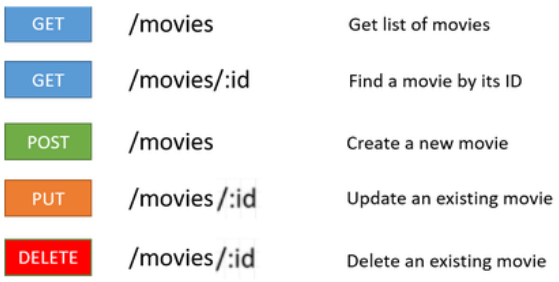
Ayudantía `APIs`

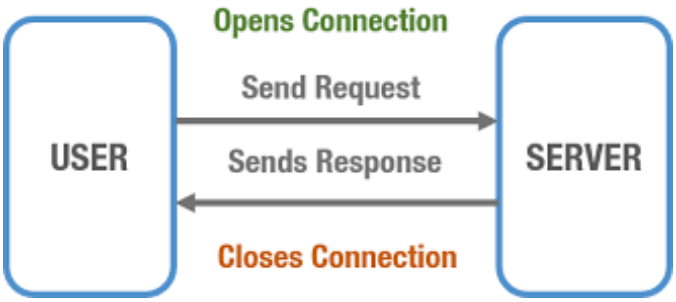
Valentina Rojas, Kevin Johnson, Nicolas Quiroz | Para la entrega 4

¿Qué es una `API`? ¿Y una `REST` `API`?

- `API`: **A**pplication **P**rogramming **I**nterface
- `REST`: **RE**presentational **S**tate **T**ransfer
  - Define varios estándares que debe tener la `API`
  - *HTTP Methods:* `POST`, `GET`, `PUT`, `PATCH`, `DELETE` ...



Ejemplo de una API



Flujo de una api

Deberán implementar una `API` para su entrega.

Virtual Environment: `pipenv`

Para poder correr su aplicación localmente, es **altamente recomendado un administrador de dependencias/paquetes,** para trabajar en un entorno aislado e instalar los paquetes solo en el *scope* del proyecto.

¿Cómo lo instalo?

```
# Windows sin python2 o ubuntu 18.04+
pip install pipenv
# MacOs y Ubuntu < 18.04 (ojo con tener varias versiones de python instaladas!)
pip3 install pipenv
```

¿Cómo lo uso?

Instalemos `pandas`

```
pipenv install pandas
```

Si por alguna razón tu consola no reconoce el comando, puedes intentar

```
python -m pipenv install pandas
```

O quizás

```
python3 -m pipenv install pandas
```

¡Y listo! (Si necesitas ayuda siempre existe `pipenv -h` (help))

Instalar desde un `Pipfile`
Un **`Pipfile`** es una declaración de librerías que el proyecto necesita para funcionar. Para esta ayudantía el `Pipfile` es:

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

# Estos son los paquetes que no se van instalar en los servidores o en "produccion"
[dev-packages]
```

```
mypy = "*"
flake8 = "*"

# Estos son los paquetes que siempre se van instalar
[packages]
matplotlib = "*"
numpy = "*"
pandas = "*"
pymongo = "*"
gunicorn = "*"
Flask = "*"
Jinja2 = "*"

[requires]
python_version = "3.6"
```

> *el `*` significa "utilizar la última versión", ya que se pueden poner versiones específicas (o `<=`)*

Para instalar sus contenidos solo realizamos

```
pipenv install
```

¡Y listo 🎉!

¿Y el `Pipfile.lock` 🤔?

Mantiene las versiones y dependencias de manera segura y congelada en el tiempo. Si existe, se prioriza la instalación de los paquetes declarados en este archivo.

Su objetivo es evitar que un usuario instale una versión más avanzada de una librería y provoque una falla en la aplicación.

`flask`: La `API` en Python 🐍

¿Qué es *`flask`* 🤯?
Clic aquí para ir a la documentación

Es una librería de Python *que sirve para hacer* *`API`*s (y mucho mas...)
Ejemplo (no del taller) tipo `"hello world"`

```
# main.py
from flask import Flask
app = Flask(__name__)

@app.route("/") # Solicitud del tipo GET
def hello():
    return "Hello World!"
if __name__ == "__main__":
    app.run()
```

Todas las solicitudes son por defecto `GET` (no solo en *`flask`*, en la vida 😀).

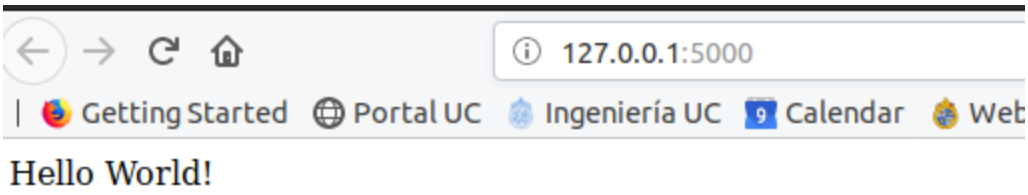
Antes de ejecutarlo:
Activar el entorno virtual

```
pipenv shell
```

Cuando lo ejecute

```
python main.py
## * Running on http://localhost:5000/
```

Si voy al navegador



Aplicación ejecutandose en el navegador

¿Pero y para la entrega?

JSON & `pymongo`

¡Los diccionarios son muy similares a archivos JSON y los documentos de Mongo!

Diccionario en Python

```
{"nombre": "valentina", "edad": "58"}
```

Documento en MongoDB

```
{nombre: "valentina", edad:"58"}
```

Por lo tanto, ¡las `db` de Mongo pueden trabajarse como diccionarios anidados! 🎉

¿Cómo? Con la librería `pymongo`

```
client = MongoClient() # El host y port donde está corriendo el servidor de MongoDB.
# Por defecto será localhost:5432
db = client["entidades"] # La base de datos a usar
usuarios = db.usuarios
usuarios.find({}, {"_id":0}) # Todos los usuarios
```

Click aquí para ir a la documentación de pymongo.

🚀 La entrega: **ejemplo**

En `main.py`

```
from flask import Flask, render_template, request, abort, json
from pymongo import MongoClient
import pandas as pd
import matplotlib.pyplot as plt
import os
import atexit
import subprocess
USER_KEYS = ['name', 'last_name', 'occupation', 'follows', 'ag
e']
# Levantamos el servidor de mongo. Esto no es necesario, puede
abrir
# una terminal y correr mongod. El DEVNULL hace que no vemos e
l output
mongod = subprocess.Popen('mongod', stdout=subprocess.DEVNULL)
# Nos aseguramos que cuando el programa termine, mongod no que
de corriendo
atexit.register(mongod.kill)
# El cliente se levanta en localhost:5432
client = MongoClient('localhost')
# Utilizamos la base de datos 'entidades'
db = client["entidades"]
# Seleccionamos la colección de usuarios
usuarios = db.usuarios
# Iniciamos la aplicación de flask
app = Flask(__name__)


@app.route("/")
```

```
def home():
    return "<h1>HELLO</h1>"
# Mapeamos esta función a la ruta '/plot' con el método get.
@app.route("/plot")
def plot():
    # Obtengo todos los usuarios
    users = usuarios.find({}, {"_id": 0})
    # Hago un data frame (tabla poderosa) con la columna 'nam
e' indexada
    df = pd.DataFrame(list(users)).set_index('name')
    # Hago un grafico de pi en base a la edad
    df.plot.pie(y='age')
    # Export la figura para usarla en el html
    pth = os.path.join('static', 'plot.png')
    plt.savefig(pth)
    # Retorna un html "rendereado"
    return render_template('plot.html')

@app.route("/users")
def get_users():
    resultados = [u for u in usuarios.find({}, {"_id": 0})]
    # Omitir el _id porque no es json serializable
    return json.jsonify(resultados)

@app.route("/users/<int:uid>")
def get_user(uid):
    users = list(usuarios.find({"uid": uid}, {"_id": 0}))
    return json.jsonify(users)

@app.route("/users", methods=['POST'])
```

```
def create_user():
    '''
    Crea un nuevo usuario en la base de datos
    Se  necesitan todos los atributos de model, a excepcion de
_id
    '''
    # Si los parámetros son enviados con una request de tipo a
pplication/json:
    data = {key: request.json[key] for key in USER_KEYS}
    # Se genera el uid
    count = usuarios.count_documents({})
    data["uid"] = count + 1
    # Insertar retorna un objeto
    result = usuarios.insert_one(data)
    # Creo el mensaje resultado
    if (result):
        message = "1 usuario creado"
        success = True
    else:
        message = "No se pudo crear el usuario"
        success = False
    # Retorno el texto plano de un json
    return json.jsonify({'success': success, 'message': messag
e})

@app.route('/users/<int:uid>', methods=['DELETE'])
def delete_user(uid):
    '''
    Elimina un usuario de la db.
    Se requiere llave uid
    '''
```

```
    # esto borra el primer resultado. si hay mas, no los borra
    usuarios.delete_one({"uid": uid})
    message = f'Usuario con id={uid} ha sido eliminado.'
    # Retorno el texto plano de un json
    return json.jsonify({'result': 'success', 'message': messa
ge})

@app.route('/users/many', methods=['DELETE'])
def delete_many_user():
    '''
    Elimina un varios usuarios de la db.
    - Se requiere llave idBulk en el body de la request applic
ation/json
    '''
    if not request.json:
        # Solicitud faltan parametros. Codigo 400: Bad request
        abort(400)  # Arrojar error
    all_uids = request.json['uidBulk']
    if not all_uids:
        # Solicitud faltan parametros. Codigo 400: Bad request
        abort(400)  # Arrojar error
    # Esto borra todos los usuarios con el id dentro de la lis
ta
    result = usuarios.delete_many({"uid": {"$in": all_uids}})
    # Creo el mensaje resultado
    message = f'{result.deleted_count} usuarios eliminados.'
    # Retorno el texto plano de un json
    return json.jsonify({'result': 'success', 'message': messa
ge})

@app.route("/test")
```

```
def test():
    # Obtener un parámero de la URL
    param = request.args.get('name', False)
    print("URL param:", param)
    # Obtener un header
    param2 = request.headers.get('name', False)
    print("Header:", param2)
    # Obtener el body
    body = request.data
    print("Body:", body)
    return "OK"

if os.name == 'nt':
    app.run()
```

¿Y para ejecutarlo?

Simple, solo debes usar

- En Windows

```
python main.py
```

- En cualquier otro sistema operativo

```
gunicorn main:app --workers=3 --reload
```

Esto lo ejecuta con `gunicorn`, con tres trabajadores (`workers`) en paralelo (las *requests* son más rápidas) y `--reload` hace que la aplicación se reinicie cada vez que cambie el contenido del archivo principal.

¿Qué es `gunicorn`?

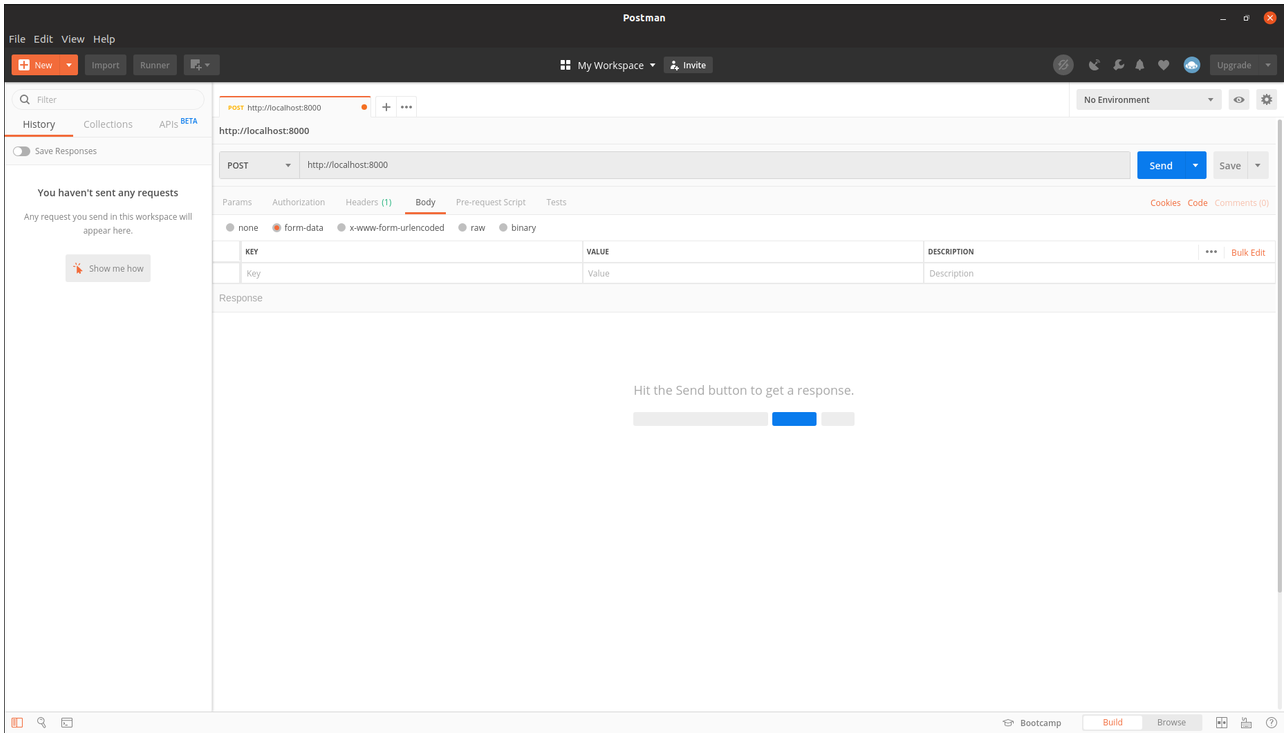
The Gunicorn "Green Unicorn" is a Python Web Server Gateway Interface HTTP implementation.

Básicamente, es una interfaz que se ubica entre el cliente y el servidor y se encarga de redireccionar las solicitudes HTTP a los servidores que correspondan.

Lamentablemente solo funciona en sistemas operativos `*NIX`, y Windows no es uno de esos.
Documentación

Postman

Postman es una herramienta muy simple, para probar *requests* HTTP.
Lo puedes descargar aquí.



Esta es la interfaz. Donde sale `localhost:8000` colocas el URL a tu API.

Donde sale `POST` puedes cambiar el tipo de *request* que estás haciendo.
Y en `Body`, puedes agregar los parámetros necesarios para la solicitud. Se recomienda enviarlos como `"raw"` y escogiendo el formato `"application/json"`.

Para probar una *request*, solo debes apretar el botón `Send`.