

Hands-on Lab API REST utilizando FLASK, Python y VSCode

Duración:

120 min

Contactos:

Wladimir Brborich

Objetivo

El objetivo de este laboratorio es poner en práctica los conocimientos obtenidos a lo largo de la capacitación en programación procedimental y Python, desarrollando un API RESTFUL sencilla utilizando el micro-framework Flask.

Prerrequisitos

Este laboratorio asume que se tiene:

- Entendimiento básico sobre el lenguaje de programación Python.
- Conocimientos básicos de bases de datos relacionales, JSON y el protocolo HTTP
- Un entorno de desarrollo configurado previamente (guía de laboratorio para configuración de entorno de desarrollo).

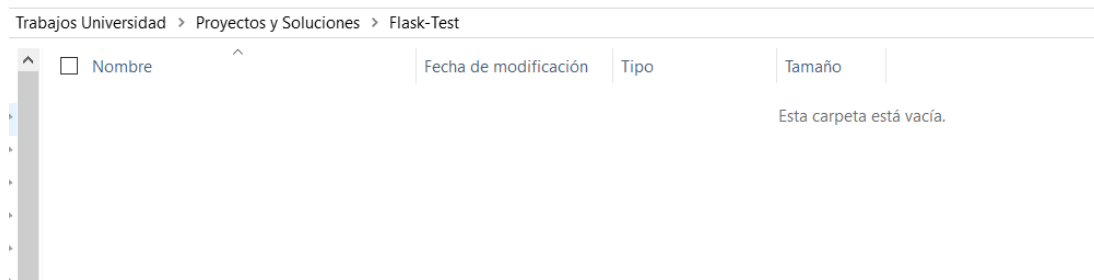
Requerimientos de software y sistema

- Windows 10 home
- Python 3.7.4
- PIP
- Visual Studio Code
- Paquete de extensiones de Python para visual studio code.
- Conexión estable a internet

Ejercicio 0: Creación del proyecto

Duración estimada: 20 minutos

1. Crear una carpeta vacía en donde alojaremos el proyecto

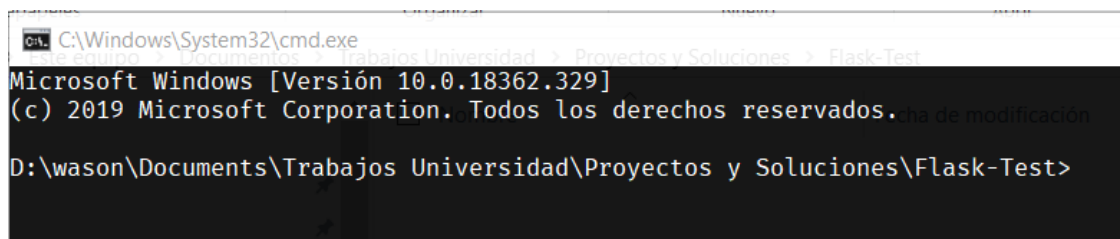


2. Copiar el template del proyecto que se encuentra en el siguiente repositorio

https://github.com/BryanWladimir/Experiment_POO_PP/blob/master/Templates_Lab/Flask-REST-Template.zip

<input type="checkbox"/> Nombre	Fecha de modificación	Tipo	Tamaño
.vscode	05/09/2019 10:16 p. m.	Carpeta de archivos	
app	05/09/2019 10:33 p. m.	Carpeta de archivos	
migrations	05/09/2019 11:05 p. m.	Carpeta de archivos	
venv	05/09/2019 10:09 p. m.	Carpeta de archivos	
.gitattributes	28/08/2019 12:23 p. m.	Documento de tex...	1 KB
.gitignore	28/08/2019 12:23 p. m.	Documento de tex...	2 KB
books.sqlite	05/09/2019 11:41 p. m.	Archivo SQLITE	24 KB
LICENSE	28/08/2019 12:23 p. m.	Archivo	2 KB
manage.py	05/09/2019 10:57 p. m.	Archivo de origen ...	1 KB
README.md	28/08/2019 12:23 p. m.	Archivo de origen ...	1 KB
requirements.txt	05/09/2019 10:06 p. m.	Documento de tex...	1 KB

3. Abrir una consola de comandos dentro de la carpeta



4. Ejecutar el comando "virtualenv venv" para crear un entorno virtual

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.18362.329]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

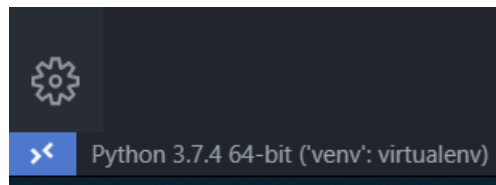
D:\wason\Documents\Trabajos Universidad\Proyectos y Soluciones\Flask-Test>virtualenv venv
Using base prefix 'd:\python\python37'
New python executable in D:\wason\Documents\Trabajos Universidad\Proyectos y Soluciones\Flask-Test\venv\Scripts\python.exe
Installing setuptools, pip, wheel...
done.

D:\wason\Documents\Trabajos Universidad\Proyectos y Soluciones\Flask-Test>
```

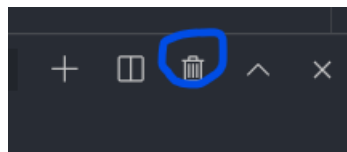
5. Ejecutar el comando “code .” en la misma consola de comandos para abrir Visual Studio Code

```
D:\wason\Documents\Trabajos Universidad\Proyectos y Soluciones\FlaskSimpleService>code .
D:\wason\Documents\Trabajos Universidad\Proyectos y Soluciones\FlaskSimpleService>
```

6. Una vez dentro del editor de texto abrir el archivo manage.py, esto activara la extensión de python y seleccionara de manera automática el intérprete del entorno virtual



7. Cerrar la terminal que se abre por defecto usando el siguiente icono



8. Abrir una nueva terminal para abrir un entorno virtual (ctrl + ` teclado en inglés, ctrl + ñ teclado en español)

```
Microsoft Windows [Versión 10.0.18362.329]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

D:\wason\Documents\Trabajos Universidad\Proyectos y Soluciones\FlaskSimpleService>"d:\wason\Documents\Trabajos Universidad\Proyectos y Soluciones\FlaskSimpleService\venv\Scripts\activate.bat"

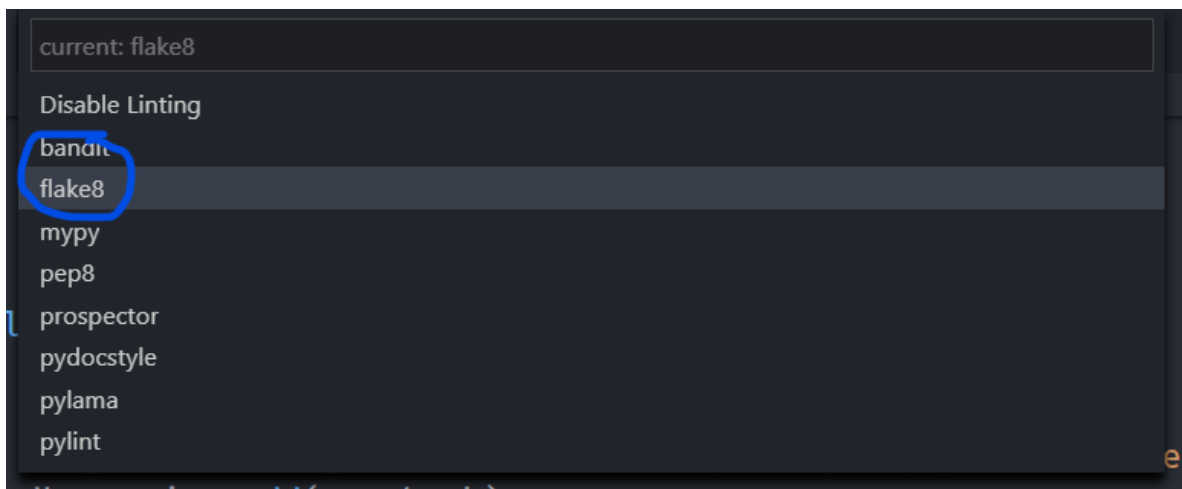
(venv) D:\wason\Documents\Trabajos Universidad\Proyectos y Soluciones\FlaskSimpleService>
```

9. Correr el siguiente comando “pip install –r requirements.txt” para instalar todas las dependencias en nuestro entorno virtual

```
pip install -r requirements.txt
```

10. Activar el linter de python, de esta manera se resaltarán algunos errores y podremos dar formato automático al código.

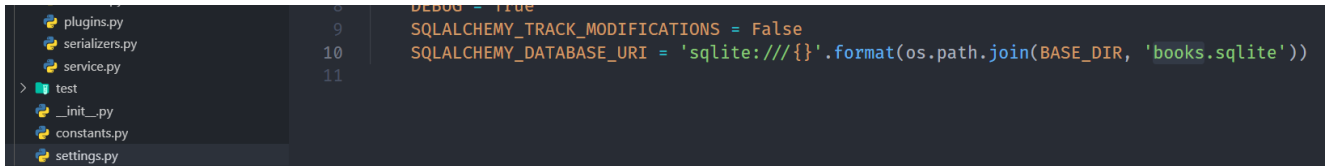
- a. Ejecutamos el comando ctrl + shift + p
- b. Escribimos linter
- c. Buscamos la opción "Python Linter"
- d. Una vez seleccionada buscamos "flake8" y lo activamos



Ejercicio 1: Creación de modelos

Duración estimada: 30 minutos

1. Actualizar el nombre de la base de datos sqlite en el archivo settings.py



```
plugins.py
serializers.py
service.py
test
__init__.py
constants.py
settings.py

DEBUG = True
SQLALCHEMY_TRACK_MODIFICATIONS = False
SQLALCHEMY_DATABASE_URI = 'sqlite:///{}'.format(os.path.join(BASE_DIR, 'books.sqlite'))
```

2. Generar Modelos en el archivo “models.py”

- a. Importar el plugin de la base de datos

```
from .plugins import db
```

- b. Generar un modelo llamado Books

```
class Book(db.Model):
    _id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    publication_date = db.Column(db.DateTime)
    author_id = db.Column(db.Integer, db.ForeignKey('author._id'))
    author = db.relationship('Author', backref=db.backref('author'))
```

- c. Generar un modelo llamado “Author”

```
class Author(db.Model):
    _id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    dni = db.Column(db.String(10), unique=True)
    birth_date = db.Column(db.DateTime)

    books = db.relationship('Book', backref=db.backref('book'))
```

Los modelos son abstracciones de una base de datos relacional, estos se encargan de hacer las consultas respectivas y convertir cada fila de la tabla en un objeto alojado en memoria.

Algo importante a tener en cuenta es que SQLAlchemy nos permite hacer referencias hacia objetos relacionados por una clave foránea, de manera que en tiempo de ejecución nuestro autor tiene la siguiente estructura:

```
{
  "_id": 1,
  "name": "Author 1",
  "dni": "0000000000",
  "birth_date": "00-00-00",
  "books": [
    {
      "_id": 1,
      "name": "Book 1",
      "publication_date": "00-00-00",
      "author_id": 1
    },
    {
      "_id": 2,
      "name": "Book 2",
      "publication_date": "00-00-00",
      "author_id": 1
    }
  ]
}
```

Y la estructura de los libros de esta manera:

```
{
  "_id": 1,
  "name": "Book 1",
  "publication_date": "00-00-00",
  "author_id": 1,
  "author": {
    "_id": 1,
    "name": "Author 1",
    "dni": "0000000000",
    "birth_date": "00-00-00"
  }
}
```

3. Importar los modelos en el archivo manage.py

```
from flask_migrate import Migrate, MigrateCommand
import pytest
from app.main.models import Book, Author

flask_app.app_context().push()
```

Al importar los modelos en este archivo Flask recoge las referencias en memoria y puede realizar efectivamente las migraciones

4. Crear la base de datos corriendo los siguientes comandos

```
python manage.py db init
```

```
python manage.py db migrate
```

```
python manage.py db upgrade
```

Ejercicio 2: Creación de serializadores

Duración estimada: 30 minutos

1. Generar Serializadores en el archivo "serializers.py"
 - a. Importar el plugin para los serializadores y los modelos que generamos en el paso anterior

```
from .plugins import ma
from .models import Book, Author
```

- b. Generar un serializador para el model Book sin las relaciones hacia Author

```
class SimpleBookSerializer(ma.ModelSchema):
    publication_date = ma.DateTime(format='%Y-%m-%d')

    class Meta:
        model = Book
        fields = ('_id', 'name', 'publication_date')
```

- c. Generar un serializador para el modelo Book incluyendo las relaciones hacia Author

```
class BookSerializer(ma.ModelSchema):

    author = ma.Nested('SimpleAuthorSerializer')
    publication_date = ma.DateTime(format='%Y-%m-%d')

    class Meta:
        model = Book
        fields = ('_id', 'name', 'publication_date', 'author', 'author_id')
```

- d. Generar un serializador para el model Author sin las relaciones hacia Book


```
class SimpleAuthorSerializer(ma.ModelSchema):

    birth_date = ma.DateTimeField(format='%Y-%m-%d')

    class Meta:
        model = Author
        fields = ('name', 'dni', 'birth_date')
```

- e. Generar un serializador para el modelo Author incluyendo las relaciones hacia Book

```
class AuthorSerializer(ma.ModelSchema):
    Nested
    books = ma.Nested('SimpleBookSerializer', many=True)
    birth_date = ma.DateTimeField(format='%Y-%m-%d')

    class Meta:
        model = Author
        fields = ('_id', 'name', 'dni', 'birth_date', 'books')
```

Los serializadores representan una estructura que convierte un objeto de tipo modelo en una estructura JSON, para ello se definen cuáles son los campos que podemos serializar y cuales queremos dejar fuera.

Por otra parte, podemos modificar el comportamiento de serialización de los campos, por ejemplo, las relaciones que se hacen con los libros, para incluir una lista de libros por cada autor.

Algo importante a tomar en cuenta es que los serializadores con la notación “Simple” nos ayudan a evitar relaciones circulares al momento de serializar libros y autores.

Ejercicio 3: Creación de rutas

Duración estimada: 40 minutos

1. Generar rutas en el archivo "service.py"

a. Importar las herramientas necesarias

```
from flask import Blueprint, request, jsonify
from app.constants import POST, GET
from .plugins import db
from .models import Author, Book
from .serializers import AuthorSerializer, BookSerializer

urls = Blueprint('urls', __name__)

# define your routes here
```

b. Generar las rutas para Author

```
@urls.route('/author', methods=POST)
def create_author():
    author_serializer = AuthorSerializer()
    new_author = author_serializer.load(request.json, partial=True)
    db.session.add(new_author)
    db.session.commit()
    return jsonify(author_serializer.serialize(new_author)), 201

@urls.route('/author', methods=GET)
def get_authors():
    author_serializer = AuthorSerializer(many=True)
    authors = Author.query.all()
    serialized_authors = author_serializer.dump(authors)
    return jsonify(serialized_authors)

@urls.route('/author/id/<_id>', methods=GET)
def get_author(_id):
    author = Author.query.get(_id)
    return jsonify(author_serializer.serialize(author))
```

c. Generar las rutas para Book

```

@urls.route('/book', methods=POST)
def create_book():
    book_serializer = BookSerializer()
    new_book = book_serializer.load(request.json, partial=True)
    db.session.add(new_book)
    db.session.commit()
    return book_serializer.jsonify(new_book), 201

@urls.route('/book', methods=GET)
def get_books():
    book_serializer = BookSerializer(many=True)
    books = Book.query.all()
    serialized_books = book_serializer.dump(books)
    return jsonify(serialized_books)

@urls.route('/book/id/<_id>', methods=GET)
def get_book(_id):
    book = Book.query.get(_id)
    return BookSerializer().jsonify(book)

```

2. Correr el proyecto con el comando “python manage.py run”

```

(venv) D:\wason\Documents\Trabajos Universidad\Proyectos y Soluciones\FlaskSimpleService>python manage.py run
* Serving Flask app "app.main" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 231-926-551
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

```

3. Usar un cliente http como postman para verificar el funcionamiento del proyecto

The screenshot shows the Postman interface with a GET request to `http://127.0.0.1:5000/api/author`. The response status is 200 OK, and the body is a JSON object representing an author and their books.

```

{
  "_id": 1,
  "birth_date": "1997-08-17",
  "books": [
    {
      "_id": 1,
      "name": "Mi primer libro",
      "publication_date": "2019-08-17"
    },
    {
      "_id": 2,
      "name": "Mi Segundo libro",
      "publication_date": "2019-09-15"
    }
  ],
  "dni": "1724561921",
  "name": "Wladimir"
}

```