

Resumen Laboratorio IV - BackEnd (2022)

Python:

Es un lenguaje dinámico, sin tipos al igual que JavaScript, se puede crear una variable y luego asignarle un str, int, etc.

Verificar si tenemos instalado Python: en CMD o PowerShell escribimos "Python" y nos dice si lo tenemos instalado y que versión.

Podemos utilizar la librería de **Anaconda**: <https://www.anaconda.com/> (Sirve para Data Science) o la versión regular de **Python**: <https://www.python.org/downloads/>

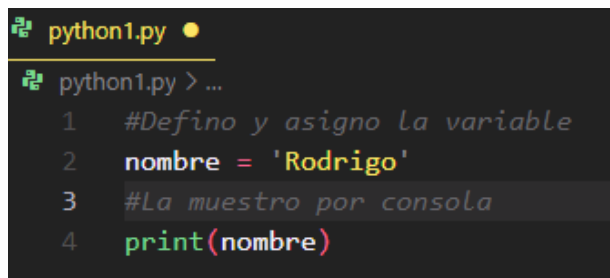
La que vamos a usar en LabIV para backend es **FastAPI**: <https://fastapi.tiangolo.com/> (Sirve para Web API)

Definición y asignación de variables (En consola): (Por convención se declaran en minúscula y espacios con '_')

```
(base) PS D:\Documentos\UTN - Laboratorio IV
2022\5_Backend\Clase_121022\Clase121022> python
Type "help", "copyright", "credits" or "license" for more information.
>>> nombre
'Rodrigo'
>>> print(nombre)
Rodrigo
>>> type(nombre)
<class 'str'>
>>> nombre = 123
>>> print(nombre)
123
>>> type(nombre)
<class 'int'>
>>>
```

(Salimos de la línea comando con "exit()" o con CTRL+Z)

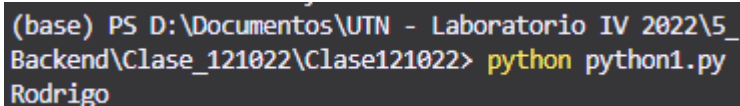
En VSC:



```
python1.py
python1.py > ...
1  #Defino y asigno la variable
2  nombre = 'Rodrigo'
3  #La muestro por consola
4  print(nombre)
```

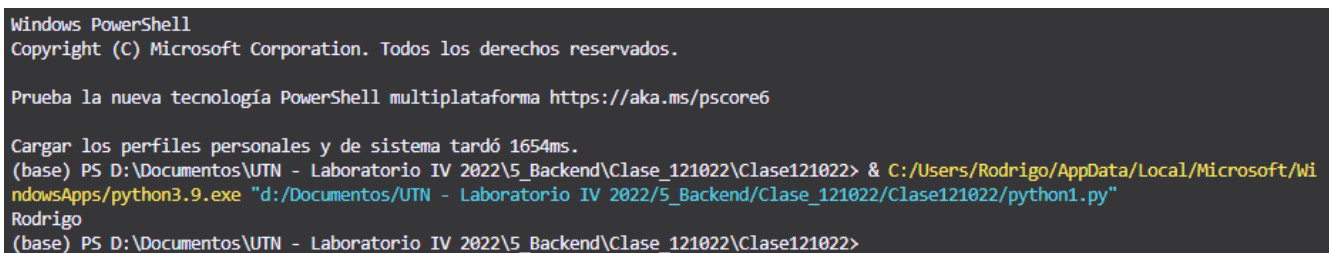
Para ejecutarlo, desde una consola y verificando que estemos en la carpeta del archivo *.py ejecutamos el comando:

python nombrearhivo.py (También podemos hacer clic en el ícono ejecutar de VSC)



```
(base) PS D:\Documentos\UTN - Laboratorio IV 2022\5_Backend\Clase_121022\Clase121022> python python1.py
Rodrigo
```

Con el ícono de ejecutar:



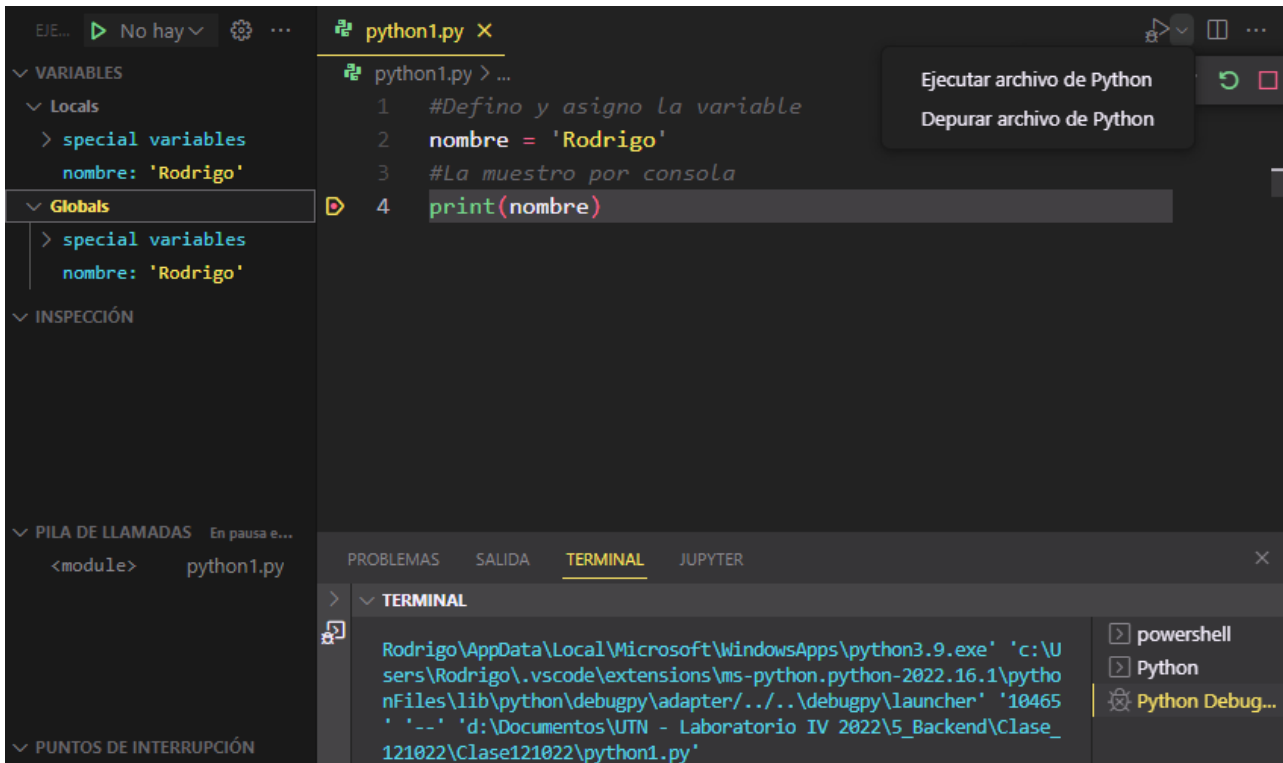
```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

Cargar los perfiles personales y de sistema tardó 1654ms.
(base) PS D:\Documentos\UTN - Laboratorio IV 2022\5_Backend\Clase_121022\Clase121022> & C:/Users/Rodrigo/AppData/Local/Microsoft/WindowsApps/python3.9.exe "d:/Documentos/UTN - Laboratorio IV 2022/5_Backend/Clase_121022/Clase121022/python1.py"
Rodrigo
(base) PS D:\Documentos\UTN - Laboratorio IV 2022\5_Backend\Clase_121022\Clase121022>
```

Resumen Laboratorio IV - BackEnd (2022)

Podemos colocar un breakpoint y luego ejecutar en modo depuración para hacer el debug. “Depurar archivo de python”



Función `input`:

```
#Tener en cuenta que todo lo que toma input
#Lo devuelve como un str
op1 = input('Ingrese el primer operando: ')
op2 = input('Ingrese el segundo operando: ')
print(op1+op2)
```

```
Ingrese el primer operando: 10
Ingrese el segundo operando: 5
105
```

```
(function) input: (__prompt: object = ..., /) -> str
```

Read a string from standard input. The trailing newline is stripped.

The prompt string, if given, is printed to standard output without a trailing newline before reading input.

En este caso, en vez de sumar lo que hace es concatenar los valores 10 y 5 porque los toma como str y el + sirve para concatenar. Entonces hay que castear el valor que devuelve, en este caso a un entero, de esta forma:

```
#Casteo von int()
op1 = int(input('Ingrese el primer operando: '))
op2 = int(input('Ingrese el segundo operando: '))
print(op1+op2)
```

```
Ingrese el primer operando: 10
Ingrese el segundo operando: 5
15
```

Ahora, si quisiera mostrar un mensaje con un str, y mostrar el resultado obtenido no puedo hacer esto porque me da error de concatenación. (Solo se pueden concatenar str):

Resumen Laboratorio IV - BackEnd (2022)

```
#Mostrar el resultado con un mensaje:  
print('El resultado es: '+op1+op2)
```

Se produjo una excepción: **TypeError** ×

can only concatenate str (not "int") to str

```
File "D:\Documentos\UTN - Laboratorio IV 2022\5_Backend\Clase_121022\Clase121022\python1.py", line 12, in  
<module>  
    print('El resultado es: '+op1+op2)
```

Entonces se puede hacer lo siguiente, convertir el resultado de la operación en un str:

```
#Mostrar el resultado con un mensaje:  
print('El resultado es: '+ str(op1+op2))
```

```
Ingrese el primer operando: 10  
Ingrese el segundo operando: 5  
El resultado es: 15
```

Interpolación de strings (string f): Otra forma, es colocando la **f** y las comillas simples `'` y la variable entre llaves `{variable}`.

```
#Mostrar el resultado con un mensaje opción 2:  
print(f'El resultado es: {op1+op2}!!')
```

```
Ingrese el primer operando: 10  
Ingrese el segundo operando: 2  
El resultado es: 12!
```

Condicionales: (La indentación con tabulación o espacios es fundamental)

```
numero_azar.py > ...  
9   #Lo muestro por consola  
10  print(numero_secreto)  
11  #Pido que ingresen un numero para comparar:  
12  prueba = int(input('Ingrese un num entre 1 y 100: '))  
13  #Comprobamos (Controlar Los TAB!)  
14  if numero_secreto>prueba:  
15      print('Mi número es mayor')  
16  elif numero_secreto<prueba:  
17      print('Mi número es menor')  
18  else:  
19      print('Ese era mi numero! Felicitaciones!!!')
```

Bloque while:

```
python1.py  numero_azar.py ●  
numero_azar.py > ...  
1  from random import random  
2  #Importa la libreria random completa  
3  import random  
4  #Obtengo un int al azar  
5  numero_secreto = random.randint(1,100)  
6  #Lo muestro por consola  
7  print(numero_secreto)  
8  #Declaro prueba fuera del rango del random  
9  prueba = 0  
10 #Hago un while:  
11 while prueba != numero_secreto:  
12     #Pido que ingresen un numero para comparar:  
13     prueba = int(input('Ingrese un num entre 1 y 100: '))  
14     #Comprobamos condiciones  
15     if numero_secreto>prueba:  
16         print('Mi número es mayor')  
17     elif numero_secreto<prueba:  
18         print('Mi número es menor')  
19     else:  
20         print('Ese era mi numero! Felicitaciones!!!')
```

```
Ingrese un num entre 1 y 100: 50  
Mi número es mayor  
Ingrese un num entre 1 y 100: 56  
Mi número es menor  
Ingrese un num entre 1 y 100: 55  
Ese era mi numero! Felicitaciones!!!  
(base) PS D:\Documentos\UTN - Laboratorio
```

Resumen Laboratorio IV - BackEnd (2022)

Extraer una función:

Podemos realizarla en el mismo código de la siguiente manera.

Def función(parámetros):

Si devuelve algo se agrega un return, en el caso de que no, se termina el bloque de código y no se agrega nada. Es conveniente que las funciones sólo realicen una tarea. Hay muchas formas de hacerlo.

```
numero_azar.py > ...
1  import random
2  #Importa la libreria random completa
3
4  #Declaro la función "comparar" con 2 parametros
5  def comparar(secreto, p):
6      #Comprobamos condiciones
7      if secreto>p:
8          print('Mi número es mayor')
9      elif secreto<p:
10         print('Mi número es menor')
11     else:
12         print('Ese era mi numero! Felicitaciones!!!')
13     #No retorna nada64
14
15     #Obtengo un int al azar
16     numero_secreto = random.randint(1,100)
17     #Declaro prueba fuera del rango del random
18     prueba = 0
19     #Lo muestro por consola
20     print(numero_secreto)
21
22     #Hago un while:
23     while prueba != numero_secreto:
24         #Pido que ingresen un numero para comparar:
25         prueba = int(input('Ingrese un num entre 1 y 100: '))
26         #llamo a la función "comparar"
27         comparar(numero_secreto, prueba)
```

Sacar la función en otro archivo (Librería propia): Una forma es trasladando la función a otro archivo "útiles" por ejemplo, y luego importarlo desde el archivo donde queremos usar la función con "import útiles" Esto trae todo lo que hay en el archivo "útiles". Luego para usar la función comparar hacemos útiles.comparar(...,...) para utilizarla. Para ordenar mejor el código, se sugiere importar la función de esta forma:

Resumen Laboratorio IV - BackEnd (2022)

```
numero_azar.py
1 #Importa la libreria random completa
2 import random
3 #importo la función comparar desde útiles
4 from utiles import comparar
5
6 #Obtengo un int al azar
7 numero_secreto = random.randint(1,100)
8 #Declaro prueba fuera del rango del random
9 prueba = 0
10 #Lo muestro por consola
11 print(numero_secreto)
12
13 #Hago un while:
14 while prueba != numero_secreto:
15     #Pido que ingresen un numero para comparar:
16     prueba = int(input('Ingrese un num entre 1 y 100: '))
17     #Llamo a la función "comparar"
18     comparar(numero_secreto, prueba)
19
utiles.py
1 #Declaro la función "comparar" con 2 parametros
2 def comparar(secreto, p):
3     #Comprobamos condiciones
4     if secreto>p:
5         print('Mi número es mayor')
6     elif secreto<p:
7         print('Mi número es menor')
8     else:
9         print('Ese era mi numero! Felicitaciones!!!')
10     #No retorna nada64
```

Negación (not): En python se utiliza la palabra **not** para negar un valor, es decir obtener su valor opuesto en la tabla de verdad. (Tener en cuenta que en python, los valores verdadero y falso son objetos y se declaran con mayúscula **True** o **False**)

Ej:

```
if not letra_encontrada:
    acciones.....
```

Diccionarios: Se declara con un nombre de variable, y **entre llaves {}** se agrega por cada elemento 1ro el nombre del valor de la clave y 2do el valor que corresponde a esa clave, es decir **clave:valor**. Cada elemento se separa por una coma y se puede acceder a la posición de cada elemento como en los arrays.

Ejemplo:

```
serpDict = { "Cobra": "Elápidos", "Pelota": "Pitón", "Trimeresurus": "Víbora" }
```

+info:

<https://www.freecodecamp.org/espanol/news/compresion-de-diccionario-en-python-explicado-con-ejemplos/#:~:text=%C2%BFQu%C3%A9%20es%20un%20diccionario%20en,un%20par%20de%20corchetes%20%7B%7D%20.>

Ejemplo de programa en python:

Juego del ahorcado:

```
# juego del ahorcado
# definir una constante con una palabra - palabra secreta
# el usuario ingresa una palabra
# se mira letra por letra a ver cuales están en la palabra secreta y se
muestran.
# Las letras que no coinciden se muestran con asteriscos

secreto = 'limon'

#Diccionario
estados_ahorcado = {
    0: '  |  ---
|'
```

```

    ' ',
1: ' ' |---
    0

    ' ',
2: ' ' |---
    _0_

    ' ',
3: ' ' |---
    _0_
    |

    ' ',
4: ' ' |---
    _0_
    | |

    ' ',
5: ' ' |---
    _0_
    | | |

    ' ',
6: ' ' |---
    _0_
    | | |
    /

    ' ',
7: r' ' |---
    _0_
    | | |
    / \
    ' ' }

```

```

lista_usuario = []
for i in range(len(secreto)):
    lista_usuario.append('*')

letra = ''
palabra_usuario = ''
errores = 0
while secreto != palabra_usuario:
    letra = input('ingrese una letra: ')
    letra_encontrada = False
    for indice in range(len(secreto)):

```

Resumen Laboratorio IV - BackEnd (2022)

```
if secreto[indice] == letra:
    lista_usuario[indice] = letra
    letra_encontrada = True
if not letra_encontrada: #negación de un valor "not"
    errores += 1

print(estados_ahorcado[errores])

palabra_usuario = ''.join(lista_usuario)
print(palabra_usuario)
if errores == 7:
    print('Perdiste')
    break
```

FastAPI: Es un framework para hacer **servicios web**, fue pensado con el objetivo de hacer fácil la programación de sitios web, proyectos del tipo webAPI (rutas, endpoints, etc.)

No es un servidor, así que necesita un servidor que escuche los http y responda. Se instala aparte como un servidor web que se pueda programar en python (Asíncrono) pueden ser **uvicorn** y **gunicorn**. Usaremos **uvicorn** como servidor.

(Podemos ver el tutorial: <https://fastapi.tiangolo.com/tutorial/>)

<https://fastapi.tiangolo.com/>

Instalación:

```
pip install "fastapi[all]" (De esta forma instala también el uvicorn)
```

```
pip install "uvicorn[standard]"
```

Creando una aplicación con FastAPI (De forma manual):

Primero creamos un directorio donde agregar los archivos, ej: api1.py y luego avanzamos en ese archivo.

```
#1- Importamos el módulo de FastAPI:
from fastapi import FastAPI

#2- Creamos una variable de tipo FastAPI, es una instancia de la clase FastAPI
#en python no se pone new, directamente se pone el nombre de la clase
app = FastAPI()

#3- Creamos una función que nos devuelve un string
#4- Ahora la convertimos en un endpoint asignándole un atributo get y su ruta
@app.get("/") #Ruta de inicio
def read_root():
    return "Hola"
```

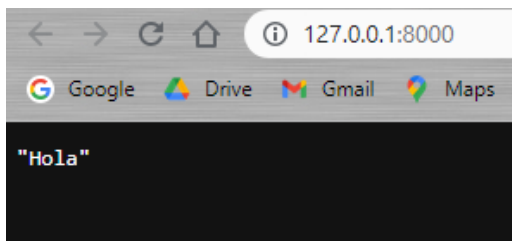
Para ejecutar esto, necesitamos un servidor para eso tenemos que ejecutar **uvicorn**, de la siguiente forma. Desde CMD **ubicados en la carpeta donde está el archivo** escribimos:

```
uvicorn nombearchivo:nombredevariable
```

Resumen Laboratorio IV - BackEnd (2022)

```
(base) PS D:\Documentos\UTN - Laboratorio IV 2022\5_Backend\Clase_191022\fastapi_191022\ejemplo1\ejemplo1> uvicorn api:app
INFO: Started server process [9140]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

La aplicación se está ejecutando en: <http://127.0.0.1:8000/> (La instancia de FastAPI está en la variable "app")



Bajamos el servidor con CTRL+C

Creando una aplicación con FastAPI (Ejecutando el uvicorn directamente desde Python):

Importamos uvicorn, llamamos a la variable app utilizando el método `run`. Luego ejecutamos desde VSC con play.

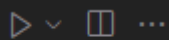
```
#5- Importamos uvicorn
import uvicorn

#1-Importamos el módulo de FastAPI:
from fastapi import FastAPI

#2- Creamos una variable de tipo FastAPI, es una instancia de la clase FastAPI
#en python no se pone new, directamente se pone el nombre de la clase
app = FastAPI()

#3- Creamos una función que nos devuelve un string
#4- Ahora la convertimos en un endpoint asignándole un atributo get y su ruta
@app.get("/") #Ruta de inicio
def read_root():
    return "Hola"

#6- Llamamos a uvicorn
uvicorn.run(app)
```



Lo ejecutamos con el botón de play:

```
WindowsApps/python3.9.exe "d:/Documentos/UTN - Laboratorio IV 2022/5_Backend
INFO: Started server process [19400]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: 127.0.0.1:14099 - "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:14099 - "GET / HTTP/1.1" 200 OK
```

Si no funciona, ver que esté usando el entorno de python correcto, en mi caso **3.10.9 (Conda)**:

```
★ Python 3.11.3 64-bit /usr/local/bin/python3
Python 3.10.9 ('base') ~/Applications/anaconda3/bin/python
Python 3.11.3 64-bit /usr/local/bin/python3
```


Resumen Laboratorio IV - BackEnd (2022)

Como está escrito el código ahora, si yo importo este archivo desde otro haciendo “import **api1**”, Python traería todo el contenido del archivo, es decir lee, define y ejecuta todas las variables que contiene api1.py. Es muy probable que, si cuando yo importo el archivo, no quiera que se ejecute el servidor, es decir que se definan todas las variables y queden listas para usar, **pero luego ejecutar el servidor desde donde lo importo para tener más control.**

Hay una forma en Python de indicarle que cierto código se ejecute solamente cuando se ejecuta a través de Python, y no cuando se importa. Para eso, se define una **función name** de la forma **__name__** (Variables mágicas). Esta variable mantiene el nombre del módulo que se está ejecutando, cuando este módulo se ejecuta porque lo **importamos** desde otro lado el nombre **__name__** va a ser el nombre del archivo (**api1**), ahora cuando lo ejecutamos directamente ese nombre va a ser: “**__main__**”. Con esa instrucción le estamos diciendo que si este archivo se ejecuta desde la línea de comandos (python api1.py) entonces, **ejecute** el servidor, ahora si yo lo importo desde otro lado, **no ejecute** ese código. Eso se logra con la verificación del if, Python se va a dar cuenta que el name del archivo desde donde importó, no es el mismo que el del archivo donde está la función de ejecutar el servidor, entonces lo pasa por alto y no lo ejecuta. El código queda:

#7- Permite que no se ejecute el servidor cuando importamos api1.py desde otro lado

```
print(__name__) #Muestra el nombre del archivo donde se está ejecutando, si lo hacemos desde python api1.py mostrará __main__. En cambio si lo hacemos desde un archivo donde lo importamos __name__ será diferente de “__main__” por lo que mostrará el archivo donde se encuentra y el servidor uvicorn no se ejecutará ya que está condicionado en el “if”
```

```
if __name__ == "__main__":  
    #6- Llamamos a uvicorn  
    uvicorn.run("api1:app", reload=True) #("nombrearchivo:nombrevariable", reload=true) De esta forma permite el reload automático
```

Importo api1.py en otro archivo “api2.py”:

```
import api1  
  
print("estoy en api2")
```

Ejecutando python api1.py:

```
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)  
INFO:      Started reloader process [4928] using WatchFiles  
__mp_main__  
api1  
INFO:      Started server process [4930]  
INFO:      Waiting for application startup.  
INFO:      Application startup complete.
```

Ejecutando desde python api2.py (Importando):

```
● (base) negrux@Rodrigos-MBP backend_fast_api % python api2.py  
api1  
estoy en api2  
○ (base) negrux@Rodrigos-MBP backend_fast_api %
```

Resumen Laboratorio IV - BackEnd (2022)

Ahí vemos que muestra el nombre del archivo importado "api1" y luego continúa a la ejecución del código de api2 sin levantar el servidor. *Es decir, el archivo api1.py ejecuta el servidor uvicorn sólo cuando se ejecuta desde el mismo archivo y no cuando es importado como un módulo en otro archivo.*

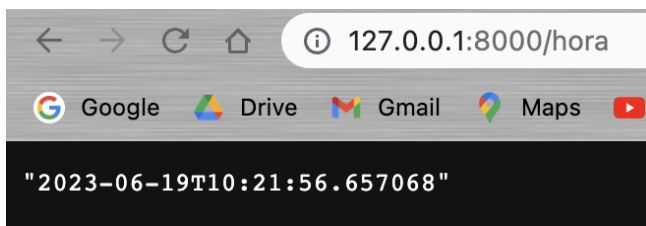
*Si queremos que cada vez que hagamos una modificación no tengamos que bajar y volver al servidor, lo que hacemos es agregar estos parámetros al servidor uvicorn:

"nombrearchivo:nombrevariable", reload=True

```
if __name__ == "__main__":  
    uvicorn.run("api1:app", reload=True) # Reload automático
```

Para crear un endpoint. Ejemplo que devuelve la hora cuando colocamos la ruta "/hora"

```
from datetime import datetime  
  
#Ejemplo de endpoint para obtener la hora  
@app.get('/hora') #Ruta del endpoint  
def get_hora():  
    return datetime.now()
```



Declarar una clase:

- 1- Creamos un nuevo archivo que se llame **version.py**
- 2- Dentro colocamos class Version: **(nombre de clase con mayúscula, nombre de archivo con minúscula)**
- 3- Luego definimos 3 **atributos** (mayor, menor, patch), *como Python es de tipo dinámico no se definen los tipos de variable.*
- 4- Para crear los atributos definimos con un inicializador de instancia de la siguiente forma:

```
def __init__(self, mayor, menor, patch)
```

- def __init__ es un inicializador
 - self: Es el primer parámetro que representa la instancia de esa clase. Si no le colocamos self queda como método global lo que generaría errores probablemente, por lo tanto, hay que ponerlo siempre en los métodos de clase.
- 5- Asigno a cada atributo de la instancia un valor que viene del inicializador, por ejemplo self.mayor = mayor
 - 6- Luego creo un método que devuelve un str con el número de versión (Equivalente al toString() de C#). **Se tiene que colocar self como parámetro para que Python sepa que es un método de la misma clase y nombrarse como __str__.** Este método pertenece a la clase Object de la cual descenden todas las otras clases. Si no redefinimos este método nos muestra la dirección de memoria y el tipo del objeto, lo cual es ilegible para las personas en general. Cada vez que tenga que convertir a str el objeto lo hará mostrando la info definida en el __str__.

```
class Version:
    def __init__(self, mayor, menor, patch):
        self.mayor = mayor
        self.menor = menor
        self.nro_de_patch = patch

#Definimos un método de la clase, para eso tiene que agregarse el self como
parámetro. (Equivalente al toString() de c#)
    def __str__(self):
        return f'V{self.mayor}.{self.menor}.{self.nro_de_patch}'
```

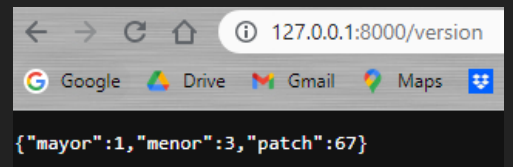
- 7- Creamos un **método propio de la clase** que devuelve un diccionario con información. Los métodos de clase deben incluir el "self".

```
def as_dict(self):
    return {
        'mayor': self.mayor,
        'menor': self.menor,
        'patch': self.nro_de_patch,
    }
```

- 8- Luego importamos esa clase desde api1.py y creamos una instancia de la versión.

```
from version import Version

#9- Ejemplo de endpoint que obtiene un diccionario que luego FastAPI convierte
en JSON
@app.get('/version') #Ruta del endpoint
def get_version():
    v = Version(1,3,67) #Instancio la clase
    return v.as_dict() #Llamo al método as_dict()
```



Ejemplo2: Clase auto

```
# declarar una clase Auto con
# marca: string
# modelo: string
# anio: integer
# fecha_compra: date
# tipo: string

# agregar un endpoint que cree una instancia de la clase y la devuelva como
json

from datetime import datetime
```

Resumen Laboratorio IV - BackEnd (2022)

```
class Auto: #Fecha de compra y tipo están seteados como parámetros opcionales
con valores x defecto

    def __init__(self, marca, modelo, anio, fecha_compra= datetime.now(), tipo
= 'automovil'):
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.fecha_compra = fecha_compra
        self.tipo = tipo

    def as_dict(self):
        return {
            'marca': self.marca,
            'modelo': self.modelo,
            'anio': self.anio,
            'fecha_compra': self.fecha_compra,
            'tipo': self.tipo
        }
```

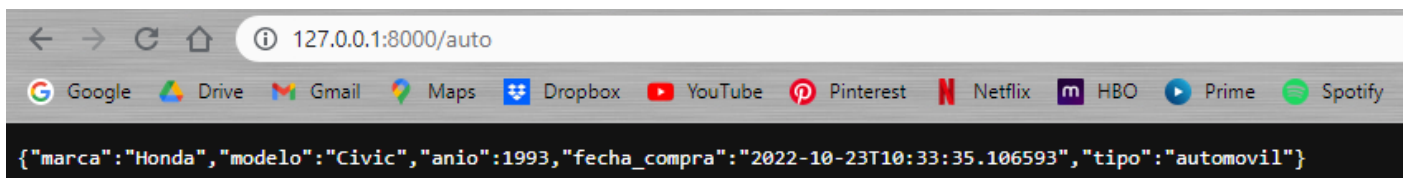
Clase api2:

```
from imp import reload
from auto import Auto
import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get('/auto') #Ruta de inicio
def get_auto(): #Acá no espera el "self" porque es función global, no de clase
    a = Auto('Honda', 'Civic', 1993)
    return a.as_dict()

if __name__ == "__main__":
    #De esta forma obtiene el nombre de la aplicación
    uvicorn.run(f"{__name__}:app", reload=True)
```



Ahora vamos a definir una clase **Moto**, pero como un modelo de datos de FastAPI:

Definimos la clase diciéndole que hereda de otra clase que se llama **BaseModel**, hay que importar el BaseModel desde una librería que se llama **pydantic**, esto me permite utilizar tipos en las variables, y no solo utilizarlos si no que fuerza a las variables a que pertenezcan esos tipos.

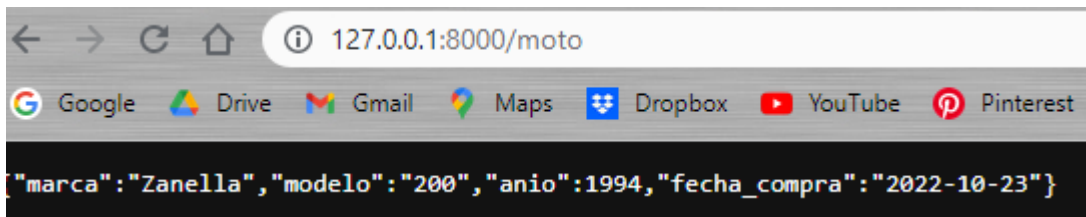
La clase se declara de esta forma:

Resumen Laboratorio IV - BackEnd (2022)

```
from pydantic import BaseModel #Permite utilizar/forzar tipos en las variables

#Moto hereda de la clase BaseModel de la librería pydantic (Importarla)
class Moto(BaseModel):
    marca: str #De esta forma obligamos a las variables a tener tipo
    modelo: str
    anio: int
    fecha_compra: date

@app.get('/moto') #Ruta de inicio
def get_moto():
    #La forma de instanciar es similar a c# asignando los valores
    #Se puede dejar sin parámetros también
    m = Moto(
        marca = 'Zanella',
        modelo = '200',
        anio = 1994,
        fecha_compra = date.today()
    )
    return m
```



BaseModel nos hace la conversión automática a JSON a través de FastAPI. Por ejemplo la forma en que aparecen los valores nulos es “null” y no “none” como se muestran en Python.

Si quisiera declarar atributos como opcionales, agrego la opción “**Optional**”, para ello hay que importar la librería **typing**. Por ejemplo hacemos la variable fecha_compra opcional. (Otra forma variable: `int = None`)

```
from typing import Optional

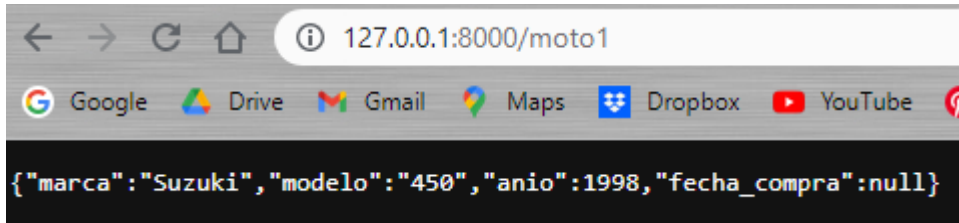
class Moto(BaseModel):
    marca: str #De esta forma obligamos a las variables a tener tipo
    modelo: str
    anio: int
    fecha_compra: Optional[date] #fecha_compra: date = None (desde Python 3.2)
```

Crear un array de Motos:

```
#Creo un array de motos:
motos = [
    Moto( marca = 'Honda', modelo = 'CBR', anio = 2000),
    Moto( marca = 'Suzuki', modelo = '450', anio = 1998),
    Moto( marca = 'Ducati', modelo = 'Corso', anio = 2010)
]
```

Resumen Laboratorio IV - BackEnd (2022)

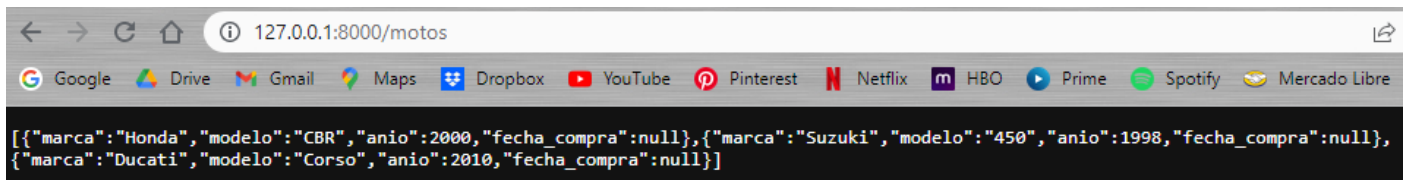
```
#Creo un endpoint que me devuelve la segunda moto del array:
@app.get('/moto1') #Ruta de inicio
def get_moto():
    m = motos[1]
    return m
```



A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:8000/moto1`. Below the address bar, there is a row of search engine and service icons including Google, Drive, Gmail, Maps, Dropbox, YouTube, and Pinterest. The main content area of the browser displays a JSON object: `{"marca": "Suzuki", "modelo": "450", "anio": 1998, "fecha_compra": null}`.

Creo un endpoint que me devuelva el **array completo**, como un JSON:

```
@app.get('/motos') #Ruta de inicio
def get_lista_motos():
    return motos
```

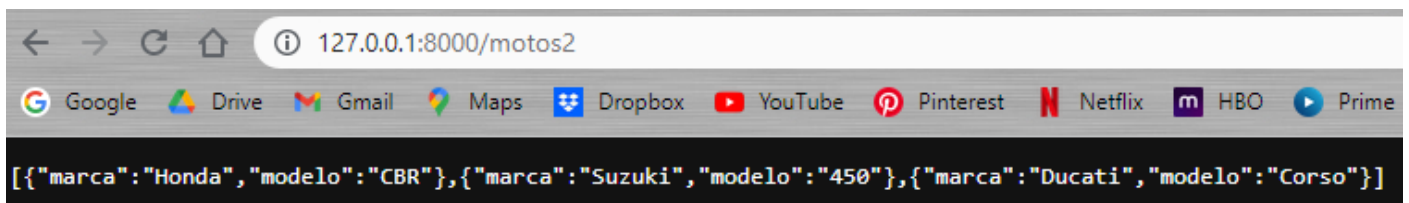


A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:8000/motos`. Below the address bar, there is a row of search engine and service icons including Google, Drive, Gmail, Maps, Dropbox, YouTube, Pinterest, Netflix, HBO, Prime, Spotify, and Mercado Libre. The main content area of the browser displays a JSON array of three motorcycle objects: `[{"marca": "Honda", "modelo": "CBR", "anio": 2000, "fecha_compra": null}, {"marca": "Suzuki", "modelo": "450", "anio": 1998, "fecha_compra": null}, {"marca": "Ducati", "modelo": "Corso", "anio": 2010, "fecha_compra": null}]`.

Ahora, si quisiéramos que, en vez de los datos completos de cada objeto, nos devuelva solo algunos atributos particulares como por ejemplo **marca y modelo**, lo que hacemos es crear otro modelo de clase que herede de **BaseModel**, **obviamente los atributos tienen que tener el mismo nombre** para que FastAPI pueda interpretar los atributos a devolver. Luego creamos otro endpoint donde le decimos: Devolveme el array completo, pero ahora adaptalo al modelo nuevo. Quedaría de la siguiente forma.

```
#Creamos un nuevo modelo para esa respuesta
class MotoLista(BaseModel):
    marca: str
    modelo: str
```

```
#Creo un endpoint que me devuelve una lista (list) de motos en
# base al nuevo modelo de respuesta:
@app.get('/motos2', response_model=list[MotoLista]) #Ruta de inicio y modelo de
response
def get_lista_motos2():
    return motos
```



A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:8000/motos2`. Below the address bar, there is a row of search engine and service icons including Google, Drive, Gmail, Maps, Dropbox, YouTube, Pinterest, Netflix, HBO, and Prime. The main content area of the browser displays a JSON array of three motorcycle objects, but only the 'marca' and 'modelo' attributes are present: `[{"marca": "Honda", "modelo": "CBR"}, {"marca": "Suzuki", "modelo": "450"}, {"marca": "Ducati", "modelo": "Corso"}]`.

Gracias a **FastAPI** nos ahorramos crear un diccionario con solo los atributos de marca y modelo, y luego tener que iterar sobre ese diccionario. Esto es más eficiente y optimizado.

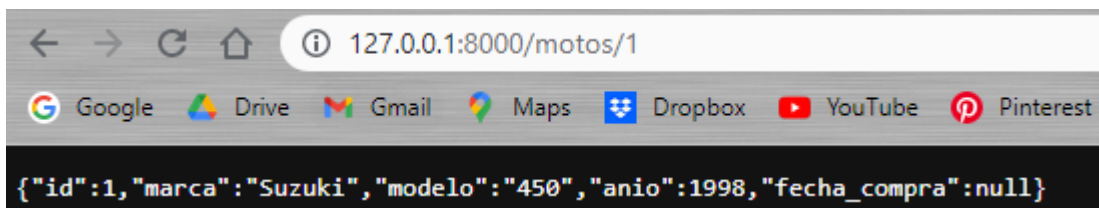
Resumen Laboratorio IV - BackEnd (2022)

Si quisiéramos pasar un parámetro, por ejemplo el **id** lo hacemos de la siguiente forma:

```
class Moto(BaseModel):  
    id: int #Agregamos una nueva variable para buscar por parámetro.  
    marca: str #De esta forma obligamos a las variables a tener tipo  
    modelo: str  
    anio: int  
    fecha_compra: Optional[date]
```

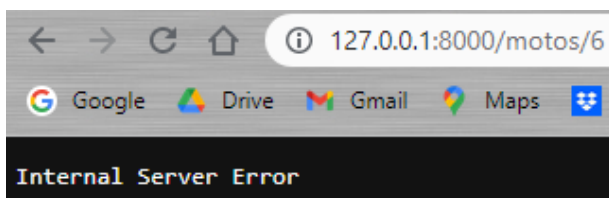
```
motos = [  
    Moto( id=0, marca = 'Honda', modelo = 'CBR',anio = 2000),  
    Moto( id=1, marca = 'Suzuki', modelo = '450',anio = 1998),  
    Moto( id=2, marca = 'Ducati', modelo = 'Corso',anio = 2010)  
]
```

```
#Creo un endpoint que me devuelve la moto según el id:  
@app.get('/motos/{id}') #Ruta de inicio - Parámetro entre {}  
def get_moto_id(id: int): #el parámetro que pasamos en la ruta se tiene  
    q' llamar igual {id}=id  
    m = motos[id]  
    return m
```

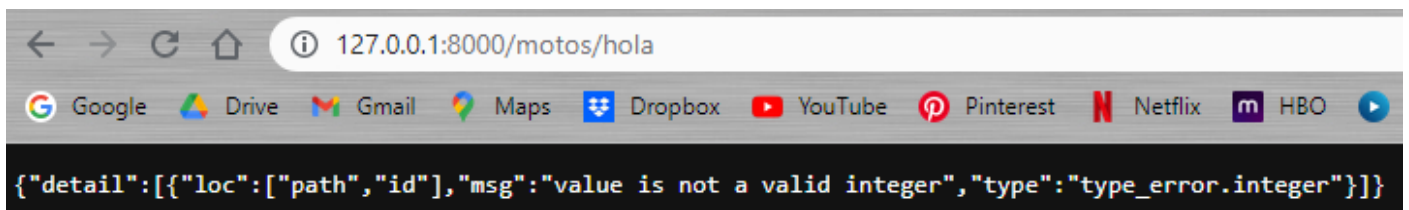


```
{\"id\":1,\"marca\":\"Suzuki\",\"modelo\":\"450\",\"anio\":1998,\"fecha_compra\":null}
```

En el caso de colocar un índice fuera de rango del array, o que no corresponde el valor, nos devolverá un error:



Internal Server Error



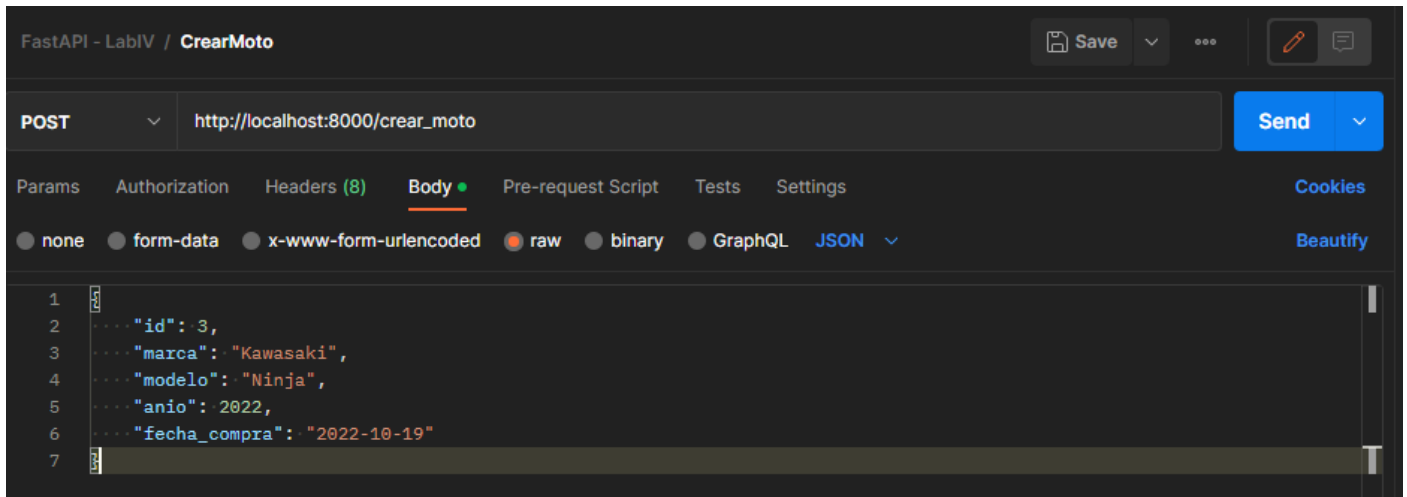
```
{\"detail\": [{\"loc\": [\"path\", \"id\"], \"msg\": \"value is not a valid integer\", \"type\": \"type_error.integer\"}]}
```

Si queremos **crear un nuevo objeto** a través de un endpoint de POST: (Usamos la herramienta PostMan para enviar los JSON a FastAPI)

```
#FastAPI va a tomar esa carga que viene desde un JSON y  
# Lo va a convertir en un diccionario, y si el parámetro es  
# es del tipo de una clase (Ej: Moto) automáticamente lo convierte  
# a una instancia de esa clase.
```

Resumen Laboratorio IV - BackEnd (2022)

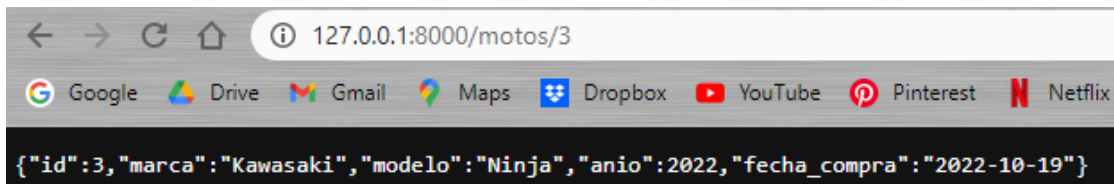
```
@app.post('/crear_moto')
def crear_moto(m: Moto): #Recibe una Moto
    #Espera un JSON con Los datos para crear una Moto.
    motos.append(m) #Agrega una moto a La Lista
    return 201, 'Moto creada correctamente'
#El return devuelve un cod HTTP y un msj.
```



El mensaje que nos devuelve postman es:

```
[
  201,
  "Moto creada correctamente"
]
```

Para corroborar pedimos todas las motos:



Si en alguna variable le erro al tipo por ejemplo, me devuelve un JSON con el formato de error:

```
{
  "id": 3,
  "marca": "Kawasaki",
  "modelo": "Ninja",
  "anio": "Acá mando fruta",
  "fecha_compra": "2022-10-19"
}
```

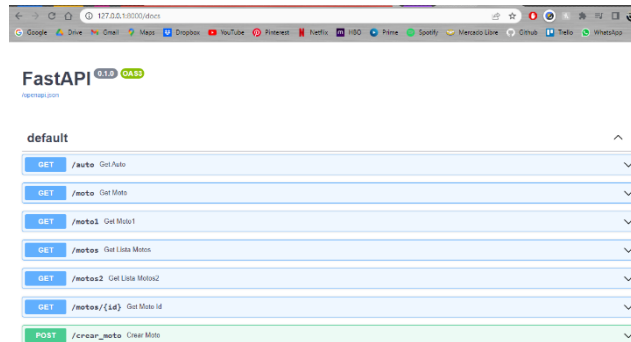
Respuesta de postman:

```
{
  "detail": [
    {
      "loc": [
        "body",
        "anio"
      ],
      "msg": "value is not a valid integer",
      "type": "type_error.integer"
    }
  ]
}
```


Resumen Laboratorio IV - BackEnd (2022)

FastAPI se fija en el modelo base y valida contra él, verifica que los valores que envío como parámetros en el JSON se puedan convertir en una Moto, es decir que se correspondan. En el caso que no pueda convertirlo porque los datos no son correctos envía una respuesta con un mensaje detallado, esto lo podemos ver en POSTMAN.

Además, FastAPI puede generar la **documentación** de la aplicación de forma automática. Eso podemos verlo a través del endpoint `"/docs"`. Lo que hace es generar un JSON, que es un archivo estándar que se llama OpenAPI (Antes Swagger), ese formato es un JSON que especifica todo lo que define a la API. También nos permite ejecutar los endpoint, agregar parámetros, etc.

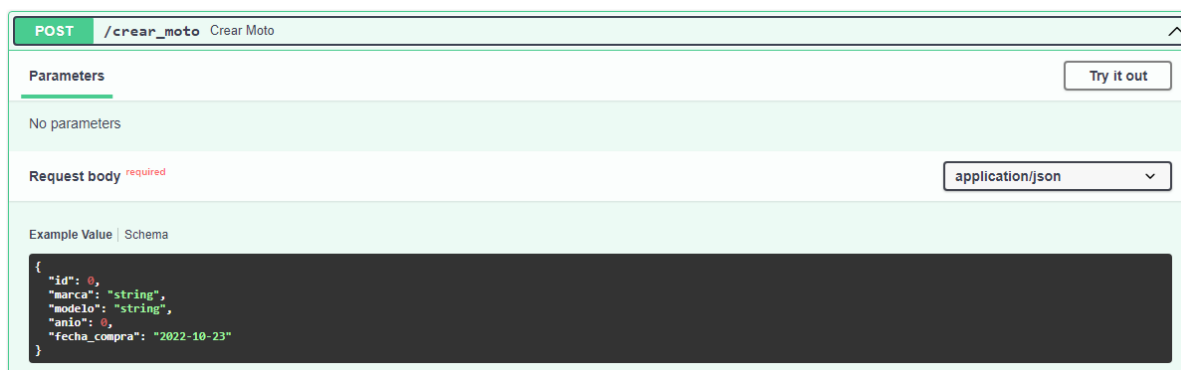


Si hacemos click en `openapi.json` nos devuelve: (Podríamos tomar ese código y utilizar herramientas para convertirlo en una aplicación de java, c#, etc.)



```
{
  "openapi": "3.0.2",
  "info": {
    "title": "FastAPI",
    "version": "0.1.0"
  },
  "paths": {
    "/auto": {
      "get": {
        "summary": "Get Auto",
        "operationId": "get_auto_auto_get",
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {}
              }
            }
          }
        }
      }
    },
    "/moto": {
      "get": {
        "summary": "Get Moto",
        "operationId": "get_moto_moto_get",
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {}
              }
            }
          }
        }
      }
    },
    "/moto1": {
      "get": {
        "summary": "Get Moto1",
        "operationId": "get_moto1_moto1_get",
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {}
              }
            }
          }
        }
      }
    },
    "/motos": {
      "get": {
        "summary": "Get Lista Motos",
        "operationId": "get_lista_motos_motos_get",
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {}
              }
            }
          }
        }
      }
    },
    "/motos2": {
      "get": {
        "summary": "Get Lista Motos2",
        "operationId": "get_lista_motos2_motos2_get",
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {}
              }
            }
          }
        }
      }
    },
    "/motos/{id}": {
      "get": {
        "summary": "Get Moto Id",
        "operationId": "get_moto_id_motos_id_get",
        "parameters": [
          {
            "required": true,
            "schema": {
              "title": "Id",
              "type": "integer",
              "name": "id",
              "in": "path"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {}
              }
            }
          },
          "422": {
            "description": "Validation Error",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/HTTPValidationError"
                }
              }
            }
          }
        }
      }
    },
    "/crear_moto": {
      "post": {
        "summary": "Crear Moto",
        "operationId": "crear_moto_crear_moto_post",
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/Moto"
              }
            }
          },
          "required": true
        },
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {}
              }
            }
          },
          "422": {
            "description": "Validation Error",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/HTTPValidationError"
                }
              }
            }
          }
        }
      }
    }
  },
  "components": {
    "schemas": {
      "HTTPValidationError": {
        "title": "HTTPValidationError",
        "type": "object",
        "properties": {}
      },
      "Moto": {
        "title": "Moto",
        "required": [
          "id",
          "marca",
          "modelo",
          "anio"
        ],
        "type": "object",
        "properties": {
          "id": {
            "title": "Id",
            "type": "integer"
          },
          "marca": {
            "title": "Marca",
            "type": "string"
          },
          "modelo": {
            "title": "Modelo",
            "type": "string"
          },
          "anio": {
            "title": "Año",
            "type": "integer"
          },
          "fecha_compra": {
            "title": "Fecha Compra",
            "type": "string",
            "format": "date"
          }
        }
      },
      "MotoLista": {
        "title": "MotoLista",
        "required": [
          "marca",
          "modelo"
        ],
        "type": "object",
        "properties": {
          "marca": {
            "title": "Marca",
            "type": "string"
          },
          "modelo": {
            "title": "Modelo",
            "type": "string"
          }
        }
      },
      "ValidationError": {
        "title": "ValidationError",
        "required": [
          "loc",
          "msg",
          "type"
        ],
        "type": "object",
        "properties": {
          "loc": {
            "title": "Location",
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "msg": {
            "title": "Message",
            "type": "string"
          },
          "type": {
            "title": "Error Type",
            "type": "string"
          }
        }
      }
    }
  }
}
```

Podemos utilizar los métodos Post, Get, etc de FastAPI y no utilizar Postman.



Nos muestra los esquemas de validación, los tipos y nos marca con `"*"` los campos obligatorios.

Resumen Laboratorio IV - BackEnd (2022)

Schemas

HTTPValidationError >

```
Moto ▾ {
  id*          integer
               title: Id
  marca*       string
               title: Marca
  modelo*      string
               title: Modelo
  anio*        integer
               title: Anio
  fecha_compra string($date)
               title: Fecha Compra
}
```

Ejemplo de una agenda de contactos:

```
from imp import reload
from typing import Optional
import uvicorn
from fastapi import FastAPI
from pydantic import BaseModel
from datetime import date

app = FastAPI()

if __name__ == "__main__":
    uvicorn.run(f"{__name__}:app", reload=True)

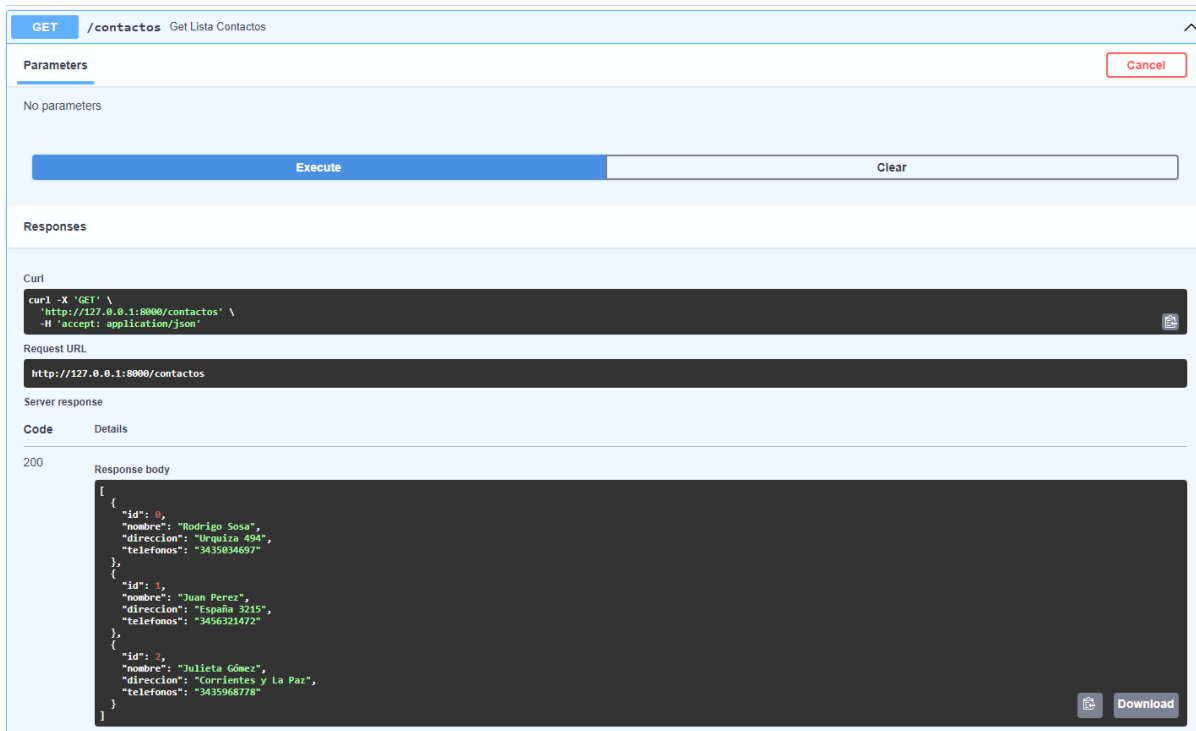
class Contacto(BaseModel):
    id: int
    nombre: str
    direccion: str
    telefonos: str

contactos = [
    Contacto( id=0, nombre = 'Rodrigo Sosa', direccion='Urquiza 494', telefonos = '3435034697'),
    Contacto( id=1, nombre = 'Juan Perez', direccion='España 3215', telefonos = '3456321472'),
    Contacto( id=2, nombre = 'Julietta Gómez', direccion='Corrientes y La Paz', telefonos = '3435968778')
]

#Devuelvo la lista de contactos completa
@app.get('/contactos') #Ruta de inicio
def get_lista_contactos():
    return contactos
```

Desde “/docs” puedo comprobar que el método get con **Try it out** y luego **Execute**:

Resumen Laboratorio IV - BackEnd (2022)



Creamos otros métodos para devolver un contacto según su id y para crear un nuevo contacto:

```
#Devuelvo un contacto según su id
@app.get('/contactos/{id}') #Ruta de inicio - Parámetro entre {}
def get_contacto_id(id: int): #el parámetro que pasamos en la ruta se tiene
    #llamar igual {id}=id
    c = contactos[id]
    return c

# * Otra forma menos eficiente porque genera otra lista en memoria cada vez que
# busca:

@app.get('/contactos2/{id}')
def get_contacto_id2(id: int):
    contactos_con_el_id = [m for m in contactos if m.id == id]
    if len(contactos_con_el_id) > 0:
        return contactos_con_el_id[0]
    return 404, "Contacto no encontrado"

# ** Otra forma utilizando en vez de una lista un generador (next):
@app.get('/contactos3/{id}')
def get_contacto_id3(id: int):
    #El generador no se ejecuta hasta que le digamos que se ejecute.
    contacto = next((m for m in contactos if m.id == id), None) #Si no
    #encuentra devuelve None
    if contacto:
        return contacto
    return 404, "Contacto no encontrado"

#En general este tipo de búsquedas no lo vamos a utilizar porque la búsqueda es
#tarea de la BBDD. Lo que vamos a realizar es una request.
```

```
#Método post para crear un contacto:
@app.post('/crear_contacto')
def crear_contacto(c: Contacto):
    contactos.append(c) #Agrega un contacto a la lista
    return 201, 'Contacto creado correctamente!' #El return devuelve un código
    HTTP y un msj.
```

Continuamos con este ejemplo separando la clase “contacto” siguiendo las siguiente metodología:

```
from pydantic import BaseModel
from typing import Optional

#ContactoCreate descende de BaseModel y eso nos da la
#posibilidad de usar tipos y validarlos en el ingreso
#y en el egreso.
class ContactoCreate(BaseModel): #Para el @post solo me pide estos datos
    nombre: str
    direccion: Optional[str]
    telefonos: Optional[str]

#La clase Contacto tiene un solo atributo que es el "id"
#pero hereda de "ContactoCreate", esto es para que cuando
#creamos una nueva instancia al endpoint le pasamos como parámetro
#el tipo que queremos pasarle como dato, ese tipo en los endpoint de
#tipo POST (crear) no deberían tener el id (porque el id es algo que se crea
En el servidor) Entonces creamos una clase para el "create - POST" que tiene
todos los atributos, menos el "id". Pero cuando lo quiero devolver con un GET,
#lo que hago es definir otra clase más (que también hereda de BaseModel) y
# que ya tiene todos los atributos más el "id", entonces hago otra clase
#que hereda de ContactoCreate y le agrega el "id", Contacto queda:

class Contacto(ContactoCreate): #Hereda de ContactoCreate y de BaseModel
    id: int #Suma a los atributos de ContactoCreate el atributo "id"
```

Además, también se usa un modelo para usar en las listas, ya que en las listas no mostramos todos los atributos de la clase. Es una buena práctica ya que, si en algún momento quiero mostrar algo más o algo menos en la lista que devuelve, solo modifico ese modelo de clase. Entonces, para definir una clase que hereda de otra, pero no agrega nada se utiliza la palabra “**pass**”, sin eso python no nos va a aceptar la definición de la clase. Pass es una palabra clave que internamente agrega una instrucción en la definición de la clase.

```
#Clase que hereda de ContactoCreate y muestra lo mismo con pass
class ContactoLista(ContactoCreate):
    pass
```

De esta forma tenemos 3 clases: 1 modelo para las listas (**ContactoLista**), un modelo para crear que tiene 3 atributos (**ContactoCreate**) y un modelo completo que además tiene un “id”. (**Contacto**).

Ahora vamos a crear un array que por el momento será nuestra “Base de Datos”, en ella creamos instancias de la clase completa (**Contacto**). Esto lo ubicamos en un nuevo archivo llamado **store.py**.

Resumen Laboratorio IV - BackEnd (2022)

```
#archivo: store.py

from contacto import Contacto

contactos = [
    Contacto( id=0, nombre = 'Rodrigo Sosa', direccion='Urquiza 494', telefonos
= '3435034697'),
    Contacto( id=1, nombre = 'Juan Perez', direccion='España 3215', telefonos =
None),
    Contacto( id=2, nombre = 'Julieta Gómez', direccion='Corrientes y La Paz',
telefonos = '3435968778')
]
```

Ahora revisamos los endpoints en el archivo principal “agenda.py”. Creo un endpoint para que muestre un mensaje al inicio:

```
#archivo: agenda.py

@app.get('/') #Ruta
def home():
    #Muestra un msj al inicio
    return "Bienvenido! Esta es la página de inicio"
```

Con respecto al endpoint que trae la lista de contactos del store, importamos la clase y definimos el endpoint, el return **devuelve una lista (por eso los corchetes)**, con respecto a la sintaxis del for es “cada elemento de la lista va a ser x, para x dentro de la lista contactos, es decir recorre la lista elemento por elemento y lo agrega a la nueva lista, realiza una **copia por comprensión** (Sintaxis de python). Tener cuenta de que, **si hacemos return contactos, lo que hacemos es devolver la lista original y no la copia.**

```
#archivo: agenda.py

from store import contactos

#Devuelvo la lista de contactos completa
@app.get('/contactos') #Ruta (Por conversión en plural)
def get_all_contactos():
    #Devuelve una copia de la lista con todos los contactos
    return [x for x in contactos]
```

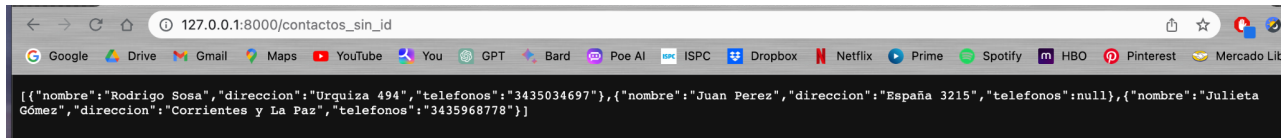
Devuelve una lista en JSON donde cada elemento de esta lista es de tipo “Contacto”

```
[{"id":0,"nombre":"Rodrigo Sosa","direccion":"Urquiza 494","telefonos":"3435034697"}, {"id":1,"nombre":"Juan Perez","direccion":"España 3215","telefonos":null}, {"id":2,"nombre":"Julieta Gómez","direccion":"Corrientes y La Paz","telefonos":"3435968778"}]
```

Ahora vamos a usar la clase “**ContactoLista**” que es la que creé para devolver una lista de contactos, pero sin el id. Para eso modificamos el endpoint definiendo un **modelo de respuesta** gracias a FastApi. Es común tener clases con modelos especiales para utilizarlos como **response** a los diferentes endpoints, pueden tener menos o más datos o mostrarlos de forma diferente de cómo se encuentran en la BBDD. *FastAPI realiza la conversión automática.*

Resumen Laboratorio IV - BackEnd (2022)

```
# Uso un modelo de respuesta ContactoLista
@app.get('/contactos_sin_id', response_model=list[ContactoLista])
def get_all_contactos():
    # Devuelve una copia de la lista contactos
    return [x for x in contactos]
```



Luego tenemos el endpoint que devuelve todos los datos de un **contacto particular según su "id"**. Le pasamos el id en el path, entonces el código busca el contacto en la lista y si lo encuentra devuelve el contacto con todos sus atributos, por eso **usa el response_model de Contacto**. Esto es generalmente utilizado para traer todos los detalles de un solo contacto para mostrarlos o editarlos. En este caso uso una función para buscar el contacto y retornarlo, la sintaxis: **encontradas = [x for x in contactos if x.id == id]** me crea una lista nueva con solamente los contactos que tienen el id igual al id que le paso a la función buscar_contacto(id). **También hay que tener en cuenta que para utilizar el status_code de error 404 hay que importar status de la librería FastAPI.**

```
from fastapi import FastAPI, Response, status

#Devuelvo un contacto según su id (Retorna una lista de un elemento o un error)
@app.get('/contactos/{id}', response_model=Contacto) #Uso el model Contacto xq
devuelvo todos los atributos
def get_contacto(id: int): #el parámetro que pasamos en la ruta se tiene q' llamar
igual {id}=id
    c = buscar_contacto(id)
    if c is None:
        return Response(content = 'No encontrado!', status_code=
status.HTTP_404_NOT_FOUND)
    return c

#Función para buscar un contacto.
def buscar_contacto(id: int):
    encontrados = [x for x in contactos if x.id == id] #Devuelve otra lista
    if len(encontrados) == 0:
        return None #No existe un contacto con ese id.
    if len(encontrados) > 1: #Si hay más de uno lanzo una excepción.
        raise Exception('Algo está mal: Hay más de un contacto con el mismo
id!')
    return encontrados[0] #Si encontró un solo contacto devuelvo el único
elemento
```

Ejemplo de error 404: Con este código podemos avisarle al Front que hay un error y mostrar un mensaje personalizado.

✓ TERMINAL

INFO: 127.0.0.1:1896 - "GET /contactos/10 HTTP/1.1" 404 Not Found

No encontrado!

Resumen Laboratorio IV - BackEnd (2022)

Seguimos con el **método agregar (POST)**, acá si bien usamos la misma ruta que el método GET que trae la lista de contactos, al estar precedido por el verbo POST *se da cuenta que su función es crear un contacto*. En la función que va a responder a este POST tengo la obligación de decirle dónde me va a poner FastAPI los datos que le paso, entonces se hace con un parámetro que se puede llamar como quiera (**ítem** en este caso), y hay que pasarle el tipo (**ContactoCreate** en este ej.), *ese tipo descende de BaseModel, entonces el JSON que tome FastAPI intentará adecuarlo a ese tipo*, en una instancia de esa clase y va a verificar que los tipos coincidan y existan los atributos obligatorios (nombre, por ejemplo).

Ahora, la lista que nosotros tenemos tiene elementos de tipo **Contacto**, no ContactoCreate (No tiene id) que es lo que estamos queriendo agregar entonces, voy a **crear una instancia de tipo Contacto**, donde todos los atributos de la instancia Contacto los traslado desde lo que me están pasando en el POST. **El id se construye llamando a una función que se define mas abajo y lo que hace es buscar el id más alto y le suma 1** (En una BBDD esto se haría con un generador por ej.)

```
#Método post (Otra forma):
@app.post('/contactos') #Aunque la ruta sea la misma que traer la lista, como
tiene un verbo POST se da cuenta que tiene que crear un contacto.
def agregar_contacto(item: ContactoCreate, response: Response):
    try: #Se ejecuta si no hay error
        c=Contacto(id = new_id(), nombre = item.nombre,
                    direccion = item.direccion, telefonos = item.telefonos)
        contactos.append(c) #Agrega un contacto a la lista, en BBDD sería un
INSERT de SQL
        response.status_code = status.HTTP_201_CREATED
        return 'Contacto creado correctamente.'
    except Exception as ex: #Si hay un error sale x acá.
        response.status_code = status.HTTP_500_INTERNAL_SERVER_ERROR
        return 'Error al agregar el contacto' + str(ex)

#Generador de id, busca el más alto en la lista y le suma 1.
def new_id():
    id = 0
    for c in contactos:
        if c.id>id:
            id = c.id
    return id + 1
```

Teoría sobre los generadores de números (serial) de las bases de datos: Estos generadores auto incrementales que utilizan las BBDD **no vuelven hacia atrás y están fuera de las transacciones**. Se genera un número y se avanza, no le importa si nosotros lo usamos o no, en el caso de no haberlo utilizado va a quedar un “hueco” sin usar, los ids que queden van a ser únicos, por ejemplo si borramos un objeto con id=2, y agregamos otro, va a ser id=3, no va a volver atrás y generar otro objeto con id=2. Es bueno para asegurarnos que todos los IDs son únicos, aunque esté agregando datos de 10 máquinas diferentes, se generan en el servidor cuando se pide grabar el dato, si no se pudo grabar se pierde el id y queda un hueco en la serie, pero no afecta a los demás. Es decir generamos una secuencia de valores siempre creciente y de valores únicos, pero puede no contener valores dentro de esa secuencia. Es por esto que **nunca se debe usar un autonumérico del servidor con datos que deben ser auditables**, por ejemplo los casos de números de facturas, porque estos no pueden tener faltantes, tienen que poder ser auditables. Se puede usar para el id de factura pero no para el número de factura.

En el método POST tenemos una variable “response” que permite crear una respuesta con es status_code y un content.

Resumen Laboratorio IV - BackEnd (2022)

Si lo probamos desde **/docs** vemos que si lo ejecutamos nos da un modelo para modificar, eso lo sabe porque tiene un `response_model` que le dice que es lo que tiene que contener el JSON y el tipo de variable.

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** /contactos
- Parameters:** No parameters
- Request body:** application/json (required). The body contains a JSON object:

```
{  "nombre": "Nuevo contacto",  "direccion": "una dirección",  "telefonos": "343434343"}
```
- Request URL:** http://127.0.0.1:8000/contactos
- Server response:** 201 (undocumented). The response body is "Contacto creado correctamente."
- Response headers:** content-length: 32, content-type: application/json, date: Sun, 30 Oct 2022 20:17:54 GMT, server: uvicorn.
- Responses table:**

Code	Description	Links
200	Successful Response	No links

Para corroborar pido la lista de todos los contactos:

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** /contactos
- Parameters:** No parameters
- Request URL:** http://127.0.0.1:8000/contactos
- Server response:** 200. The response body is a JSON array of contact objects:

```
[  {    "nombre": "Rodrigo Sosa",    "direccion": "Urquiza 494",    "telefonos": "343503469"  },  {    "nombre": "Juan Perez",    "direccion": "España 3215",    "telefonos": null  },  {    "nombre": "Julietta Gómez",    "direccion": "Corrientes y La Paz",    "telefonos": "3435968778"  },  {    "nombre": "Nuevo contacto",    "direccion": "una dirección",    "telefonos": "343434343"  }]
```

Seguimos con el endpoint **DELETE** (Borra un elemento que existe por id):

```
@app.delete('/contactos/{id}')
def borrar_contacto(id: int):
    try:
        c = buscar_contacto(id) #Busca el contacto en la lista
        if not c: #Si no lo encuentra devuelve un 404
            return Response(content='No se encuentra el contacto!',
                             status_code=404)
        contactos.remove(c) #Si lo encuentra lo borro y devuelvo un 200
        return Response(content = 'Contacto borrado correctamente',
                         status_code=200)
```


Resumen Laboratorio IV - BackEnd (2022)

```
except Exception as ex: #Si hubo un error lanzó una excepción y devuelvo un  
500  
    return 'Error al borrar el contacto: ' + str(ex), 500
```

Curl

```
curl -X 'DELETE' \  
  'http://127.0.0.1:8000/contactos/2' \  
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/contactos/2
```

Server response

Code	Details
200	<p>Response body</p> <pre>["Contacto borrado correctamente", 200]</pre> <p>Response headers</p> <pre>content-length: 38 content-type: application/json date: Sun, 30 Oct 2022 20:37:20 GMT server: uvicorn</pre>

Por último, vamos a generar el **método PUT (Modificar)**. El **PUT** es similar al **POST** ya que tenemos que mandar un JSON con los nuevos datos, se va a convertir en una clase, tenemos que buscar esa instancia para ver si **existe** y en el caso de que no exista *no se puede modificar y devolvemos un 404*. En el caso de que exista, lo modificamos, lo grabamos y devolvemos un código **200**. Si hay un error devuelve un código **500**. Es importante utilizar **Response** en las respuestas de los endpoint para poder interpretar los códigos de error en el Front.

```
@app.put('/contactos/{id}') #Le pasamos el id que tiene que buscar  
def modificar_contacto(id: int, item: ContactoCreate): #Tengo un id y además en  
el cuerpo del Request recibo un ContactoCreate (JSON sin id)  
    try: #Toma el id que le pasé en la ruta y busca el contacto  
        c: Contacto = buscar_contacto(id)  
        if not c: #Si c es None devuelve un response 404.  
            return Response('No se encuentra el contacto', 404)  
        #Si se encontró un contacto con ese id, reemplazo los valores por los que  
vienen del JSON (Menos el id que queda igual)  
        c.nombre = item.nombre #Reemplaza por el nuevo valor que viene en la  
Request  
        c.direccion = item.direccion #Reemplaza por el nuevo valor que viene en  
la Request  
        c.telefonos = item.telefonos #Reemplaza por el nuevo valor que viene en  
la Request  
        return Response('Contacto modificado correctamente', 200) #Siempre usar  
Response en las respuestas  
    except Exception as ex:  
        return Response('Error al modificar el contacto' + str(ex), 500)
```

Podemos verificarlo desde FastAPI, el método **PUT** nos pide un **id** y el cuerpo de la **Request** en formato **JSON** de la siguiente forma:

Resumen Laboratorio IV - BackEnd (2022)

Si queremos modificar el Contacto con id=2 (Juan Perez)

Response body

```
[
  {
    "nombre": "Rodrigo Sosa",
    "direccion": "Urquiza 494",
    "telefonos": "3435034697"
  },
  {
    "nombre": "Juan Perez",
    "direccion": "España 3215",
    "telefonos": null
  },
  {
    "nombre": "Julieta Gómez",
    "direccion": "Corrientes y La Paz",
    "telefonos": "3435968778"
  }
]
```

Name Description

id * required

integer
(path)

2

Request body *required*

```
{
  "nombre": "Juan Perez",
  "direccion": "España 3215",
  "telefonos": "343455645",
  "id": 2
}
```

Server response

Code

Details

200

Response body

Unrecognized response type; displaying content as text.

Contacto modificado correctamente



Download

Response headers

```
content-length: 33
date: Mon, 31 Oct 2022 00:08:04 GMT
server: uvicorn
```

El **PUT** según especificación, si se ejecuta tiene que reemplazar la instancia completa (todos los valores) y no fijarse si algún se modificó y otros no, se cambia por una nueva instancia directamente. Entonces, **si queremos que antes de modificar compruebe si los valores cambiaron usamos el método PATCH**, este reemplaza solamente los valores que se mandaron en el cuerpo de la Request.

```
#Patch modifica solo los valores que vienen en la request (Hay APIs que no lo implementan)
@app.patch('/contactos/{id}') #Le pasamos el id que tiene que buscar
def modificar_contacto(id: int, item: ContactoCreate): #Tengo un id y además en el cuerpo del Request recibo un ContactoCreate (JSON sin id)
    try: #Toma el id que le pasé en la ruta y busca el contacto
        c: Contacto = buscar_contacto(id)
        if not c: #Si c es None devuelve un response 404.
            return Response('No se encuentra el contacto', 404)
        #Si se encontró un contacto con ese id, reemplazo los valores por los que vienen del JSON (Menos el id que queda igual)
        c.nombre = item.nombre #Reemplaza solamente si viene en la Request
        c.direccion = item.direccion #Reemplaza solamente si viene en la Request
        c.telefonos = item.telefonos #Reemplaza solamente si viene en la Request
        return Response('Contacto modificado correctamente', 200) #Siempre usar Response en las respuestas
    except Exception as ex:
        return Response('Error al modificar el contacto' + str(ex), 500)
```

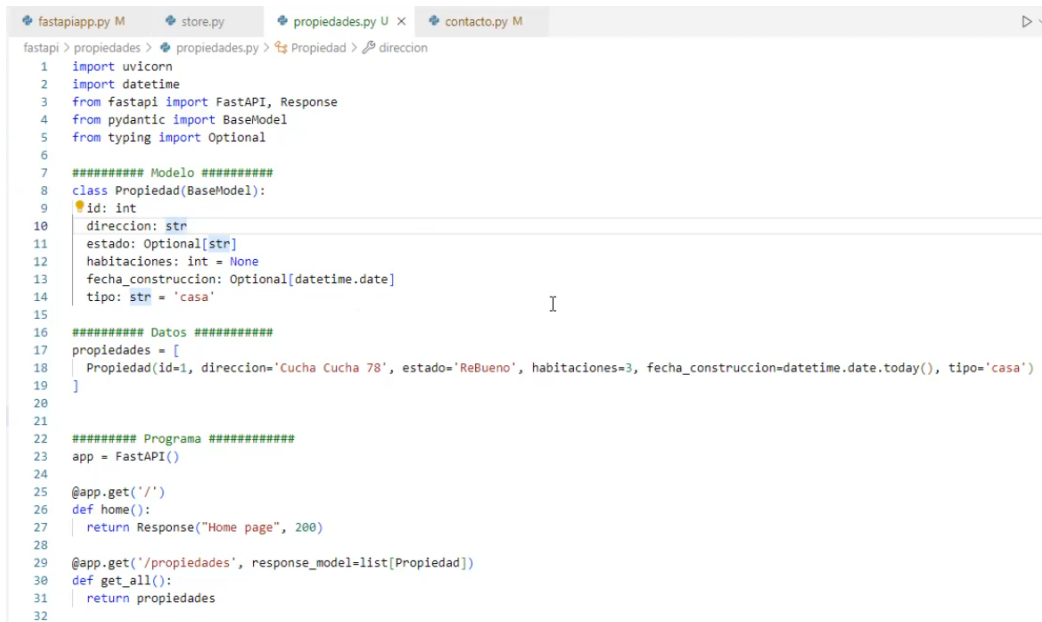
Resumen Laboratorio IV - BackEnd (2022)

Muchas APIs no implementan el patch y nosotros no lo usaremos ahora porque en la aplicación de React vamos a tener todos los datos, los que no se modificaron quedarán como estaban y los que se modificaron cambiarán, es decir directamente pisamos todos los datos con PUT. Tener en cuenta que, cuando usamos PostMan y si no ponemos todos los datos va a pisar la instancia con datos incompletos.

Otro ejemplo: Crear una API que tenga endpoints para:

- Devolver una lista de propiedades
- Devolver el detalle de una propiedad
- Agregar una propiedad nueva
- Modificar una propiedad existente
- Borrar una propiedad existente.

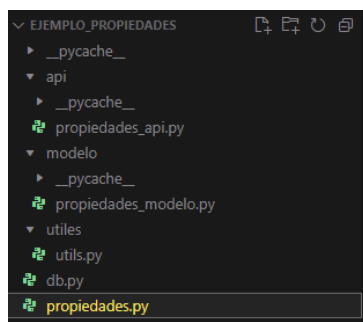
Lo ideal es comenzar dividiendo el código dentro de un mismo archivo en modelo, datos y programa. Luego cuando estamos seguros que funciona comenzamos a dividir en carpetas como API, Modelo, Utiles, y la db.



```
1 import uvicorn
2 import datetime
3 from fastapi import FastAPI, Response
4 from pydantic import BaseModel
5 from typing import Optional
6
7 ##### Modelo #####
8 class Propiedad(BaseModel):
9     id: int
10     direccion: str
11     estado: Optional[str]
12     habitaciones: int = None
13     fecha_construccion: Optional[datetime.date]
14     tipo: str = 'casa'
15
16 ##### Datos #####
17 propiedades = [
18     Propiedad(id=1, direccion='Cucha Cucha 78', estado='ReBueno', habitaciones=3, fecha_construccion=datetime.date.today(), tipo='casa')
19 ]
20
21 ##### Programa #####
22 app = FastAPI()
23
24 @app.get('/')
25 def home():
26     return Response("Home page", 200)
27
28 @app.get('/propiedades', response_model=list[Propiedad])
29 def get_all():
30     return propiedades
31
32
```

Una vez probado el código realizamos los siguientes pasos:

- 1) **Separación** del código en archivos y carpetas:



- 2) Código:
 - a. **propiedades.py** (Main)

```
#### imports ####
import uvicorn
```

Resumen Laboratorio IV - BackEnd (2022)

```
from fastapi import FastAPI, Response
from api.propiedades_api import apiprops #Importamos la instancia de APIRouter

#### Programa ####
app = FastAPI()
app.include_router(apiprops) #vinculamos el contenedor de apis que está en
"propiedades_api" *Esto se ve en detalle en el archivo "propiedades_api"

#Solo ubico acá el endpoint de inicio (Después se traslada a propiedades_api)
@app.get('/') #Pagina Inicio
def home():
    return Response('Home page', 200)

if __name__ == "__main__":
    uvicorn.run('propiedades:app', reload=True)
```

b. **db.py** (Simula una base de datos. También contiene métodos para recorrerla)

```
#### Imports ####

from modelo.propiedades_modelo import Propiedad
import datetime

#### Datos ####
propiedades = [
    Propiedad(id=1, direccion='Urquiza 494', estado='A estrenar',
habitaciones=1, fecha_construccion = datetime.date(1980,7,5),
tipo='Departamento'),
    Propiedad(id=2, direccion='Laprida 83', estado='Antigua', habitaciones=2,
fecha_construccion = None, tipo='Departamento'),
    Propiedad(id=3, direccion='España 8974', estado='Muy bueno',
habitaciones=3, fecha_construccion = datetime.date.today(), tipo='Casa')
]

#### Funciones internas ####

def buscar_propiedad(id: int):
#Busca la propiedad en la lista por si id, si no encuentra el id devuelve None
    return next((x for x in propiedades if x.id == id), None) #Los () crean un
generador,
    #La diferencia es que no se crea una nueva lista en la memoria, si no q
espera al método
    # next(), busca valor x valor, y si no lo encuentra asigna el valor x
defecto, None en este caso

#Crea un nuevo id buscando el id + alto y sumando 1
def nuevo_id():
    resultado = 0
    for p in propiedades: #Recorre el array y devuelve la última propiedad
        if p.id>resultado: #Verifica si el id de la última prop. es > 0
```

Resumen Laboratorio IV - BackEnd (2022)

```
resultado = p.id #De ser así se lo asigna a resultado
return resultado+1 #Al resultado le suma 1 y lo retorna.
```

c. propiedades_modelo.py (Contiene los modelos de respuesta y objetos)

```
#### Imports ####

from pydantic import BaseModel
from typing import Optional
import datetime

#### Modelos ####

class PropiedadBase(BaseModel):
    #https://pydantic-docs.helpmanual.io/usage/types/ (VER)
    #En general el id no se usa en la propiedadBase porque lo asigna el
    servidor
    direccion: str
    estado: Optional[str] #Optional de pydantic
    tipo: str = "Casa" #Por defecto asigna "Casa"

class PropiedadSinId(PropiedadBase): #Utilizado para el post
    habitaciones: int = None #Es otra alternativa a Optional darle un valor
    None x defecto
    fecha_construccion: Optional[datetime.date] #Formato utilizado para fechas
    en pydantic (pydantic-docs)

class Propiedad(PropiedadSinId):
    #Se utiliza solo para salidas, entonces no necesita validaciones de entrada
    id: int

class PropiedadLista(PropiedadBase):
    #Se utiliza solo para salidas, entonces no necesita validaciones de entrada
    id: int #Solo agregado para ver los id de la lista y comprobar, luego
    sacar.
    #pass (Luego de sacar el id solo colocamos el pass)
```

d. propiedades_api.py (Contiene los endpoints de la API)

Para organizar/separar el código FastAPI nos ofrece la herramienta **APIRouter**, que sirve para *agrupar endpoints*. Declaramos una variable "**apiprops**" por ej. e instanciamos un APIRouter, esta variable nos sirve para contener los endpoints, métodos que respondan a URLs, de esta forma le decimos que todas estas api (endpoints) están dentro de ese APIRouter, en esa instancia llamada apiprops:

```
#### Imports ####
from fastapi import Response, status, APIRouter
from modelo.propiedades_modelo import Propiedad, PropiedadSinId, PropiedadLista
from db import propiedades, nuevo_id, buscar_propiedad
```

```

apiprops = APIRouter(prefix='/propiedades') #Luego esto tenemos que vincularlo
al "app" que está en el main.

#El atributo prefix establece un prefijo para los endpoints ya que generalmente
se repiten ej: propiedades/ contactos/ autos/ etc. Entonces en vez de ser
@app.get... se reemplaza por @apiprops.get()... Ahora tengo todos los endpoint
de las propiedades metidos dentro de un solo objeto llamado "apiprops".
Entonces esto debo agregarlo al "app" que se encuentra en el main, esto lo
hacemos en el archivo principal del programa de la siguiente forma:
app.include_router(apiprops) y se importa como "from api.propiedades_api import
apiprops"

#El prefix se ubica en el primer parámetro del endpoint '' = '/propiedades':
@apiprops.get('', response_model=list[PropiedadLista])#Devuelve con el formato
de Model "PropiedadLista"
def get_all():
    return propiedades

@apiprops.get('/{id}', response_model=Propiedad) #Devuelvo con modelo completo
(Propiedad)
def get_propiedad(id: int):
    p = buscar_propiedad(id)
    if p == None:
        return Response('Propiedad no encontrada', status.HTTP_404_NOT_FOUND)
    return p

@apiprops.post('') #Crea propiedades
def create_propiedades(nueva_propiedad:PropiedadSinId): #Viene una Propiedad
sin id
#Creo una instancia de Propiedad y le asigno el id que retorna la función
nuevo_id()
#y asigno los valores que vienen de la request.
p = Propiedad(id=nuevo_id(), #Asigna un id
    direccion=nueva_propiedad.direccion,
    estado=nueva_propiedad.estado,
    habitaciones=nueva_propiedad.habitaciones,
    fecha_construccion=nueva_propiedad.fecha_construccion,
    tipo=nueva_propiedad.tipo)
#Agrega la propiedad creada a la lista
propiedades.append(p)
#Retorna con un mensaje y un código 201
return Response('Propiedad agregada correctamente!', status_code=201)

@apiprops.delete('/{id}') #Borra Propiedades
def delete_propiedad(id:int):
    try:
        p = buscar_propiedad(id)
        if not p:

```

```

        return Response(content='No se encuentra la propiedad',
status_code=404)
    propiedades.remove(p)
    return Response(content='Propiedad borrada correctamente',
status_code=200)
except Exception as ex:
    return Response(content='Error al borrar la propiedad' + str(ex),
status_code=500)

@apiprops.put('/{id}') #Modifica Propiedades
def update_propiedad(id:int, item: PropiedadSinId):
    try:
        p: Propiedad = buscar_propiedad(id)
        if not p:
            return Response(content='No se encuentra la propiedad',
status_code=404)

        p.direccion=item.direccion,
        p.estado=item.estado,
        p.habitaciones=item.habitaciones,
        p.fecha_construccion=item.fecha_construccion,
        p.tipo=item.tipo

        return Response('Propiedad modificada correctamente!', status_code=200)
    except Exception as ex:
        return Response(content='Error al modificar la propiedad' + str(ex),
status_code=500)

```

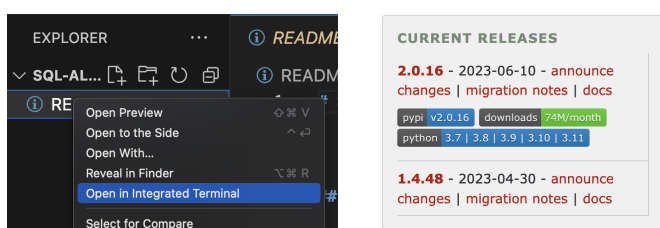
FastAPI integrado con SQL Alchemy:

SQL Alchemy es una librería que tiene dos funciones, por un lado permite *acceder a base de datos*, ejecutar comandos, obtener información de la BBDD, etc. y por otro lado tiene una API que permite hacer el mapeo de las clases de python a tablas en la BBDD relacional, es decir *es un ORM* (Object Relational Mapping). Trabaja de forma declarativa, es decir que cuando definimos las clases podemos decir que atributo se mapea en que tabla y en qué columna y como se van a relacionar entre ellos. Es decir nosotros trabajamos con clases, y al momento de grabar le decimos a SQL Alchemy que nos grabe la instancia y el sabe a dónde va cada atributo del objeto, con que PK, FK, etc. También sirve para hacer consultas a la BBDD, SQL Alchemy realiza los select, join, where, etc.

La versión que vamos a utilizar es la 1.4.48: <https://docs.sqlalchemy.org/en/14/>

Instalación SQL Alchemy: <https://docs.sqlalchemy.org/en/14/intro.html>

Creo una carpeta “sql-alchemy-orm” y abro un terminal en la misma carpeta. Lo podemos hacer por comandos cd, ls, etc o con botón derecho en Code y “open integrated terminal”



Ejecutamos los comandos:

```
pip install SQLAlchemy
```

Podemos verificar la versión:

```
(base) negrux@Rodrigos-MBP sql-alchemy-orm % python
Python 3.10.9 (main, Mar 1 2023, 12:33:47) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlalchemy
>>> sqlalchemy.__version__
'1.4.39'
>>>
```

Podemos actualizar la versión:

```
pip install SQLAlchemy --upgrade
```

1. Creamos un archivo “contactosBd.py”
2. Ahora vamos a conectar a la base de datos con una función “create_engine”: Esta función, cuando la ejecutemos con ciertos parámetros va a crear la conexión y nos va a devolver un objeto “engine” que representa la conexión y es lo que vamos a usar para comunicarnos con la base de datos.
3. Para crear el engine creamos una variable y llamamos a la función “create_engine()” en la cual pasamos los siguientes parámetros:
 - a. Una URL, es decir un string de conexión, este tiene un formato con la siguiente sintaxis:

```
>>> from sqlalchemy import create_engine
```

```
>>> engine = create_engine("sqlite+pysqlite:///memory:", echo=True) donde: pysqlite= Es el
driver específico que va a usar SQL Alchemy, sqlite= Es el tipo de BBDD, ///:memory: Indica que se
guarda en memoria, echo=True: Me muestra en la consola todas las instrucciones SQL que va
generando (Sirve para debug, pero se saca para producción porque llena el log de msg.)
```

Este string le indica con qué dialecto va a trabajar la BBDD.

<https://docs.sqlalchemy.org/en/14/tutorial/engine.html>,

<https://docs.sqlalchemy.org/en/14/dialects/index.html> (Dialectos es el lenguaje SQL que utiliza cada BBDD, cada uno tiene su particularidad)

- b. Obtenemos la conexión: https://docs.sqlalchemy.org/en/14/tutorial/dbapi_transactions.html

```
>>> from sqlalchemy import text (Importamos text)
```

```
>>> with engine.connect() as conn:
```

```
... result = conn.execute(text("select 'hello world'"))
```

```
... print(result.all()) El método “all()” devuelve todo lo que obtuvo desde el servidor en el formato
de una tupla por cada registro que viene del servidor.
```

- c. Ejecutamos el código para verificar la conexión: (Si muestra el “Hola mundo!” funciona. Lo que devuelve es una tupla de la BBDD por eso tiene la coma al final. De otro modo sería una cadena de String)

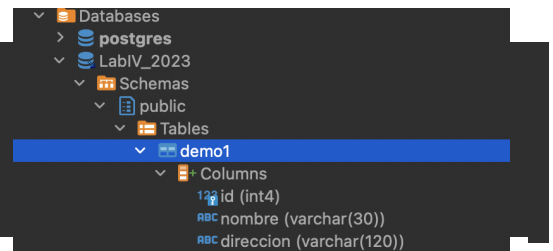
Resumen Laboratorio IV - BackEnd (2022)

```
from sqlalchemy import create_engine, text
engine = create_engine("sqlite+pysqlite:///memory:", echo=True)
with engine.connect() as conn:
    result = conn.execute(text("select 'Hola Mundo!'"))
print(result.all())
```

▶ (base) negrux@Rodrigos-MBP sql-alchemy-orm % /Users/negrux/Applications/anaconda3/bin/python "/Users/negrux/Proyecto Final LabIV/UTN - Laboratorio IV (2022)/5_Backend/Backend-4_Clase_021122/sql-alchemy-orm/contactosBd.py"
2023-06-20 17:42:59,505 INFO sqlalchemy.engine.Engine select 'Hola Mundo!'
2023-06-20 17:42:59,505 INFO sqlalchemy.engine.Engine [generated in 0.00015s] ()
[('Hola Mundo!'),]
▶ (base) negrux@Rodrigos-MBP sql-alchemy-orm %

- d. El objeto “engine” obtiene conexiones con la BBDD, cuando se deja de usar, se deben cerrar o destruir las conexiones, eso lo realiza con el “with”, ya que se crea una variable “conn” que solo existe en el ámbito de “with”, y una vez utilizada (Se ejecuta el bloque de código) se cierra la conexión y se libera o se devuelve a un pool de conexiones. Como la apertura y cierre de conexiones lleva un tiempo, se trata de no abrir y cerrar en reiteradas ocasiones, entonces se crea un pool de conexiones, que es un listado de conexiones abiertas que espera que se vuelva a necesitar, se reutiliza la conexión anterior. Por eso hay que utilizar siempre la estructura de **with** y metemos todo el código dentro, para trabajar bien con el pool si es que lo tenemos. Usando with no necesitamos cerrar la session (**session.close()**) ya que with lo hace automáticamente al terminar el bloque de código.
4. Una vez que ya probamos la conexión, podemos levantar un servidor **Postgres** para crear una base de datos: “LabIV_2023”, dentro creamos una tabla:

```
create table demo1(
id int primary key,
nombre varchar(30) not null,
direccion varchar(120)
)
```



5. La sintaxis de conexión en Postgres es:
`postgresql+psycopg2://user:password@host:port/dbname[?key=value&key=value...]` Hay que instalar el driver psycopg2 primero. Para eso en la terminal hacemos: `pip install psycopg2`, luego adaptamos la conexión a nuestro proyecto:

```
engine = create_engine("postgresql+psycopg2://postgres:admin@localhost/LabIV_2023", echo=True)
```

```
contactosBd.py > ...
1 from sqlalchemy import create_engine, text
2
3 engine = create_engine(
4     "postgresql+psycopg2://postgres:admin@localhost/LabIV_2023", echo=True)
5
6 with engine.connect() as conn:
7     result = conn.execute(text("select * from demo1"))
8     print(result.all())
9
```

<https://docs.sqlalchemy.org/en/14/dialects/postgresql.html#module-sqlalchemy.dialects.postgresql.psycopg2>

Luego agregamos dos valores a la tabla demo1 desde postgres y los vamos a poder visualizar con la instrucción “select * from demo1” de nuestro código:

Resumen Laboratorio IV - BackEnd (2022)

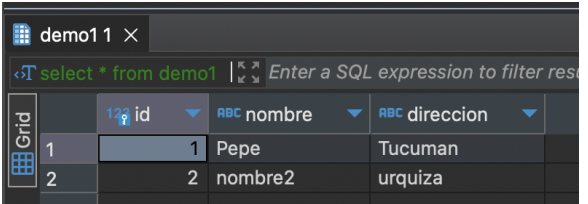
2023-06-20 19:24:49,037 INFO sqlalchemy.engine.Engine select * from demo1

2023-06-20 19:24:49,038 INFO sqlalchemy.engine.Engine [generated in 0.00016s] {}

[(1, 'Pepe', 'Tucuman'), (2, 'nombre2', 'urquiza')]

(base) negrux@Rodrigos-MBP sql-alchemy-orm %

6. Devuelve una lista de registros. Cada registro es una tupla, y dentro de cada tupla cada valor corresponde a una columna.



	id	nombre	direccion	
1	1	Pepe	Tucuman	
2	2	nombre2	urquiza	

En nuestro caso no lo vamos a hacer de forma directa, sino que vamos a usar el ORM. Lo que si vamos a utilizar es el “engine”.

Entonces vamos a necesitar importar nuevas funciones como la función “**declarative_base**”, esta nos va a devolver una instancia (metaclass). Luego para que SQL Alchemy se entere que la queremos mapear una clase propia a una tabla, hay que hacerla heredar de la clase “**Base**” (Se dice base declarativa) y luego le indicamos cómo mapear la base de datos desde SQL Alchemy. En el caso de trabajar con una tabla que ya está presente en la BBDD tenemos que tener atención en que los valores que colocamos en código coincidan con los de la tabla, de lo contrario generará errores.

```
from sqlalchemy import Column, Integer, String, create_engine
from sqlalchemy.orm import declarative_base
Base = declarative_base()

class Contacto(Base): #Hereda de la clase Base. Luego le decimos como debe mapear esa
    clase.

    __tablename__ = "contactos" #(Nombre de la tabla)
    id = Column(Integer, primary_key = True) #(Mapeo de una columna)
    nombre = Column(String)

engine = create_engine(
    "postgresql+psycopg2://postgres:admin@localhost/LabIV_2023", echo=True)
Base.metadata.create_all(engine)
```

Esto sirve ya que cuando programamos, en general vamos a hacer un diseño de clases, no un diseño de tablas. Entonces uno diseña todas las clases con sus relaciones y luego decide a donde mapear y en qué tabla. Lo que hacemos es decirle a SQL Alchemy, interpreteme esta clase que creé y créame la tabla, o mapea este objeto a tales tablas. Esto lo hacemos con **Base.metadata.create_all(engine)**, este create_all() toma todas las definiciones de clases que tenemos anotadas y que heredan de Base y va a tratar de crearlas donde se le indique en el “engine”.

Ejecutamos:

Resumen Laboratorio IV - BackEnd (2022)

```
contactosBd.py > ...
1 from sqlalchemy import Column, Integer, String, create_engine
2 from sqlalchemy.orm import declarative_base
3
4 Base = declarative_base()
5
6
7 # Hereda de la clase Base. Luego le decimos como debe mapear esa clase.
8 class Contacto(Base):
9     __tablename__ = "contactos" # (Nombre de la tabla)
10    id = Column(Integer, primary_key=True) # (Mapeo de una columna)
11    nombre = Column(String)
12
13
14 engine = create_engine(
15     "postgresql+psycopg2://postgres:admin@localhost/LabIV_2023", echo=True)
16 Base.metadata.create_all(engine)
17
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2023-06-20 20:05:06,036 INFO sqlalchemy.engine.Engine select current_schema()
2023-06-20 20:05:06,036 INFO sqlalchemy.engine.Engine [raw sql] {}
2023-06-20 20:05:06,037 INFO sqlalchemy.engine.Engine show standard_conforming_strings
2023-06-20 20:05:06,037 INFO sqlalchemy.engine.Engine [raw sql] {}
2023-06-20 20:05:06,038 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2023-06-20 20:05:06,039 INFO sqlalchemy.engine.Engine select relname from pg_class c join pg_namespace n on n.oid
=c.relnamespace where pg_catalog.pg_table_is_visible(c.oid) and relname=%(name)s
2023-06-20 20:05:06,039 INFO sqlalchemy.engine.Engine [generated in 0.00017s] {'name': 'contactos'}
2023-06-20 20:05:06,041 INFO sqlalchemy.engine.Engine
CREATE TABLE contactos (
  id SERIAL NOT NULL,
  nombre VARCHAR,
  PRIMARY KEY (id)
)
2023-06-20 20:05:06,041 INFO sqlalchemy.engine.Engine [no key 0.00020s] {}
2023-06-20 20:05:06,054 INFO sqlalchemy.engine.Engine COMMIT
o (base) negrux@Rodrigos-MBP sql-alchemy-orm %
```

Ver que por defecto define la columna id como serial. En PGAdmin se crea la tabla “contactos” con las indicaciones de nuestra query.

En el caso que sigamos agregando columnas, relaciones o modificaciones a la tabla, cuando ejecutemos no va a modificar la tabla en la BBDD, dado que ya está creada.

Por ejemplo, si modificamos la declaración de la tabla de la siguiente manera:

```
# Hereda de la clase Base. Luego le decimos como debe mapear esa clase.
class Contacto(Base):
    __tablename__ = "contactos" # (Nombre de la tabla)
    id = Column(Integer, primary_key=True) # (Mapeo de una columna)
    # 60 indica el máx de caracteres que puede tener la variable
    nombre = Column(String(60), nullable=False)
    apellido = Column(String(60), nullable=False)
    # column_property es una función que devuelve una columna calculada.
    nombre_completo = column_property(nombre + " " + apellido)
    # DateTime es de SQLAlchemy y datetime de python, 1arg: Tipo, 2arg: Valor x defecto
    (Caso que venga sin valor)
    fecha_nacimiento = Column(DateTime, default=datetime.now())
```

Cuando ejecutemos, no nos va a avisar que la tabla no se modificó. Lo que tenemos que hacer es ir a PGAdmin, **borrar la tabla (DELETE/DROP) y volver a ejecutar** el código SQL Alchemy. Ahí nos volverá a crear la tabla pero modificada.

Resumen Laboratorio IV - BackEnd (2022)

```
CREATE TABLE contactos (  
    id SERIAL NOT NULL,  
    nombre VARCHAR(60) NOT NULL,  
    apellido VARCHAR(60) NOT NULL,  
    fecha_nacimiento TIMESTAMP WITHOUT TIME ZONE,  
    PRIMARY KEY (id)  
)
```

Vemos que el “nombre_completo” no se creó, esto es porque esa columna no se crea en el servidor (En este dialecto x lo menos) , directamente esta es una columna que va a quedar en la clase de python y la podemos usar en la clase de python, pero el que se va a encargar de darle valores es SQL ALCHEMY, no el servidor. Misma situación con el valor por default de la “fecha_nacimiento”.

Cuando trabajamos con el ORM, trabajamos con un objeto llamado **Session**, este objeto Session hay que importarlo y se crea pasándole como parámetro el **engine** (motor de ejecución de SQL Alchemy) y es el objeto que nos va a dar todos los métodos para acceder a la base de datos (Consultar, insertar, borrar, modificar, etc).

Esto nos permite por ejemplo crear un contacto de la forma:

```
from sqlalchemy.orm import declarative_base, column_property, Session
```

También podemos agregar un método init (constructor) para los contactos de igual forma que como lo hacemos en python:

```
# Creamos un inicializador con los valores que no pueden ser nulos.  
def __init__(self, id, nombre, apellido):  
    self.id = id  
    self.nombre = nombre  
    self.apellido = apellido
```

y utilizamos **Session** para crear un nuevo contacto (Prestar atención al **commit()**):

```
with Session(engine) as session:  
    c = Contacto(3, 'Rodrigo', 'Sosa')  
    c.fecha_nacimiento = datetime(1984, 7, 19)  
    # Lo agrega a la session, hasta que se haga el commit, y envia a la BBDD.  
    session.add(c)  
    # Realiza la transacción, sin esto no se agrega a la BBDD.  
    session.commit()
```

Ejecutamos y nos marca:

```
2023-06-21 17:19:03,171 INFO sqlalchemy.engine.Engine [generated in 0.00032s] {'id': 3, 'nombre':  
'Rodrigo', 'apellido': 'Sosa', 'fecha_nacimiento': datetime.datetime(1984, 7, 19, 0, 0)}  
2023-06-21 17:19:03,218 INFO sqlalchemy.engine.Engine COMMIT
```

Tener en cuenta que si quisiera ejecutarlo de nuevo, al nosotros colocar el id en el inicializador, el código nos dará un **error** de violación de clave única, ya que querría volver a cargar el mismo id existente.

Resumen Laboratorio IV - BackEnd (2022)

Enumeraciones en python: Son clases que descienden de la clase `"enum.Enum"`. Tienen la forma `identificador = "valor"`. Los enum son soportados en este ORM a través del tipo `sqlalchemy.Enum` (Tener cuidado al importar porque el de python se llama igual pero viene del módulo `enum` con minúscula 🧑)

```
# enumeracion de python
class TipoUsuario(enum.Enum):
    tipo1 = 'Tipo Uno'
    tipo2 = 'Tipo Dos'
```

```
fecha_ingreso = Column(DateTime, default=datetime.now())
tipo_usuario = Column(sqlalchemy.Enum(TipoUsuario)) ## Enum es de SQLAlchemy
```

Conviene importar los módulos completos y al momento de usarlo utilizar la librería por ejemplo `sqlalchemy.Enum` y lo mismo con el `enum.Enum` en la de python.

Sintaxis para realizar consultas (Seguimos usando el objeto Session):

Le pedimos a la **session** que ejecute (**execute**) un método que se importa de sqlalchemy (sqlalchemy común), ahora usamos el método **select** indicando que es lo que vamos a seleccionar como parámetro, en este caso "contactos". Luego lo imprimimos por consola.

```
with Session(engine) as session:
# Creo una lista con todos los contactos que encuentre en la columna contacto
# importo select de sqlalchemy
contactos = session.execute(select(Contacto))
print(contactos)
```

Lo que imprime por pantalla es: `<sqlalchemy.engine.result.ChunkedIteratorResult object at 0x7f84d4a0ba60>`

Modificamos el print agregando el método `all` para que nos muestre las tuplas:

```
print(contactos.all())
```

Pero sigue mostrando la dirección de memoria en forma de tuplas, por esto hay que **sobreescribir el método `toString()`** a esta clase. Para que nos de valores legibles por las personas.

```
# Redefinir método ToString()
def __repr__(self): #También puede ser __str__
return f"Contacto ({self.id}: {self.apellido}, {self.nombre})"
```

```
2023-06-21 18:07:21,892 INFO sqlalchemy.engine.Engine SELECT contactos.nombre || %(param_1)s || contactos.apellido AS anon_1, contactos.i
d, contactos.nombre, contactos.apellido, contactos.fecha_nacimiento
FROM contactos
2023-06-21 18:07:21,892 INFO sqlalchemy.engine.Engine [generated in 0.00015s] {'param_1': ' '}
[(Contacto (3: Sosa, Rodrigo)), (Contacto (5: Gonzalez, Juan)), (Contacto (4: Perez, Pepe
)),]
```

Podemos utilizar la función `where` (también se importa de sqlalchemy junto con el `select`)

```
with Session(engine) as session:
# Creo una lista con todos los contactos que encuentre en la columna contacto
# importo select de sqlalchemy
contactos = session.execute(select(Contacto).where(Contacto.id == 4))
print(contactos.all())
```

Resumen Laboratorio IV - BackEnd (2022)

Otros comandos usando **TEXT**: https://docs.sqlalchemy.org/en/14/tutorial/dbapi_transactions.html#fetching-rows

Usando el **ORM**: https://docs.sqlalchemy.org/en/14/tutorial/orm_data_manipulation.html

Ejemplo de update usando el ORM:

ORM-enabled UPDATE statements

As previously mentioned, there's a second way to emit UPDATE statements in terms of the ORM, which is known as an **ORM enabled UPDATE statement**. This allows the use of a generic SQL UPDATE statement that can affect many rows at once. For example to emit an UPDATE that will change the `User.fullname` column based on a value in the `User.name` column:

```
>>> session.execute(
...     update(User)
...     .where(User.name == "sandy")
...     .values(fullname="Sandy Squirrel Extraordinaire")
... )
```

```
UPDATE user_account SET fullname=? WHERE user_account.name = ?
[...] ('Sandy Squirrel Extraordinaire', 'sandy')
```

```
<sqlalchemy.engine.cursor.CursorResult object ...>
```

When invoking the ORM-enabled UPDATE statement, special logic is used to locate objects in the current session that match the given criteria, so that they are refreshed with the new data. Above, the `sandy` object identity was located in memory and refreshed:

```
>>> sandy.fullname
'Sandy Squirrel Extraordinaire'
```

Ejemplo. Crear con el ORM una clase país (Tabla: `paises`) y una clase provincia (Tabla: `provincias`) con las siguientes características:

```
create table paises(id serial not null primary key, nombre varchar(80) not null);

create table provincias(id serial not null primary key, nombre varchar(120), pais_id int references paises(id));
```

```
from sqlalchemy.orm import declarative_base
import sqlalchemy

Base = declarative_base()

class Pais(Base):
    __tablename__ = 'paises' # Atributo que indica a que tabla hace referencia
    id = sqlalchemy.Column(sqlalchemy.Integer, primary_key=True)
    nombre = sqlalchemy.Column(sqlalchemy.String(80), nullable=False)

    def __repr__(self):
        return f"Pais (id:{self.id}, nombre: {self.nombre})"

class Provincia(Base):
    __tablename__ = 'provincias' # Atributo que indica a que tabla hace referencia
    id = sqlalchemy.Column(sqlalchemy.Integer, primary_key=True)
    nombre = sqlalchemy.Column(sqlalchemy.String(120), nullable=False)
```

Resumen Laboratorio IV - BackEnd (2022)

```
# Acá hay una relación entre pais_id e id de provincia.
# Se utiliza la función FOREIGNKEY de sqlalchemy la que espera la columna con la que se
# referencia (estructura de la BBDD), usamos nombreTabla.nombreCampo y lo pasamos como
STRING
pais_id = sqlalchemy.Column(
    sqlalchemy.Integer, sqlalchemy.ForeignKey('países.id'))

# Creamos las tablas:
engine = sqlalchemy.create_engine(
    "postgresql+psycopg2://postgres:admin@localhost/LabIV_2023", echo=True)
Base.metadata.create_all(engine)
```

```
2023-06-21 18:53:34,288 INFO sqlalchemy.engine.Engine [no key 0.00013s] {}
2023-06-21 18:53:34,420 INFO sqlalchemy.engine.Engine
CREATE TABLE provincias (
    id SERIAL NOT NULL,
    pais_id INTEGER,
    PRIMARY KEY (id),
    FOREIGN KEY(pais_id) REFERENCES países (id)
)
```

Creamos instancias:

```
with Session(engine) as s:
    # Aunque no tenga inicializador puedo crear objetos de esta forma:
    argentina = Pais(id=1, nombre='Argentina')
    s.add(argentina)
    s.commit()

with Session(engine) as s:
    er = Provincia(id=1, nombre='Entre Ríos', pais_id=1)
    s.add(er)
    s.commit()
```

Si quisiéramos agregar una provincia con una clave foránea de un país que no existe nos marca un error:

```
with Session(engine) as s:
    # argentina = Pais(id=1, nombre='Argentina')
    # s.add(argentina)

    s.add(Provincia(id=2, nombre='Santa Fe', pais_id=2))

    s.commit()
```

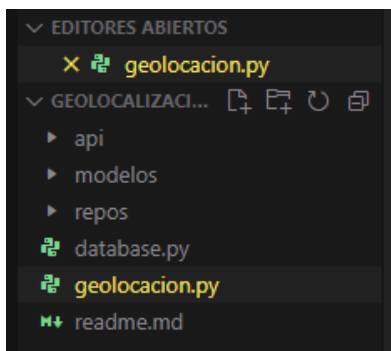

Resumen Laboratorio IV - BackEnd (2022)

```
cursor.execute(statement, parameters)
sqlalchemy.exc.IntegrityError: (psycopg2.errors.ForeignKeyViolation) inserción o actualización en la ta
bla «provincias» viola la llave foránea «provincias_pais_id_fkey»
DETAIL: La llave (pais_id)=(2) no está presente en la tabla «paises»
```

Se puede tener valores nulos de FK, aunque no es lo ideal, el ORM lo permite. Lo que no permite es asignar valores que no existen. Es decir, **una Foreign Key puede ser nula, no puede tener un valor que no exista en otra tabla**. También le podemos pedir que no permita nulos, con **nullable = False**. Son dos restricciones distintas.

Integración de FastAPI con SQL Alchemy (Clase 5)

Lo primero es organizar el código, por un lado, la **API**, por otro lado, los **modelos** y otra carpeta para los **repositorios**.



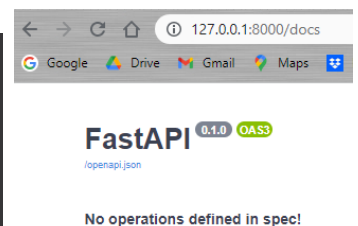
Luego creamos el **servidor FastAPI** en el archivo **geolocacion.py**, lo ubicamos en la raíz del proyecto. Este servidor nos permitirá ingresar a la página swagger con la ruta **servidor/docs**. Hacemos clic en “run” para iniciar:

```
#Creamos el servidor FastAPI
import uvicorn #Importo uvicorn
from fastapi import FastAPI #Importo FastAPI

app = FastAPI() #Creo una instancia de FastAPI

if __name__ == '__main__': #inicializo FastAPI
    uvicorn.run('geolocacion:app', host='127.0.0.1', port=8000, reload=True)
```

```
(base) PS D:\Documentos\UTN - Laboratorio IV 2022\5_Backend\Backend-5_Clase_091122\geolocalizacion_fastapi_sqlalchemy> & C:/
Users/Rodrigo/AppData/Local/Microsoft/WindowsApps/python3.9.exe "d:/Documentos/UTN - Laboratorio IV 2022/5_Backend/Backend-5
_Clase_091122/geolocalizacion_fastapi_sqlalchemy/geolocacion.py"
INFO: Will watch for changes in these directories: ['D:\\Documentos\\UTN - Laboratorio IV 2022\\5_Backend\\Backend-5_Cla
se_091122\\geolocalizacion_fastapi_sqlalchemy']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [3444] using watchgod
INFO: Started server process [6708]
INFO: Waiting for application startup.
INFO: Application startup complete.
```



Los **endpoints** necesarios serán:

```
# Geolocalización
## Países
### Endpoints
* GET /países: devuelve todos los países
* GET /países/{id}: devuelve detalle del país {id}
* POST /países: agrega un país
* PUT /países/{id}: modifica un país
* DELETE /países/{id}: borra un país
```


Resumen Laboratorio IV - BackEnd (2022)

Dentro de la carpeta “api” vamos a guardar los endpoints. Dentro de la misma creamos un archivo **países_api.py**

```
#países_api.py

from fastapi import APIRouter #Importamos el router

países_api = APIRouter(prefix='/países') #Creamos una instancia de APIRouter

@países_api.get('/') #' es lo mismo que '/'
def get_all():
    result = países_repo.get_all() #En repositorio es donde estarán las listas
    de países (Lista, bd, o lo que sea que los contenga)
    return result
```

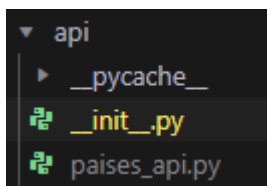
En el archivo principal incluimos el **APIRouter** e importamos el archivo **países_api**:

```
import uvicorn
from fastapi import FastAPI #Importo FastAPI
from api.países_api import países_api #Importamos países api

app = FastAPI() #Creo una instancia de FastAPI
app.include_router(países_api) #Pedimos que incluya las rutas (endpoints)
definidas en el archivo "países_api"

if __name__ == '__main__': #inicializo FastAPI
    uvicorn.run('geolocalizacion:app', host='127.0.0.1', port=8000, reload=True)
```

Nota: En la carpeta “api” donde guardamos **países_api** se crea un archivo **__init__.py**. **Este archivo le indica a python que ese directorio es un paquete.** Esto **nos permite importar** los paquetes. Aunque no es necesario en versiones nuevas de python, lo agregamos igual por si utilizamos alguna versión vieja. Es un archivo con ese nombre, donde se colocan configuraciones iniciales, por ejemplo, crear clases necesarias. Nosotros solo lo vamos a crear en cada una de las carpetas que vamos a importar como paquetes, pero lo dejaremos vacío. (Para evitar errores con versiones viejas)



En el directorio repos, creamos un archivo **países_repositorio.py** que es donde vamos a *colocar todos los datos*, entonces creamos una clase **países_repositorio**, la misma contiene el método **get_all**. Para probar podemos hacer una lista de prueba.

```
class PaísesRepositorio():
    #Contiene el metodo get_all que llamamos desde la API
    def get_all(self): #No olvidar el self porque es met. de clase!!!
        return [País(id=1, nombre="Argentina"), #Modelo país
                País(id=2, nombre="Uruguay")]
],
```

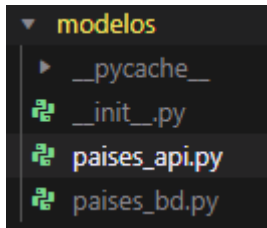
Resumen Laboratorio IV - BackEnd (2022)

Luego la importamos al archivo **países_api.py** y creamos una instancia del mismo:

```
from repos.países_repositorio import PaísesRepositorio #Importamos el repositorio

países_repo = PaísesRepositorio() #Creamos una instancia del repositorio
```

Creamos el **modelo** de países dentro de la carpeta “modelos” como **países_api.py**, necesitamos **dos modelos** el que contiene los modelos que nos pide el api (**Response model de FastAPI**) y otro modelo que nos pide la **base de datos** (**SQLAlchemy**).



El archivo **países_api.py** queda:

```
#Este es el modelo de FastAPI (pydantic)
from pydantic import BaseModel

class Pais(BaseModel):
    id: int
    nombre: str
```

Lo **importamos** en el **repositorio**:

```
from modelos.países_api import Pais
```

Seguimos, probamos el **swagger**: <http://127.0.0.1:8000/docs/> nos debería mostrar el primer get.

Ahora vamos a crear una **tabla de base de datos**, a nivel del proyecto raíz creamos un nuevo archivo “database.py” que **contiene los objetos y las funciones necesarias para conectar con la base de datos**. Esto es bastante estándar, tenemos las variables de conexión, las llamadas a **SQLAlchemy**, el **create_engine** con la **cadena de conexión** y le pedimos que nos muestre todos los comandos de SQL que crea en la consola (**echo=True**).

```
#database.py: (A nivel del main geolocalacion.py)

from sqlalchemy.orm import sessionmaker, declarative_base
from sqlalchemy import create_engine #Importamos la funcion p/crear el engine

SQLALCHEMY_DATABASE_URL =
"postgresql+psycopg2://postgres:admin123@localhost:5432/labIV_2023" #Cadena de conexión

engine = create_engine(SQLALCHEMY_DATABASE_URL, echo=True) #Motor de conexión
```

```
SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)
```

#sessionmaker es una función que cuando la ejecuto me devuelve una sesión de conexión a la Base de Datos.

#bind: Indica a qué motor de conexión se va a conectar, en este caso engine,

#autocommit: significa que las instrucciones que requieran commit se hagan de forma manual o automática, se sugiere False y colocar un bloque try-catch (En el try ponemos todas las operaciones que queremos que se tomen en una sola transacción, se terminan las operaciones y hacemos el commit. Si hay un error sale por el catch, hay que capturar ese error y hacer un ROLLBACK, para que la BBDD quede como estaba antes de esa transacción. Se sugiere tenerlo en False.

#Autoflush: Cada tanto, el engine libera las operaciones que tenemos a la BBDD, junta operaciones y cada tanto las carga todas juntas. En False, nosotros terminamos la transacción con commit o rollback, y luego se termina la conexión.

```
BaseBd = declarative_base()
```

#Es la clase base de donde descienden todas las clases que quiero meter en la BBDD con el ORM.

#Para obtener la sesión, usamos inyección de dependencias, es una función que devuelve una variable yield

```
def get_db():
    db = SessionLocal() #Se crea la conexión a la BBDD - Inyección de dependencias
    try:
        yield db #Devuelve la variable db como si fuera un return, pero espera que se termine de usar la variable, luego continúa con las instrucciones, y después con el finally que cierra la conexión.
    finally:
        db.close() #Cierra la conexión
```

```
def create_all(): #Pasa todas las entidades que tenga definida para crear la BBDD
    BaseBd.metadata.create_all(bind=engine)
```

```
def drop_all(): #Borra todas las tablas referidas a las entidades definidas en la BBDD
    BaseBd.metadata.drop_all(bind=engine)
```

Luego lo incluimos en el archivo principal (geolocacion.py) y lo importamos.

```
import uvicorn
from fastapi import FastAPI #Importo FastAPI
from api.paises_api import paises_api #Importamos paises_api
import database #Importamos el archivo de conexión a la BBDD
import modelos.paises_bd #Importamos los modelos de BBDD
```

Resumen Laboratorio IV - BackEnd (2022)

`database.create_all()` #Crea los modelos de la BBDD, si ya existen no los vuelve a crear. (Hay que borrarlos y volver a ejecutar el código)

```
app = FastAPI() #Creo una instancia de FastAPI
app.include_router(paises_api) #Pedimos que incluya las rutas (endpoints)
definidas en el archivo "paises_api"
```

```
if __name__ == '__main__': #inicializo FastAPI
    uvicorn.run('geolocacion:app', host='127.0.0.1', port=8000, reload=True)
```

<https://docs.sqlalchemy.org/en/14/core/metadata.html>

El archivo **países_bd.py** que sirve como modelo para la creación de la BBDD:

```
#países_bd.py
from database import BaseBd
from sqlalchemy import Column, Integer, String

#Crea los modelos de tablas en la BBDD
class PaisBd(BaseBd): #Definimos los atributos de la tabla (columnas) y nombre
de la tabla. Esta clase descende de BaseBd que está en database.py
    __tablename__ = 'paises'
    id = Column(Integer, primary_key=True)
    nombre = Column(String(80), nullable=False)
```

Y en el repositorio ahora cambiamos para que haga una consulta a la BBDD, necesitamos obtener la sesión,

```
#países_repository
from modelos.paises_bd import PaisBd
from modelos.paises_api import PaisSinId
from sqlalchemy.orm import Session #Necesitamos la sesión de la bd para poder
conectarnos
from sqlalchemy import select

class PaísesRepository():

    #Devuelve las rows de la BBDD transformadas en objetos países en una lista
de python
    def get_all(self, db:Session): #Importamos Session y Select
        return
db.execute(select(PaisBd).order_by(PaisBd.nombre)).scalars().all()
#Crea la consulta SQL por medio de SQLAlchemy
#Para sacar las tuplas de adentro de las rows usamos "scalars()"
#Con .all() me devuelve todo en una lista de objetos PaisBd.
```

Ahora necesito convertir la lista de paísesBd en una lista de **paísesAPI**, que es la clase que quiero usar para la API.

Eso lo hago desde el **response_model** en países_api.py.

```
from modelos.paises_api import PaisApi
```

Resumen Laboratorio IV - BackEnd (2022)

```
@países_api.get('', response_model=list[PaisApi]) # '' es lo mismo que '/países'
def get_all(db: Session = Depends(get_db)): #Depends(): Crea la instancia usando la función get_db y
cuando termina termina llama al db.close()
    result = países_repo.get_all(db)
    return result
```

El modelo de países_api que utiliza FastAPI no es el mismo que el de la BBDD, esto genera un error, para solucionarlo se hace lo siguiente:

Se crea una **clase interna "Config"**, la cual tiene una constante llamada **orm_mode** que tiene que estar en **True**. Esta clase se hereda, por lo que hay que agregarla en la clase base desde donde descienden las otras clases que interactúan también con la BBDD. Esto se usa solo en las clases que se DEVUELVEN de la API, es decir las que se van a serializar a JSON, y que tienen que hacer el mapeo de las clases de BBDD a esta clase de API para serializarla justamente. Si vamos a usar estas clases para ENVIAR datos a la API esto no es necesario (El config), solo se necesita para RECIBIR datos de la API (Para la conversión)

```
# Este es el modelo de FastAPI (pydantic)
from pydantic import BaseModel

class PaisSinId(BaseModel): #Sirve para crear un país (el id se genera automático)
    nombre: str

    class Config:
        orm_mode = True

class PaisApi(PaisSinId):
    id: int

#Como en este caso PaisApi hereda de PaisSinId que ya tiene la clase interna Config, no es
necesario volver a colocarla acá.
#Pero en caso de que no herede se la agrega dentro de la clase para que no de error.
```

Si no tuviéramos esta configuración, **podríamos hacer la conversión** de países_bd a países_api de forma manual, pero para hacerlo sería mucho trabajo. Lo que hace config es algo así:

```
@países_api.get('', response_model=list[PaisApi])
def get_all(db = Depends(get_db)):
    result = países_repo.get_all(db)
    return [PaisApi(id=x.id, nombre=x.nombre) for x in result]
```

Método POST:

```
# api.países_api.py
# Devuelve el obj nuevo creado y un cod 201
@países_api.post('', response_model=PaisApi, status_code=201)
# Resive el JSON con los datos del nuevo país (Sin ID) y la session.
def nuevo(datos: PaisSinId, db: Session = Depends(get_db)):
# Utiliza la función agregar del repositorio, pasandole la session y los datos que tiene que
agregar
    result = países_repo.agregar(db, datos)
    #Nos devuelve el nuevo paísBD que devuelve el repo y cuando hace el return lo cambia a
paisAPI por el response.
    return result
```

Resumen Laboratorio IV - BackEnd (2022)

En el repositorio generamos la función agregar():

```
#países_repositorio.py
#Tengo que crear una instancia de paisBd, para poder usar los metodos de SQLAlchemy, como
session, add, etc

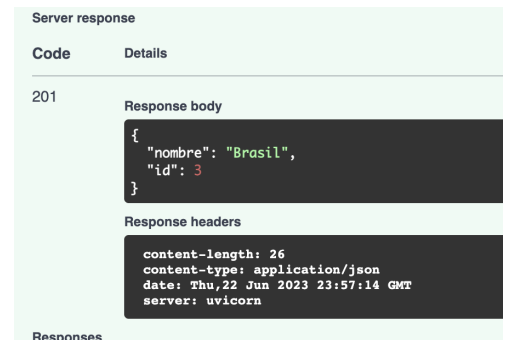
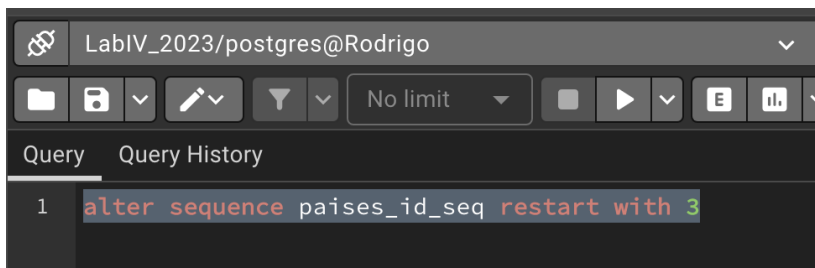
def agregar(self, db: Session, datos: PaisSinId):
#datos(paisSinId) no me sirve para pasar a la BD, entonces lo transformamos en una nueva entidad
BD
    nueva_entidad_bd: PaisBd = PaisBd(nombre=datos.nombre) #Transformo
    db.add(nueva_entidad_bd) #Agrego a la BD
    db.commit() #Comiteo y devuelvo.
    return nueva_entidad_bd
```

Ahora si ejecutamos este POST, nos va a dar error y nos va a decir que ya existe el id 1, porque en la secuencia de creación del ID de la Base de datos arranca desde el 1, y nosotros agregamos de forma manual dos valores con id 1 e id 2.

```
sqlalchemy.exc.IntegrityError: (psycopg2.errors.UniqueViolation) duplicate key value violates unique constraint "
pais_pkey"
DETAIL: Key (id)=(1) already exists.

[SQL: INSERT INTO pais (nombre) VALUES (%(nombre)s) RETURNING pais.id]
[parameters: {'nombre': 'Brasil'}]
(Background on this error at: https://sqlalche.me/e/14/gkpj)
```

Entonces tenemos que alterar el contador desde PGAdmin: **alter sequence pais_id_seq restart with 3**



Asigna el valor 3 al nuevo país creado. La otra forma es borrar los valores creados y arrancar de nuevo (No es lo ideal)

METODO GET {id}

Tener en cuenta que el código de acceso a la BBDD va en el repositorio, entonces ahí es donde ponemos el método de acceso a la base de datos. Creamos un método get_by_id(id, db) al que le pasamos por parámetro el id a buscar y la session de la base de datos.

países_repositorio.py:

```
def get_by_id(self, db: Session, id: int):
# Devuelve una lista de objetos ROW, entonces lo convertimos a obj paisBd y devolver con
scalar() (en singular)
    result = db.execute(select(PaisBd).where(PaisBd.id == id)).scalar()
    return result
```

países_api.py:

```
@países_api.get('/{id}', response_model=PaisApi)
def get_by_id(id: int, db: Session = Depends(get_db)):
    result = pais_repo.get_by_id(db, id)
#Si no lo encuentra marca un error y un status code
    if result is None:
        raise HTTPException(status_code=404, detail='Pais no encontrado')
```

```
return result
```

METODO PUT {id}

Este método pide un id del país a modificar, y envía como payload en el cuerpo de la request un JSON con los datos modificados de ese país. Le pedimos que devuelva la instancia ya modificada.

países_api.py:

```
@países_api.put('/{id}', response_model=PaisApi)
def modificar(id: int, datos: PaisSinId, db: Session = Depends(get_db)):
    #Creamos la función modificar en el repo.
    result = países_repo.modificar(db, id, datos)
    #Opción si NO lo encontró
    if result is None:
        raise HTTPException(status_code=404, detail='País no encontrado')

    #Opción si lo encontró
    return result
```

países_repositorio.py:

```
def modificar(self, db: Session, id: int, datos: PaisSinId):
    entidad: PaisBd = self.get_by_id(db, id) #Uso el método de clase get_by_id()
    #Si NO lo encuentra:
    if entidad is None:
        return None
    #Si lo encuentra, actualiza los datos y hace el commit, devuelve entidad.
    entidad.nombre = datos.nombre
    db.commit()
    return entidad
```

METODO DELETE {id}

países_api.py:

```
# 204: Significa que no está devolviendo nada pero que está todo OK.
@países_api.delete('/{id}', status_code=204)
def borrar(id: int, db: Session = Depends(get_db)):
    result = países_repo.borrar(db, id)
    if result is None:
        raise HTTPException(status_code=404, detail='País no encontrado')
    return #No devuelve nada.
```

países_repositorio.py:

```
def borrar(self, db: Session, id: int):
    entidad: PaisBd = self.get_by_id(db, id)
    if entidad is None:
        return None
    db.delete(entidad)
    db.commit()
    return entidad
```

Tablas referenciadas:

```
class ProvinciasBd(BaseBd):
    __tablename__ = 'provincias'

    id = Column(Integer, primary_key=True)
    nombre = Column(String(80), nullable=False)
    pais_id = Column(Integer, ForeignKey('países.id'))
```

AGREGANDO LAS TABLAS DE PROVINCIAS AL PROYECTO:

modelos.provincias_bd.py: (Modelo para la Base de Datos)

Resumen Laboratorio IV - BackEnd (2022)

```
from sqlalchemy.orm import relationship # Sirve para traer objetos
from sqlalchemy import Column, ForeignKey, Integer, String
from database import BaseBd

class Provincias(BaseBd):
    __tablename__ = 'provincias'

    id = Column(Integer, primary_key=True)
    nombre = Column(String(80), nullable=False)
    # Colocamos el campo que hace la relación como FK:
    ForeignKey('nombre_tabla_en_bd'.nombre_columna_en_bd')
    # Teoría: Tener en cuenta, que el id no es necesario que sea un PK, también puede ser un
    "unique"
    pais_id = Column(Integer, ForeignKey('países.id'))

    # Atributo de Navegación: Es un objeto de tipo PaisBd, es una instancia TEMPORAL
    completa de la clase PaisBD, se usa para que SQL Alchemy me traiga el objeto completo de
    "país", no solo el ID, esto sirve para agilizar y no tener que estar haciendo muchos select,
    join, etc. Lo puedo llamar y obtener sus atributos.
    # *LAZY LOADING: Recién cuando se acceda, por ejemplo haciendo "p.pais.nombre", el ORM
    automáticamente hace un select a la tabla de países y me lo trae de forma completa. Por ejemplo
    si muestro una lista de provincias, solo muestra el id de países, ahora si en esa lista
    necesito mostrar el nombre del país, accedo a esta relationship, el orm hace el select a tabla
    países y accedo a su nombre.
    pais = relationship('PaisBd')

    # TEORÍA: La relación entre países y provincias es 1 a muchos (1:n)
    # Si por ej. fuera 1:1 = La relación entre país y provincia, sería lo mismo poner el FK
    en provincias o en países
    # En este caso si colocamos el FK en países, porque restringimos a cada país a una sola
    provincia, entonces: EL ID con el FK SIEMPRE se pone del lado del MUCHOS (Provincias en este
    caso)

    # Caso 1:1 = Paciente -> Historia Clínica
    # Caso n:n = Cursos:Alumnos. Se hace una tabla intermedia que tiene las dos tablas
    relacionadas. Se plantea una entidad, por ejemplo inscripción y ponemos los dos id, cada id va
    a tener una FK a la tabla que correspondan y además ambos id van a ser la PK de la tabla
    inscripción. (Ver apunte SQL Alchemy como el orm arma automáticamente la tabla)
```

modelos.provincias_api.py: (Modelo para FastAPI)

```
from pydantic import BaseModel
from modelos.países_api import PaisApi # Lo necesito para

# Modelo para AGREGAR provincias
class ProvinciaSinId(BaseModel):
    nombre: str
    pais_id: int

    class Config:
        orm_mode = True

# Modelo para mostrar lista de provincias, trae Provincia(nombre, id_pais, objetoPais)
class ProvinciaList(ProvinciaSinId):
    pais: PaisApi

# Modelo para mostrar el detalle de UNA Provincia (id, nombre, id_pais)
class ProvinciaApi(ProvinciaSinId):
    id: int
```


Resumen Laboratorio IV - BackEnd (2022)

repos.provincias_repo.py: (metodos para los endpoint)

```
import sqlalchemy

from modelos.paises_bd import PaisBd
from modelos.provincias_bd import ProvinciaBd
from modelos.provincias_api import ProvinciaSinId
from sqlalchemy.orm import Session
from sqlalchemy import select

class ProvinciasRepositorio():

    def get_all(self, db: Session):
        result = db.execute(select(ProvinciaBd, PaisBd).join(
            PaisBd).order_by(ProvinciaBd.nombre)).scalars().all()
        return result

    def get_by_id(self, db: Session, id: int):
        result = db.execute(select(ProvinciaBd).where(
            ProvinciaBd.id == id)).scalar()
        return result

    def agregar(self, db: Session, datos: ProvinciaSinId):
        nueva_entidad_bd: ProvinciaBd = ProvinciaBd(**datos.dict())
        try:
            db.add(nueva_entidad_bd)
            db.commit()
        except sqlalchemy.exc.IntegrityError as e:
            raise RuntimeError(f'Error al agregar una provincia: {e}')
        return nueva_entidad_bd

    def modificar(self, db: Session, id: int, datos: ProvinciaSinId):
        entidad: ProvinciaBd = self.get_by_id(db, id)
        if entidad is None:
            return None
        for k, v in datos.dict(exclude_unset=True).items():
            setattr(entidad, k, v)
            db.commit() (????????????)
        return entidad

    def borrar(self, db: Session, id: int):
        entidad: ProvinciaBd = self.get_by_id(db, id)
        if entidad is None:
            return None
        db.delete(entidad)
        db.commit()
        return entidad
```

api.provincias_api.py: (Endpoint de provincias)

```
from fastapi import APIRouter, Depends, HTTPException
from database import get_db
from sqlalchemy.orm import Session
from modelos.provincias_api import ProvinciaApi, ProvinciaSinId, ProvinciaList
from repos.provincias_repo import ProvinciasRepositorio

provincias_api = APIRouter(prefix='/provincias', tags=['Provincias'])
provincias_repo = ProvinciasRepositorio()

@provincias_api.get('', response_model=list[ProvinciaList])
```

Resumen Laboratorio IV - BackEnd (2022)

```
def get_all(db: Session = Depends(get_db)):
    result = provincias_repo.get_all(db)
    return result

@provincias_api.get('/{id}', response_model=ProvinciaApi)
def get_by_id(id: int, db: Session = Depends(get_db)):
    result = provincias_repo.get_by_id(db, id)
    if result is None:
        raise HTTPException(status_code=404, detail='Provincia no encontrada')
    return result

@provincias_api.post('', response_model=ProvinciaApi, status_code=201)
def new(datos: ProvinciaSinId, db: Session = Depends(get_db)):
    try:
        result = provincias_repo.agregar(db, datos)
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))
    return result

@provincias_api.put('/{id}', response_model=ProvinciaApi)
def modify(id: int, datos: ProvinciaSinId, db: Session = Depends(get_db)):
    result = provincias_repo.modificar(db, id, datos)
    if result is None:
        raise HTTPException(status_code=404, detail='Provincia no encontrada')
    return result

@provincias_api.delete('/{id}', status_code=204)
def borrar(id: int, db: Session = Depends(get_db)):
    result = provincias_repo.borrar(db, id)
    if result is None:
        raise HTTPException(status_code=404, detail='Provincia no encontrada')
    return
```

Hay que agregar los include router de provincias en el main:

```
geolocacion.py x provincias_api.py

backend > geolocacion.py > ...
1  import uvicorn # Importo uvicorn
2  from fastapi import FastAPI # Importo FastAPI
3  from api.países_api import países_api # Importamos países api
4  from api.provincias_api import provincias_api
5  import database # Importamos el archivo de conexión a la BBDD
6  import modelos.países_bd # Importamos los modelos de BBDD
7  import modelos.provincias_bd
8
9  # Crea los modelos de la BBDD, si ya existen no los vuelve a crear.
10 database.create_all()
11
12 app = FastAPI() # Creo una instancia de FastAPI
13 # Pedimos que incluya las rutas (endpoints) definidas en el archivo "países_api"
14 app.include_router(países_api)
15 # NO OLVIDARSE DE IMPORTAR EL ROUTER O NO SE CARGAN LOS ENDPOINTS!!!
16 app.include_router(provincias_api)
17
18 if __name__ == '__main__': # inicializo FastAPI
19     uvicorn.run('geolocacion:app', host='127.0.0.1', port=8000, reload=True)
20
```

Resumen Laboratorio IV - BackEnd (2022)

LOCALIDADES:

Conviene arrancar por el modelo de base de datos:

modelos.localidades_bd.py:

```
from sqlalchemy.orm import relationship # Sirve para traer objetos
from sqlalchemy import Column, ForeignKey, Integer, String
from database import BaseBd

class LocalidadBd(BaseBd):
    __tablename__ = 'localidades'

    id = Column(Integer, primary_key=True)
    nombre = Column(String(80), nullable=False)
    caracteristica_telefonica = Column(String(30), nullable=True)
    provincia_id = Column(Integer, ForeignKey(
        'provincias.id')) # 'nombre_tabla.id'

    provincia = relationship('ProvinciaBd')
```

Para verificar paso a paso, la importamos en el main y ejecutamos, el **database.create_all()** creará la tabla.

Continuamos con la api de localidades:

api.localidades_api.py:

```
from fastapi import APIRouter, Depends, HTTPException
from modelos.localidades_api import LocalidadSinId
from modelos.localidades_api import LocalidadDetalle
from modelos.localidades_api import LocalidadLista
from database import get_db
from repos.localidades_repo import LocalidadesRepositorio

# Esta capa que es la de acceso a endpoints no debería tener NADA DE SQL Alchemy
# ya que esa librería solo tiene que estar en la de acceso a las BBDD, esta capa
# no debería tener ninguna relación con el ORM, por si en el futuro cambio de librería.
# from sqlalchemy.orm import Session

localidades_api = APIRouter(prefix='/localidades', tags=['Localidades'])
repo = LocalidadesRepositorio()

# Pido que devuelva una lista!
@localidades_api.get('', response_model=list[LocalidadLista])
def get_all(db=Depends(get_db)): # Inyección de dependencias
    result = repo.get_all(db)
    return repo.get_all(db)

@localidades_api.get('/{id}', response_model=LocalidadDetalle)
def get_by_id(id: int, db=Depends(get_db)):
    result = repo.get_by_id(db, id)
    if result is None:
        raise HTTPException(status_code=404, detail='Localidad no encontrada')
    return result
```

Resumen Laboratorio IV - BackEnd (2022)

```
@localidades_api.post('', response_model=LocalidadDetalle, status_code=201)
def new(datos: LocalidadSinId, db=Depends(get_db)):
    try:
        result = repo.agregar(db, datos)
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))
    return result

@localidades_api.put('/{id}', response_model=LocalidadDetalle)
def modify(id: int, datos: LocalidadSinId, db=Depends(get_db)):
    result = repo.modificar(db, id, datos)
    if result is None:
        raise HTTPException(status_code=404, detail='Localidad no encontrada')
    return result

@localidades_api.delete('/{id}', status_code=204)
def borrar(id: int, db=Depends(get_db)):
    result = repo.borrar(db, id)
    if result is None:
        raise HTTPException(status_code=404, detail='Localidad no encontrada')
    return
```

A la par vamos realizando el repositorio, ya que contiene los métodos para que funcionen los endpoints.

repos.localidades_repo.py:

```
from sqlalchemy import select
from sqlalchemy.exc import IntegrityError
from modelos.localidades_api import LocalidadSinId
from modelos.provincias_bd import ProvinciaBd
from modelos.localidades_bd import LocalidadBd
from sqlalchemy.orm import Session

class LocalidadesRepositorio():

    def get_all(self, db: Session):
        # return [
        #     LocalidadBd(id=1, nombre='Localidad1'), #Hardcode para probar
        #     LocalidadBd(id=2, nombre='Localidad2'),
        # ]

        # INNER JOIN: return db.execute(select(LocalidadBd).join(ProvinciaBd)).scalars().all()

        # LEFT OUTER JOIN: Le pedimos que haga un JOIN entre localidad y provincia (Traeme todos los
        # datos de (TablaPrincipal)localidades y (TablaSecundaria)provincias) y
        # hacé un left outer join con provincias. Podría hacer otro outer join con paises para que lo
        # muestre. (El modelo de response (api) tiene que contener esos datos que pido)
        # https://stackoverflow.com/questions/39619353/how-to-perform-a-left-join-in-sqlalchemy
        return db.execute(select(LocalidadBd, ProvinciaBd).join(ProvinciaBd,
            isouter=True)).scalars().all()

    def get_by_id(self, db: Session, id: int):
        result = db.execute(select(LocalidadBd).where(
            LocalidadBd.id == id)).scalar()
        return result

    def agregar(self, db: Session, datos: LocalidadSinId):
        # pydantic ofrece el método .dict() que devuelve todos los atributos de esa instancia como un
        # diccionario
```

Resumen Laboratorio IV - BackEnd (2022)

```
# en formato clave=valor, y el ** significa que pasa x todos los elementos del diccionario uno
por uno
# y se lo asigna a los valores de esta instancia.
# LocalidadBd(**datos.dict())) es equivalente a:
# LocalidadBd(nombre=datos.nombre, caracteristica_telefonica =
datos.caracteristica_telefonica, provincia_id = datos.provincia_id)
nueva_entidad_bd: LocalidadBd = LocalidadBd(**datos.dict())
try:
db.add(nueva_entidad_bd)
db.commit()
# El error de integridad es cuando el id de provincia que agrego no existe en la tabla.
except IntegrityError.exc.IntegrityError as e:
raise RuntimeError(f'Error al agregar una provincia: {e}')
return nueva_entidad_bd

def modificar(self, db: Session, id: int, datos: LocalidadSinId):
# Orimero busco los datos que ya tengo en la BBDD y quiero modificar
entidad: LocalidadBd = self.get_by_id(db, id)
if entidad is None:
return None
# Me fijo todos los datos que traigo para cambiar, paso x todos los atributos de esa instancia
# por cada uno de esos valores tengo que asignarselo a la entidad que tengo en la bbdd,

# entonces el bucle for siguiente es equivalente a estas tres lineas:
# entidad.nombre = datos.nombre
# entidad.caracteristica_telefonica = datos.caracteristica_telefonica
# entidad.provincia_id = datos.provincia_id

for k, v in datos.dict(exclude_unset=True).items():
# exclude_unset = excluye los datos que no se pasan, paso los atributos que fueron asignados
solamente
# setattr = Le dice que tome esta instancia de la clase y le asigne al atributo k, el valor que
está en la variable v. (Que son los items del diccionario)
setattr(entidad, k, v)

db.commit()
return entidad

def borrar(self, db: Session, id: int):
entidad: LocalidadBd = self.get_by_id(db, id)
if entidad is None:
return None
db.delete(entidad)
db.commit()
return entidad
```

Generamos el modelo para la api:

modelos.localidades.api.py:

```
from pydantic import BaseModel
from modelos.provincias_api import ProvinciaApi

# Modelo para AGREGAR localidades

class LocalidadSinId(BaseModel):
    nombre: str
    caracteristica_telefonica: int
    # Con = None se vuelve opcional, esto es para que no de error cuando trae la lista de localidades
    provincia_id: int = None

    class Config:
        orm_mode = True

class LocalidadLista(BaseModel):
    nombre: str
```

Resumen Laboratorio IV - BackEnd (2022)

```
caracteristica_telefonica: int
# Con = None se vuelve opcional, esto es para que no de error cuando trae la lista de localidades
provincia: ProvinciaApi = None
# Se crea la clase de nuevo porque ya no hereda de BaseModel

class Config:
    orm_mode = True

class LocalidadApi(LocalidadSinId):
    id: int
```

Teoría: Como hacer los JOIN con SQL Alchemy - > https://docs.sqlalchemy.org/en/14/orm/loading_objects.html

Left outer join: <https://stackoverflow.com/questions/39619353/how-to-perform-a-left-join-in-sqlalchemy> (isouter)

TP FINAL:

VER:

Clase6_Backend-Geolocacion: (Al inicio)

Cosas que siempre faltan:

Validaciones:

 Fechas (Edad de compra por ejemplo)

 Letras donde va un número

Ambos saber como esta echo el Front y Back