_config

app = Flask(\_\_name\_\_)

app.config.from\_object(run\_config())

api = Api(app)

class HelloRest(Resource):

def get(self):

return {"key1": "value"}

def post(self):

return "post"

api.add\_resource(HelloRest, "/")

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

**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**)

**Кроме самих данных, функция может возвращать** **Список кодов состояния HTTP ( по умолчанию 200 ОК), а также значение для headers.**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| **Тело**  **Код состояния**  **Header** | Тело  Header1  Header2 | **Тело**  **Тело**  **Код состояния (тело)**  **Тело** |  |

**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'}

Написать Апи сервис для списка самых больших компаний Мира.

Apple

Amazon

Alphabet

Microsoft

a = [“Apple”, “Amazon”, “Alphabet”, “Microsoft”]

Реализовать возможность удалять значения. Добавлять значения, менять их местами и выводить весь список.

from flask import Flask, request

from flask\_restful import Api, Resource

from config import run\_config

app = Flask(\_\_name\_\_)

app.config.from\_object(run\_config())

api = Api(app)

class HelloRest(Resource):

def get(self):

return {"key1": "value"}, 200, {"customs\_header": "header\_value"}

def post(self):

return "post"

a = ["Apple", "Amazon", "Alphabet", "Microsoft"]

class Companies(Resource):

def get(self):

response = dict()

for i, element in enumerate(a):

response[i+1] = element

return response

def post(self, value): # элемент добавляется через url.

a.append(value)

return "Element added successfully"

def put(self):

# request.data - данные, которые будут перед-ся в запросе в виде json dict with keys company and positon.

# данные передаются in body of request ( see postman)

import json

data = json.loads(request.data) # превращаем json в словарь.

company = data.get("company")

position = data.get("position") - 1

a.remove(company)

a.insert(position, company)

return "List updated successfully"

def delete(self, value):

a.remove(value)

return "Element deleted successfully"

api.add\_resource(HelloRest, "/")

api.add\_resource(Companies, "/companies", "/companies/<value>")

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

Делая запрос пользователь может передавать информацию в аргументах. С помощью объекта reqparse можно валидировать и работать с аргументами. Задавать ошибки если не тот тип данных.

**1) Import reqparse:**

**from flask\_restful import** reqparse

**2) Регистрация нашего парсера:**

**parser\_company = reqparse.RequestParser() - экземпляр класс RequestParser() из модуля reqparse**

позволяет парсить (валидировать множественные аргументы в контексте единичного запроса).

Имеет метод add\_argument(), который позволяе добавлять элементы для парсинга.

**Parser\_company.add\_argument(“page”, type=int)**

**3) Объявление парсинга в нашем url:**

**args= parser\_company.parse\_args() # можно указать strict=True при вводе лишний аргументов будет вызываться ошибка.**

from flask import Flask, request

from flask\_restful import Api, Resource, reqparse

from config import run\_config

app = Flask(\_\_name\_\_)

app.config.from\_object(run\_config())

api = Api(app)

class HelloRest(Resource):

def get(self):

return {"key1": "value"}, 200, {"customs\_header": "header\_value"}

def post(self):

return "post"

a = ["Apple", "Amazon", "Alphabet", "Microsoft"]

parse\_company = reqparse.RequestParser()

parse\_company.add\_argument("page", type=int)

class Companies(Resource):

def get(self):

args = parse\_company.parse\_args(strict=True)

response = dict()

print(args)

for i, element in enumerate(a):

response[i+1] = element

return response

def post(self, value): # элемент добавляется через url.

a.append(value)

return "Element added successfully"

def put(self):

# request.data - данные, которые будут перед-ся в запросе в виде json dict with keys company and positon.

# данные передаются in body of request ( see postman)

import json

data = json.loads(request.data) # превращаем json в словарь.

company = data.get("company")

position = data.get("position") - 1

a.remove(company)

a.insert(position, company)

return "List updated successfully"

def delete(self, value):

a.remove(value)

return "Element deleted successfully"

api.add\_resource(HelloRest, "/")

api.add\_resource(Companies, "/companies", "/companies/<value>")

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

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" }

}

from flask import Flask, request

from flask\_restful import Api, Resource, reqparse

from config import run\_config

app = Flask(\_\_name\_\_)

app.config.from\_object(run\_config())

api = Api(app)

class HelloRest(Resource):

def get(self):

return {"key1": "value"}, 200, {"customs\_header": "header\_value"}

def post(self):

return "post"

a = ["Apple", "Amazon", "Alphabet", "Microsoft"]

parser = reqparse.RequestParser(bundle\_errors=True)

parser.add\_argument("page", type=int, help="wrong page")

# сюда можно добавлять location ( откуда парсер будет брать аргументы на проверку.

# Наследование от parser:

# можно добавлять новые аргументы, удалять из родительского или их менять.

parser\_2 = parser.copy()

parser\_2.add\_argument("value")

parser.add\_argument("chapter", type=int, help="wrong chapter")

class Companies(Resource):

def get(self):

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

response = dict()

print(args)

for i, element in enumerate(a):

response[i+1] = element

return response

def post(self, value): # элемент добавляется через url.

a.append(value)

return "Element added successfully"

def put(self):

# request.data - данные, которые будут перед-ся в запросе в виде json dict with keys company and positon.

# данные передаются in body of request ( see postman)

import json

data = json.loads(request.data) # превращаем json в словарь.

company = data.get("company")

position = data.get("position") - 1

a.remove(company)

a.insert(position, company)

return "List updated successfully"

def delete(self, value):

a.remove(value)

return "Element deleted successfully"

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 ().

1) Cоздаем класс class Fish:

def \_\_init\_\_(self, name, size):

self.name = name

self.size = size

экземпляр который мы хотим передать на фронтенд.

Пусть это будет fish.

2) Cоздаем класс GetFish, который наследуем от Resource и прописываем метод get для передачи на фронтенд при соответствующем запросе:

class GetFish(Resource):

def get(self):

carp = Fish("carp", "200gr")

return carp

3) Прописываем данный класс с указанием соответствующего url.

api.add\_resource(GetFish, "/fishes")

Если зайти на url, то сервер выдаст ошибку,

raise TypeError(f'Object of type {o.\_\_class\_\_.\_\_name\_\_} '

TypeError: Object of type Fish is not JSON serializable

так как кастомный класс так просто не может быть передан. Его необходимо форматировать (сериализовать).

Для этого нужно:

1) Импортировать из restful - fields поля для описания класса и декораторо marshal\_with.

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

2) Создать структуру, описывающую класc (поля, кот. мы хотим передать).

structure\_fish = {

"name": fields.String,

"size": fields.String

3) Обернуть нашу функцию-запрос get декоратором marshal\_with,

добавив структуру нашего класса fish в качестве аргумента.

class GetFish(Resource):

@marshal\_with(structure\_fish)

def get(self):

carp = Fish("carp", "200gr")

return carp

декоратор marshal\_with сериализует в объект в json ( только те поля, которые указаны в fish structure).

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.

structure\_fish = {

"name": fields.String,

"size": fields.String,

"public": fields.String(attribute="\_super\_private"),

"default": fields.String(default="river fish")

}

# чтобы скрыть название закрытого метода ( в структуре указывается его имя для публики, а в атрибуте настоящее имя поля)

# также можно задавать дефолтное значения, если токого поля в экземпляре класса не будет.

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,

}

**Cоздание собственных видов полей:**

**Для этого нужно создать свой класс, наследник от родительского fields.Raw**

**и переопределить в нем метод format.**

**class Upper(fields.Raw):**

def format(self, value):

return value.upper()

structure\_fish = {

"name": Upper,

"size": fields.String,

"public": fields.String(attribute="\_super\_private"),

"default": fields.String(default="river fish")

**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’),

}

Вывод полей класса, которые сериализует @marshal\_with определяется заданной структурой. Поэтому из плоской структуры полей класса, можно вывести вложенную структуру json на фронтенд, а также передать значения полей ввиде списка:

class House:

def \_\_init\_\_(self, owner, country, city, street, number):

self.owner = owner

self.country = country

self.city = city

self.street = street

self.number = number

self.residents = []

structure\_house = {

"owner": fields.String,

"address": {

"country": fields.String,

"city": fields.String,

"street": fields.String,

"number": fields.Integer},

"residents": fields.List(fields.String)

}

class GetHouse(Resource):

@marshal\_with(structure\_house)

def get(self):

my\_house = House("PVV", "Ukraine", "Kharkiv", "Grushevskogo", 5)

my\_house.residents = ["me", "my cat"]

return my\_house

api.add\_resource(GetHouse, "/house")

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"}'

Если поля объекта являются словарями:

необходимо создать структуру вложенного поля,

в структуре класса указать тип поля nested и в него вложить структуру вложенного поля.

structure\_family = {

"mother": fields.String,

"father": fields.String

}

"family": fields.Nested(structure\_family)

# 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)