# Hola mundo

from flask import Flask  
  
app= Flask(\_\_name\_\_)  
  
@app.route('/')  
def hello():  
 return "Hola mundo!"

mas correcto

from flask import Flask  
  
app= Flask(\_\_name\_\_)  
  
@app.route('/')  
def hello():  
 return "Hola mundo!"  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run()

o

if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True)

o

if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(host='0.0.0.0'**,** port=**3000,**debug=True)

# comandos

//ejecutar servidor (NOMBRE DEL ARCHIVO py que contiene app sin poner la extencion [main.py])

> flask --app main run

> flask --app NOMBRE\_CON\_APP\_ARCHVIO run

// ejecucion modo debug (muestra los errores y se actualiza solo)

> flask --app main --debug run

# Renderizar html puro

@app.route(‘/html’)  
def hello\_html():  
 return “””  
 <html>  
 <body>  
 <h1>Hello World!</h1>  
 </body>  
 </html>  
 “””

# Múltiples direcciones

@app.route('/uno')  
@app.route('/dos')  
def hello():  
 return "Hola mundo!"

# Parámetros en ruta

@app.route('/parametro/<elparametro>')  
def con\_parametro(elparametro):  
 return f"Hola mundo! {elparametro}"

## Tipados

string , int , float , path , uuid  
@app.route('/parametro/<string:elparametro>')  
def con\_parametro(elparametro):  
 return f"Hola mundo! {elparametro}"

### path

tener cuidado con este tipo de dato porque ejecuta código

[GET] ruta…/parámetro/<script>alert(“hola”)</script>

@app.route('/parametro/<path:elparametro>')  
def con\_parametro(elparametro):  
 return f"Hola mundo! {elparametro}"

Para evitarlo se formatea el parámetro

from markupsafe import escape

@app.route('/parametro/<path:elparametro>')  
def con\_parametro(elparametro):  
 return f"Hola mundo! { escape(elparametro)}"

## multiples

@app.route('/parametro/<string:elparametro>/<int:elparametro2>')  
def con\_parametro(elparametro):  
 return f"Hola mundo! {elparametro} {elparametro2}"

## Opcionales

@app.route('/parametro/<string:elparametro>')

@app.route('/parametro/<string:elparametro>/<int:elparametro2>')  
def con\_parametro(elparametro,elparametro2=None**,**):  
 if elparametro2==None:  
 elparametro2="nada"

return f"Hola mundo! {elparametro} {elparametro2}"

# Renderizar plantillas

## /templates/

Ubicar aquí las plantillas, por defecto es la ruta en la que busca (al mismo nivel del main)

## render\_template

from flask import Flask**,** render\_template

@app.route('/')  
def con\_plantilla():  
 return render\_template("pagina1.html")

### Jinja2

En esencia es exactamente lo mismo que django, por lo tanto no lo voy a por todo

* La herencia de plantillas es igual
* Las condicionales y bucles son iguales
* Los filtros son iguales

#### Variables

La plantilla en Jinja2

<p>{{variable1}}</p>  
<p>{{variable2}}</p>

@app.route('/')  
def con\_plantilla():  
 return render\_template("pagina1.html"**,** variable1="valor1"**,** variable2="valor2")

#### Interpretar expresiones sencillas

<p>{{variable3\*2+4}}</p>

#### Variables en un dict

<p>{{variable1}}</p>  
<p>{{variable2}}</p>

<p>{{variable3\*2+4}}</p>

@app.route('/dict')  
def con\_plantilla\_dict():  
 data={  
 "variable1":"valor1dic"**,** "variable2" : "valor2dic"**,** "variable3" : **5** }  
 return render\_template("pagina1.html"**,** \*\*data)

#### Lista

<p>{{lista[0]}}</p>  
<p>{{lista[1]}}</p>  
<p>{{lista[2]}}</p>

@app.route('/lista')  
def con\_plantilla\_lista():  
 return render\_template("lista.html"**,** lista=["valor1"**,** "valor2"**,** "valor3"])

#### Pasar una variable dict

<p>{{variable\_dict["key"]}}</p>

@app.route('/dict2')  
def con\_plantilla\_dict2():  
 return render\_template("dict.html"**,** variable\_dict={"key":"valor"}  
 )

#### Pasar un objeto

class Persona:  
 def \_\_init\_\_(self**,** nombre**,**edad):  
 self.nombre=nombre  
 self.edad=edad

<p>{{variable\_clase.nombre}}</p>  
<p>{{variable\_clase.edad}}</p>

@app.route('/clase')  
def con\_plantilla\_clase():  
 return render\_template("persona.html"**,** variable\_clase=Persona(  
 nombre="Julio"**,**edad=**2** )  
 )

#### If elif else

{% if variable == 1 %}

<p> era 1 </p>

{% elif variable == 2 %}  
 <p> era 2 </p>  
 {% else %}  
 <p> no lo era</p>  
{% endif %}

#### For

{% for elemento in lista %}  
 <p>{{ elemento }}</p>  
{% endfor %}

##### For dict

{% for key, valor in variable\_dict.items() %}  
 <p>{{ key }}</p>  
 <p>{{ valor }}</p>  
{% endfor %}

#### Filtros personalizados app.add\_template\_filter

<p>{{variable1|filtronombre}}</p>

@app.add\_template\_filter  
def filtronombre(valor):  
 return f"tu nombre es {valor}"

Otra manera de registrar un filtro

<p>{{variable1|filtronombre}}</p>

def filtronombre2(valor):  
 return f"tu nombre es {valor}"  
app.add\_template\_filter(filtronombre2**,**'filtronombreenplantilla')

#### pasar y usar funciones

<p>{{lafuncion("valorarepetir",5)}}</p>

def repeat(s**,**n):  
 return s\*n  
  
@app.route('/confuncion')  
def con\_plantilla\_confuncion():  
 return render\_template("confuncion.html"**,** lafuncion=repeat  
 )

###### funciones globales app.add\_template\_global

no es necesario pasarla a las plantillas porque ya están presentes

<p>{{funcionglobal("valorarepetir",5)}}</p>

@app.add\_template\_global

def funcionglobal(s**,**n):  
 return s\*n  
  
@app.route('/confuncion')  
def con\_plantilla\_confuncion():  
 return render\_template("confuncion.html")

#### Declarar variables en la plantilla

{% set name = "Nuevo nombre" %}  
<p>el : {{name}}</p>

Macros

Son con trozos de código tipo funciones reutilizables y declarables en la plantilla

{% macro lista\_macro(parametro1,parametro2) %}  
  
 <h1>{{parametro1}}</h1>  
 <ul>  
 {% for elemento in parametro2 %}  
 <li>{{elemento}}</li>  
 {% endfor %}  
 </ul>  
{% endmacro %}  
  
{{ lista\_macro("mi lista 1", ["a","b","d"])}}  
  
{{ lista\_macro("mi lista 2", ["z","x","c"])}}

##### Macros.jinja2

Se pueden crear estas macros en archivos y luego importarlos en nuestra plantilla

/templates/macros.jinja2

{% macro lista\_macro(parametro1,parametro2) %}  
  
 <h1>{{parametro1}}</h1>  
 <ul>  
 {% for elemento in parametro2 %}  
 <li>{{elemento}}</li>  
 {% endfor %}  
 </ul>  
{% endmacro %}

/templates/miplantilla.html

{% from "macros.jinja2" import lista\_macro %}

{{ lista\_macro("mi lista 1", ["a","b","d"])}}  
  
{{ lista\_macro("mi lista 2", ["z","x","c"])}}

#### Include

Se puede cargar directamete un archivo que contenga código de plantillas para reutilizar un pedazo, distinto de la herencia que tienes que heredar todo

{% incluede “miarchivo.html” %}

Plantilla.jinja2

Estos archivos son como plantillas pero por defecto interpretan las variables como html

/templates/ plantilla.jinja2

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
 {{variable\_html}}  
</body>  
</html>

/miapp.py

@app.route('/conhtml')  
def con\_plantilla\_html():  
 return render\_template("plantilla.jinja2"**,** variable\_html=""""  
 <h1> Esto esta en html </h1>  
 """  
 )

## Archivos estáticos

### /static/

Poner en esta carpeta los archivos

### Plantilla

/static/css/micss.css

… href=”{{ url\_for(‘static’,filename=’ css/micss.css’) }}”

# url\_for

obtener una ruta a la vista de una función, no incluye la base de la url

ruta=url\_for("funcion\_vista")

## tiene parámetros la ruta

ruta=url\_for("funcion\_vista",nombre\_parametro1=”valor”,nombre\_parametro2=”valor2”)

## usar en plantillas

es una función global asi que se puede usar en las plantillas

<a href=”{{ url\_for("funcion\_vista") }} />

# POST

La idea es usar el objeto request global para acceder a los datos de un formulario

<p>{{contenido}}</p>  
<form class="form" method="POST">  
 <label for="entry" class="form-label" >Nombre:</label>  
 <input id="entry" name="contenido" class="form-submit">  
 <button type="submit">Enviar</button>  
</form>

from flask import Flask**,** render\_template**,** request

@app.route('/formulario'**,**methods=["GET"**,**"POST"])  
def con\_plantilla\_formularios():  
 contenido="vacio"  
 if request.method=="POST":  
 contenido=request.form.get("contenido")  
 return render\_template("formularios.html"**,** contenido=contenido  
 )

flask\_wtf y wtforms

Para crear formularios parecidos a django

* pip install flask-wtf

from flask\_wtf import FlaskForm  
from wtforms import StringField**,** TextAreaField**,**PasswordField**,**SubmitField  
from wtforms.validators import DataRequired**,**Length

-1ro crear una clave secreta

app.config.from\_mapping(SECRET\_KEY='misecretkey')

-2do crear un formulario

class RegisterForm(FlaskForm):  
 username=StringField("Nombre de Usuario"**,**validators=[DataRequired()**,**Length(min=**4,**max=**25**)])  
 password=PasswordField("Password"**,**validators=[DataRequired()**,**Length(min=**4,**max=**25**)])  
 submit=SubmitField("Registrar")

-3ro preparar su uso y pasarlo a la Plantilla

@app.route('/usuarioregister'**,**methods=["GET"**,**"POST"])  
def con\_plantilla\_formularios2():  
 form=RegisterForm()  
 if form.validate\_on\_submit():  
 username=form.username.data  
 password = form.password.data  
 print("fue post")  
 return render\_template("formusuario.html"**,**form=form)

-4to crear el formulario en la platilla

<form class="form" method="POST">  
 {{ form.hidden\_tag() }}

{{form.username.label}}{{form.username}}  
 {{form.password.label}}{{form.password}}  
 {{from.submit}}  
</form>

{% if form.errors %}  
 <div style="color: red">  
 {{form.errors}}  
 </div>  
{% endif %}

# Sesiones

from flask import session

app.config.from\_mapping(SECRET\_KEY='misecretkey')

session['key']=valor

variable=session.get("key")

session.pop('key'**,**None)

# encriptar

* pip install passlib

from passlib.hash import pbkdf2\_sha256

encriptada=pbkdf2\_sha256.hash(password)

verificar=pbkdf2\_sha256.verify(password**,**encriptada)

# mongoDB

instalar la librería

* pip install pymongo[srv]

## conectarse

* 1ro crear la conexión

from pymongo import MongoClient

client = MongoClient("mongodb://localhost:27017")

* 2do agregar la conexión a la app  
  app.db=client.prueba
* 3ro realizar consultas  
  usuarios= [usuario for usuario in app.db.usuarios.find({})]
* 4to insertar un valor

usuario={"nombre":contenido}  
app.db.usuarios.insert\_one(usuario)

# Entorno

* pip install python-dotenv

-1ro crear el .env

-2do cargar las variables

import os

MONGODB\_URI =os.getenv("MONGODB\_URI")

# Crear un comando

@app.cli.command()  
def micomando():

…

Luego

flask --app main micomando

# Test

* pip install flask-testing

-1ro tener una carpeta tests

-2do crear el comando que corre los tests

import unittest

@app.cli.command()  
def test():  
 tests=unittest.TestLoader().discover("tests")  
 unittest.TextTestRunner().run(tests)

> flask --app main test

## Test básico

Los archivos de test deben de comenzar con la palabra test y a su ves sus clases de test deberían tambien comenzar con esta palabra

Al crear un test debe de heredar de ‘TestCase’ y debe de sobrescribir a create\_app para configurar la app a testing, y se debe desactivar la comprobación crf de los formularios porque en los ambientes de test no se genera los crf

from flask\_testing import TestCase  
from flask import current\_app**,**url\_for  
from main import app  
class TestApp(TestCase):  
 def create\_app(self):  
 app.config['TESTING'] = True  
 app.config['WTF\_CRF\_ENABLED'] = False  
 return app  
 def test\_app\_exists(self):  
 self.assertTrue(current\_app is not None)

## test redirect

def test\_index\_redirect(self):  
 response = self.client.get(url\_for('redirigir'))  
 esperado=url\_for("con\_plantilla\_confuncion")

self.assertEqual(response.location**,** esperado)

## test 200

def test\_200(self):  
 response = self.client.get(url\_for('con\_plantilla\_confuncion'))  
 self.assert200(response)

# SQLAlchemy

Es un ORM

* pip install -U Flask-SQLAlchemy

## Sqlite

from flask\_sqlalchemy import SQLAlchemy

-1ro confiurar la app

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///db.sqlite'  
app.config['SQLALCHEMY\_TRAK\_MODIFICATIONS'] = True

-2do crear la instancia de SQLAlchemy  
db=SQLAlchemy(app)

-3ro crear las tablas

class Equipo(db.Model):  
 id=db.Column(db.Integer**,** primary\_key=True)  
 nombre=db.Column(db.String(**100**))  
 apellido=db.Column(db.String(**100**))  
  
 def \_\_repr\_\_(self):  
 return '<Equipo %r>' % self.nombre

-4to crear la bd

db.create\_all()

## commit

cualquier cambio a los datos no se vera reflejado hasta que se llame al commit

db.session.commit()

## add

equipo=Equipo(  
 nombre="el nombre"**,** apellido="Espa"  
)  
db.session.add(equipo)  
db.session.commit()

## get all

equipos=Equipo.query.all()

## filtro

usando filter\_by

equipos=Equipo.query.filter\_by(apellido="apelli 2")

### filter

o

usando filter, existe <=,>=,==,!=

equipos=Equipo.query.filter(Equipo.apellido=="apelli 2")

#### ¡=

equipos=Equipo.query.filter(Equipo.apellido!="apelli 2")

#### in

equipos=Equipo.query.filter(Equipo.apellido.in\_(["apelli 2",”apellid 3”]))

#### like

empresa=Empresa.query.filter(Empresa.nombre.like(‘%texto%’))

#### ~

Opuesto, se pone delante del filtro

empresa=Empresa.query.filter(~Empresa.nombre.like(‘%texto%’))

#### None

Comparar con None

equipos=Equipo.query.filter(Equipo.apellido!=None)

#### count

cantidad=Equipo.query.filter(Equipo.apellido!=None).count()

#### order by

posts=Post.query.order\_by(Post.fecha.desc()).all()

## first

Equipo.query.filter\_by(apellido="apelli 2").first()

## Editar

Para editar solo hay que llamar al commit y se aplica los cambios de todas las instancias actuales

Instancia\_equipo1.nombre=”editado”

Instancia\_equipo2.nombre=”editado otro”

db.session.commit()

## delete

db.session.delete(instancia)

db.session.commit()

## ForengenKey

Hay que crear en lado muchos (el que tiene le foregenkey) el campo numero id (foregenkey) y un campo dinamico que se va a autocompletar con el objeto de esa relacion (Relationship)

class Empresa(db.Model):  
 id = db.Column(db.Integer**,** primary\_key=True)  
 nombre = db.Column(db.String(**100**))

class Equipo(db.Model):  
 id=db.Column(db.Integer**,** primary\_key=True)  
 nombre=db.Column(db.String(**100**))  
 apellido=db.Column(db.String(**100**))  
 empresa\_id=db.Column(db.Integer**,** db.ForeignKey('empresa.id'))  
 empresa=db.relationship('Empresa'**,** backref=db.backref('empresas'**,**lazy=True))

## ManyToMany

Hay que crear un tabla intermedia donde se relacionan los ids de ambas tablas

Luego en la tabla que va a tener el many hay que crear una columna Relationship que va a apuntar a la tabla intermedia y va a tener un argumento backref que va a ser el nombre de la relacion inversa para la tabla que no tiene el many

class Marca(db.Model):  
 id = db.Column(db.Integer**,** primary\_key=True)  
 nombre = db.Column(db.String(**100**))  
  
marca\_empresa=db.Table('marca\_empresa'**,** db.Column('marca\_id'**,**db.Integer**,**db.ForeignKey('marca.id')**,**primary\_key=True)**,** db.Column('empresa\_id'**,**db.Integer**,**db.ForeignKey('empresa.id')**,**primary\_key=True)**,** )  
  
class Empresa(db.Model):  
 id = db.Column(db.Integer**,** primary\_key=True)  
 nombre = db.Column(db.String(**100**))  
 marcas =db.relationship('marca\_empresa'**,**secondary=marca\_empresa**,**backref=db.backref('empresas'**,**lazy=True))

## Migraciones

* pip install Flask-Migrate

1do> flask --app main db init

inicializa el directorio de migraciones [vacio] [una sola vez, almacena el orden entre las migraciones]

2ro> flask --app main db migrate

crear el archivo con la descripcion de la bd [repetir cada vez que se realize un cambio]

3to> flask --app main db upgrade

aplica los cambios

4to> flask --app main db downgrade

revierte la ultima migracion [casi nunca se utliza]

## \_\_tablename\_\_

Si se quiere se puede especificar la tabla si tiene otro nombre (que igual generalmente se crean con los nombres en minúsculas a pesar de que las clases tienen el nombre en mayúsculas)

class Equipo(db.Model):  
 \_\_tablename\_\_ = 'equipo'  
 id=db.Column(db.Integer**,** primary\_key=True)

## nullable

class Equipo(db.Model):  
 \_\_tablename\_\_ = 'equipo'  
 id=db.Column(db.Integer**,** primary\_key=True)  
 nombre=db.Column(db.String(**100**)**,**nullable=True)

## DateTime

class Equipo(db.Model):  
 \_\_tablename\_\_ = 'equipo'  
 id=db.Column(db.Integer**,** primary\_key=True)  
 nombre=db.Column(db.String(**100**)**,**nullable=True)  
 creado=db.Column(db.DateTime**,**default=datetime.now)

o utc

creado=db.Column(db.DateTime()**,**default=datetime.utcnow)

# Blueprint

Es como ponerle un prefijo a toda la api, como una dirección base

El orden de los siguientes pasos es importante

from flask import Flask **,**Blueprint

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

apiBLprint=Blueprint("api"**,**\_\_name\_\_**,**url\_prefix="/api")

app.register\_blueprint(apiBLprint)

# Swagger

## apispec

* pip install flask-restx
* pip install apispec
* pip install apispec-webframeworks
* pip install marshmallow
* pip install passlib

se prepara para retornar el json con la especificación básica del formato de la api, (osea la información sobre la versión de openapi usada )

de esta manera es necesario tener los archivos estaticos

from apispec import APISpec  
from apispec\_webframeworks.flask import FlaskPlugin  
from apispec.ext.marshmallow import MarshmallowPlugin

from flask import Flask**,** jsonify

spec = APISpec(  
 title='Flask'**,** version='0.1'**,** openapi\_version="3.0.2"**,** plugins=[FlaskPlugin()**,**MarshmallowPlugin()]  
)  
  
@app.route("/api/swagger.json")  
def create\_swagger\_spect():  
 return jsonify(spec.to\_dict())

Para documentar un endpoint, hay que crear un esquema de la entrada y la salida

import unittest  
from flask\_migrate import Migrate

from apispec import APISpec  
from apispec\_webframeworks.flask import FlaskPlugin  
from apispec.ext.marshmallow import MarshmallowPlugin

class TodoResponseSchema(Schema):  
 id=fields.Int()  
 title=fields.Str()  
 status=fields.Boolean()  
  
class TodoListResponseSchema(Schema):  
 todo\_list=fields.List(fields.Nested(TodoResponseSchema))

#endpoint documentado

@app.route("/todo")  
def todo():  
 *"""Get List of Todo  
 ---  
 get:  
 description: Get List of Todo  
 responses:  
 200:  
 description: Return List of Todo  
 content:  
 application/json:  
 schema: TodoListResponseSchema  
 """* dummy\_data={"todo\_list":[  
 {  
 "id": **1,** "title": "otro"**,** "status": "estado1"  
 }**,** {  
 "id": **2,** "title": "asd"**,** "status": "asd2"  
 }  
 ]}  
 return TodoListResponseSchema().dump(dummy\_data)

## flask-restx

* pip install flask-restx

from flask\_restx import Api**,** Namespace**,** Resource  
from flask\_restx import fields as fieldsx

crear los datos básicos de swagger

api=Api(  
 app**,** title='Mi app'**,** description="la descripcion"**,** version="1.0"**,** doc="/swagger/"**,** validate=True  
)

Por cada nameespace

todosCtrlr=Namespace('todos'**,**path="/todos/"**,** description='')  
api.add\_namespace(todosCtrlr)

describir los datos de entrada

createTodoCommand=todosCtrlr.model("CreateMarcaCommand"**,**{  
 'id':fieldsx.Integer(required=True**,**description="todo detail")**,** 'nombre':fieldsx.String(required=True**,**description="todo detail")  
})

describir los datos de lista

todoDto= todosCtrlr.model("todoDto"**,**{  
 'id':fieldsx.Integer(required=True**,**description="todo detail")**,** 'nombre':fieldsx.String(required=True**,**description="todo detail")  
})

se crea una ruta

@todosCtrlr.route("/")  
class Todo(Resource):  
  
 @todosCtrlr.marshal\_list\_with(todoDto)  
 def get(self):  
 return [ {  
 "id":marca.id**,** "nombre":marca.nombre  
 } for marca in Marca.query.all()  
 ]  
  
 @todosCtrlr.expect(createTodoCommand)  
 def post(self):  
 data=request.get\_json()  
 nombre = data['nombre']  
 marca=Marca(nombre=nombre)  
 db.session.add(marca)  
 db.session.commit()  
 return {  
 "id":marca.id**,** "nombre":marca.nombre  
 }

@todosCtrlr.route("/<int:id>")  
class TodoGet(Resource):  
 @todosCtrlr.marshal\_with(todoDto)  
 def get(self**,**id):  
 marca=Marca.query.filter\_by(id=id).first()  
 if not marca:  
 todosCtrlr.abort(**404,**"No existe este elemento")  
 return {  
 "id":marca.id**,** "nombre":marca.nombre  
 }

@todosCtrlr.marshal\_with(todoEditDto)  
def put(self**,**id):  
 marca = Marca.query.filter\_by(id=id).first()  
 if not marca:  
 todosCtrlr.abort(**404,** "No existe este elemento")  
 data = request.get\_json()  
 nombre = data['nombre']  
 marca.nombre = nombre  
 db.session.commit()  
 return {  
 "id": marca.id**,** "nombre": marca.nombre  
 }

Un mejor ejemplo

@ns.route("/students")

class StudentsListAPI(Resource):

@ns.marshal\_list\_with(student\_dto)

def get(self):

return Student.query.all()

@ns.expect(student\_input\_dto)

@ns.marshal\_with(student\_dto)

def post(self):

payload=ns.payload

name=payload["name"]

course\_id=payload["course\_id"]

student=Student(name=name,course\_id=course\_id)

db.session.add(student)

db.session.commit()

return student,201

@ns.route("/students/<int:id>")

class StudentDetailAPI(Resource):

@ns.marshal\_with(student\_dto)

def get(self, id):

student=Student.query.get(id)

return student

@ns.expect(student\_input\_dto)

@ns.marshal\_with(student\_dto)

def put(self,id):

payload=ns.payload

name=payload["name"]

student=Student.query.get(id)

student.name=name

db.session.commit()

return student

def delete(self,id):

student=Student.query.get(id)

db.session.remove(student)

db.session.commit()

return {},204

### @variableNamespace.marshal\_with(modeloDto)

Permite que en un endpoint se transforme automáticamente la respuesta de un Query en un json, pero tienen que coincidir los datos

@ns.route("/courses")  
class CoursesAPI(Resource):  
 @ns.marshal\_list\_with(course\_model)  
 def get(self):  
 return Course.query.all()

#### Envelope

@ns.marshal\_with(course\_dto, code=200, envelope="course")

El parámetro envelope en el decorador @ns.marshal\_with(course\_dto, code=200, envelope="course") de Flask-RestPlus se utiliza para envolver la respuesta serializada en un objeto adicional. Este enfoque es útil para estructurar la salida de la API de una manera más clara y coherente.

Función del Parámetro envelope

* **Estructura de Respuesta**: Al especificar un valor para envelope, el resultado de la marshalling se envuelve dentro de un diccionario con la clave que has proporcionado. En tu ejemplo, el resultado se incluirá bajo la clave "course".
* **Ejemplo de Salida**: Si el objeto original que estás serializando tiene la forma:

json

{

"id": 1,

"name": "Mathematics",

"credits": 3

}

Con envelope="course", la respuesta final será:

json

{

"course": {

"id": 1,

"name": "Mathematics",

"credits": 3

}

}

* **Beneficios**: Este método de envoltura puede ser útil para:
  + **Claridad**: Proporcionar un contexto adicional sobre el tipo de datos que se están devolviendo.
  + **Consistencia**: Mantener una estructura uniforme en las respuestas de tu API, lo que puede facilitar su uso por parte de los clientes.

### Entrada salida

Usar expect para la entrada y marshal\_with, para la salida y la serialización

@ns.expect(student\_input\_dto)  
@ns.marshal\_with(student\_dto)  
def post(self):  
 payload=ns.payload  
 name=payload["name"]  
 course\_id=payload["course\_id"]  
 student=Student(name=name**,**course\_id=course\_id)  
 db.session.add(student)  
 db.session.commit()  
 return student**,201**

### ns.payload vs request.get\_json()

El major es ns.payload

En Flask, tanto ns.payload como request.get\_json() son métodos utilizados para manejar datos JSON en las solicitudes, pero tienen diferencias clave en su uso y contexto.

ns.payload

* **Uso en Flask-RestPlus**: ns.payload es una forma de acceder a los datos de la solicitud que han sido validados y procesados por el decorador @ns.expect(). Esto significa que antes de acceder a ns.payload, se espera que el cuerpo de la solicitud haya sido validado contra un modelo definido (en tu caso, course\_input\_dto).
* **Ventaja**: Al usar ns.payload, te aseguras de que los datos cumplen con el esquema definido, lo que puede ayudar a prevenir errores y simplificar la lógica de manejo de errores en tu código.
* **Ejemplo**:

python

@ns.expect(course\_input\_dto)

**def** put(self, id):

payload = ns.payload *# Accede a los datos validados*

request.get\_json()

* **Uso General**: Este método se utiliza para extraer datos JSON del cuerpo de la solicitud sin ninguna validación previa. Es más directo y se puede usar en cualquier parte de tu aplicación Flask.
* **Flexibilidad**: Puedes usarlo para obtener datos JSON sin necesidad de definir un esquema, pero esto significa que deberás implementar manualmente cualquier validación o manejo de errores.
* **Ejemplo**:

python

**def** put(self, id):

payload = request.get\_json() *# Obtiene los datos JSON sin validación*

Comparación

| **Aspecto** | **ns.payload** | **request.get\_json()** |
| --- | --- | --- |
| **Validación** | Sí, valida contra un modelo definido | No, no realiza validaciones |
| **Uso** | Específico para Flask-RestPlus | General en cualquier aplicación Flask |
| **Manejo de Errores** | Manejo automático si se usa con @ns.expect() | Requiere implementación manual |

### Blueprint

Si usa blue print esta es la manera de adaptarlo a la api, el orden de los pasos es importante

app= Flask(\_\_name\_\_)  
apiBLprint=Blueprint("api"**,**\_\_name\_\_**,**url\_prefix="/api")  
api=Api(  
 apiBLprint**,** title='Mi app'**,** description="la descripcion"**,** version="1.0"**,** doc="/swagger/"**,** validate=True  
)  
  
app.register\_blueprint(apiBLprint)

# shell\_context\_processor

agrega variables ya establecidas al shell del flask , para evitar tener que importarlas

@app.shell\_context\_processor  
def make\_shell\_context():  
 return {  
 'db':db  
 }

# Estructuras Correctas

## 2023

2 archivos para crear un proyecto vacio y luego el comando

/app/\_\_init\_\_.py

from flask import Flask  
def create\_app():  
 app = Flask(\_\_name\_\_)  
 return app

/ .flaskenv

MONGODB\_URI=mongodb://localhost:27017

Luego corer el commando

* flask run