FastAPI:

API interface program aplication es una sección de la aplicación que permite conectar con las diferentes partes del software y transmitir información a través de ellas.

Las api transmiten información a través del formato JSON. En Python es un diccionario.

+ framework más veloces del mundo para crear backend con python. Compite con Go y Nodejs.

+open source: código abierto (permitido hacer un fork desde github y pullrequest)

+ FastAPI utiliza otros frameworks dentro de si para funcionar:

Uvicorn: es una librería de Python que funciona de servidor, es decir, permite que cualquier computadora se convierta en un servidor

Starlette: es un framework de desarrollo web de bajo nivel, para desarrollar aplicaciones con este requieres un amplio conocimiento de Python, entonces FastAPI se encarga de añadirle funcionalidades por encima para que se pueda usar mas fácilmente

Pydantic: Es un framework que permite trabajar con datos similar a pandas, pero este te permite usar modelos los cuales aprovechará FastAPI para crear la API

# Las api transmiten información, respondiendo en formato JSON

#EN python un diccionario.

#Ejecución de la app: Utilizando uvicorn en la terminal.

#Corriendo el programa en localhost.

# uvicorn main:app --reload

# + main es el archivo a ejecutar

# + :app indica la variable que creamos para controlar el proyecto

# + --reload es el modificador para lograr el efecto de modificar el código

# y ver el cambio al instante sin necesidad de guardar el archivo y recargar

"""

Programa que utilizando el método get de fastapi

retorna una respuesta json (diccionario)

from fastapi import FastAPI

app = FastAPI()

@app.get("/") #PATH que ejecuta la path function

def home():

return {"Hello":"World"}

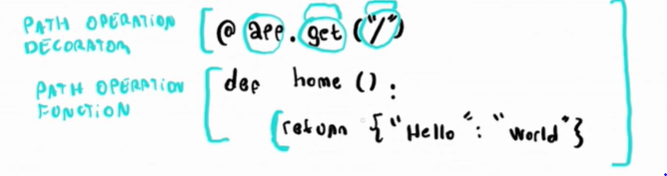
"""

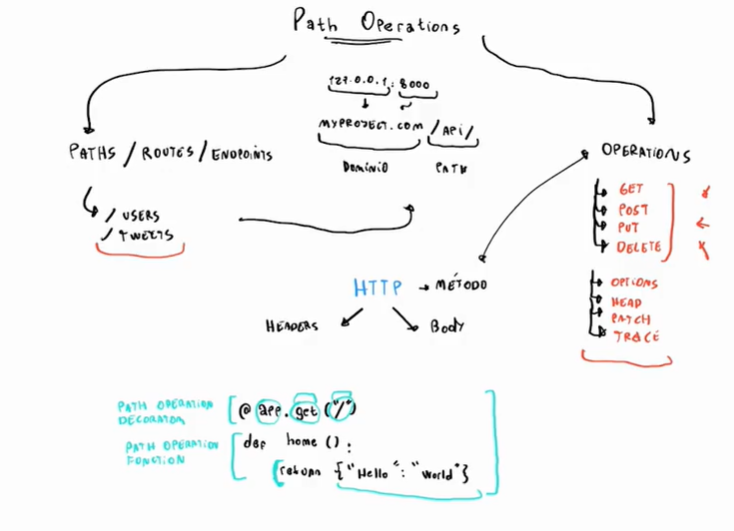
**Documentación:** Tomas las reglas provistas por OpenAPI para la creación y correcta definición de una api. A través de Swagger, y Swagger UI, toma las especificaciones y las muestra en Fast Api.

**PATH OPERATIONS: Son la base de nuestra aplicación.** Es la combinación entre una ruta de acceso de nuestra api (RUTA-PATH-ENDPOINT) y un método HTTP (Protocolo de intercambio de hyper texto: GET. POST, PUT, DELETE).

Mediante un decorador PATH OPERATION DECORATOR permito la ejecución de una PATH OPERATION FUNCTION.

Explicación del código sería “si alguien accede a el pat ( ) con el método (get) obtendrá el resultado de la operation function (return {JSON}”.





**Enviar variables obligatorias a nuestro método:**

Los PATH PARAMETERS son variables que se envían definidas dentro del PATH. Cuando la función requiera una variable se la indica con el formato de format de Python **{VARIABLE}**.

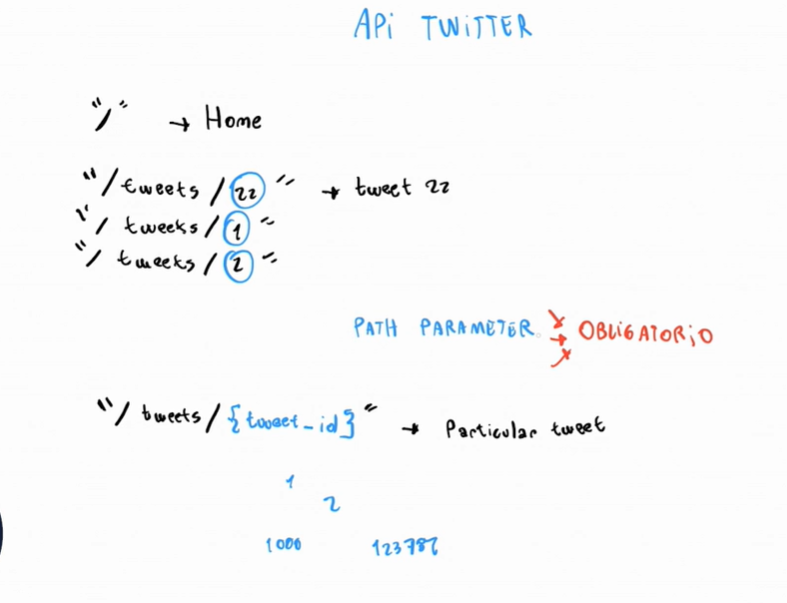
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item\_id}") #PATH

async def read\_item(item\_id):

return {"item\_id": item\_id}



**Query parameters:** Variables opcionales que no son obligatorias cuando se envía el path. En caso de enviarlas van definidas en el path.

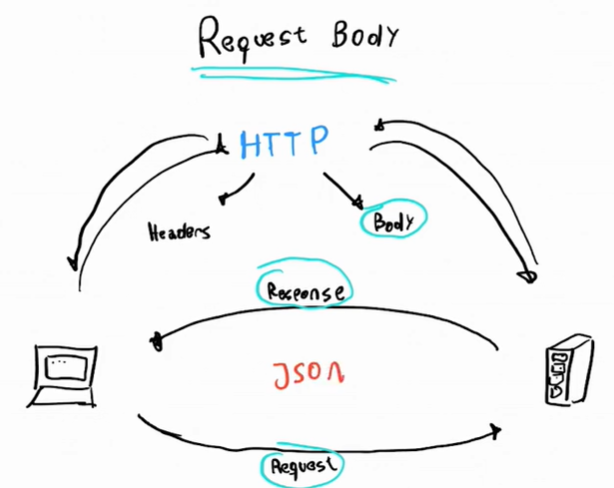
Se definen a partir del signo**?** variable = valor y se separan por &.

/users/**{user\_id}**/details**?age=20&height=184**

No afecta al sitio si se borran los parámetros del signo de interrogación en adelante.

**Request** es la petición de un cliente a un servidor

**Response** es la respuesta del servidor al cliente.



**#Request and Response Body**

"""

#Python

**from typing import Optional # Para tipado estático, tipo Optional**

#librería Pydantic

**from pydantic import BaseModel # Para crear modelos clase BaseModel**

#Importar del módulo, la clase FastAPI

from fastapi import FastAPI

**from fastapi import Body #Importar clase Body**

#Modelo es la representación descriptiva de una entidad(objeto de la vida real)

#Creación de un Modelo

class Person(BaseModel): #Hereda de BaseModel

first\_name: str #definición por tipado estático

last\_name: str

age: int

hair\_color: Optional[str] = None # El tipo optional espera un str, en caso de no recibir valores sera none (equivalente a null para base de datos)

is\_married: Optional[bool] = None

"""

#Método POST para crear

# Cuando el cliente de la api ejecute un método POST al path indicado,

# enviará un argumento person como el modelo que indica la documentación de la api.

**#el request body es un parámetro que no se coloca en la url pero si en la definición**

@app.post("/person/new") #path de mi api

def create\_person(person: Person = Body(...)): #Tipado estático variable:tipo // = Body(...) indica que el parámetro recibido será de tipo body y obligatorio

return person

**Documentación de pydantic y tipos de datos especiales:** https://pydantic-docs.helpmanual.io/usage/types/#pydantic-types

**Automatizar un ejemplo de JSON para probar los path operator:**

Crear una clase Config dentro del modelo. Dentro un atributo **schema\_extra** que contiene un diccionario, dentro de este un “example” : { }

Otra forma es utilizando el atributo example de la clase FIELD. Pero FastAPI tiene incompatibilidad para tomar ejemplos cuando el tipo de dato es un ENUM.

class Location(BaseModel):

    city : str = Field(

        ...,

        min\_length = 1,

        max\_length = 50

#**example = “BsAs”**

        )

    country : str = Field(

        ...,

        min\_length = 1,

        max\_length = 50

#**example = “Argentina”**

        )

**class Config:**

**schema\_extra = {**

**"example" : {**

**"city" : "Toronto",**

**"country" : "Canadá"**

**}**

**}**

**Añadir ejemplos a los path y query parameters con el atributo: example = <dato del tipo esperado>**

**AYUDAS**

**Instalación dentro de un entorno virtual activado:** $ pip install fastapi uvicorn

**Ejecutar programa con uvicorn en localhost:** $ uvicorn main:app --reload

(Abrir la <http://127.0.0.1:8000/>)

**Acceder a la documentación con Swagger: <**localhost>/docs

**Instanciar FASTAPI para manejar nuestro programa**

from fastapi import FastAPI

app = FastAPI()

**Método get con el path home /**

@app.get("/")

Def home ():

Return {…}

**¿Qué es el tipo Optional?**

Se puede usar para crear modelos o query parameters (pero estos con otro tipo llamado Query.

Permite que el usuario envíe el valor, y de no ocurrir cargar uno por defecto (En el json que se recibe no debe figurar este campo)

En el caso de los modelos, el usuario no envía los datos a través del path lo hace a través de un Json y si no envía un atributo, este se carga por defecto.

En el caso de los query parameters el usuario envía los datos a través del path. Y si este no fue enviado, se carga uno por defecto.

Se importa del módulo nativo de Python **from typing import Optional** y se usa con tipado estático indicando el tipo de dato que se espera, seguido del = el valor por defecto.

hair\_color: Optional[str] = None

**Crear un modelo y aplicar Optional**

El modelo es la representación descriptiva de una entidad del mundo real.

Debe heredar de la clase BaseModel de la librería pydantic y usar tipado estático para

definir el tipo de dato de cada propiedad.

El usuario no envía el modelo a través del path, lo hace a través de un JSON.

from pydantic import BaseModel

class Person(BaseModel): #Hereda

first\_name: str #definición por tipado estático

last\_name: str

age: int

hair\_color: Optional[str] = None Optional espera un str, en caso de no recibir valores será none (equivalente a null para base de datos)

is\_married: Optional[bool] = None

**Cómo Solicitar al usuario el envío de un Request Body Obligatorio:**

#Necesito tipado estático

from typing import Optional # Para tipado estático, tipo Optional

#importación de la clase Body

from fastapi import Body

Los request body no van en el path.

El request body se ajusta a un modelo ya definido.

Con **= Body(…)** se indica que el usuario esta obligado a pasar el tipo request body

Como ejemplo: API/person/new y a través de swagger se envía el modelo JSON del usuario.

@app.post("/person/new")

def create\_person(person: Person = Body(...)): #Tipado estático variable:tipo

return person

**Para Solicitar variables que no son obligatorias Query Parameters, se combina el uso de Optional y Query:**

**Así mismo Validar los datos enviados.**

Se envían a través del path Ejemplo: /users/**{user\_id}**/details**?age=20&height=184**

**Optional** indica el tipo de valor que espera, en caso de no recibir, se define un Query() el cual a su vez cuenta con parámetros de VALIDACIÓN en caso de que el usuario si envíe datos.

El tipo query también se importa de: **from typing import Query**

VALIDAR: Query permite validaciones al momento de enviar pedidos al servidor:

Para validar strings:

max\_length =

min\_length =

regex =

Para validar números

ge - greater or equal than >= (Mayor igual que)

le - less or equal than <= (Menor igual que)

gt - greater than > (Mayor)

lt - less than < (Menor)

Si en la documentación se quiere agregar un título como información de las variables requeridas están las propiedades:

= Query( **None**, title="ID del usuario", description="El ID se consigue entrando a las configuraciones del perfil");

También puede requerir que el query parameter se obligatorio y no opcional

= Query(…)

from typing import Optional

from fastapi import FastAPI

**from fastapi import Body, Query**

app = FastAPI()

# Validations: Query Parameters

# Se envían a través del path. Se detallan después del sigo ?. Se separan por &.

# Hay casos donde se necesitan query parameters obligatorios.

# Query tiene propiedades para validar ue los datos sean correctos.

# Ejemplo "API/person/detail?name=xxx&age=20"

@app.get("/person/detail")

def show\_person(

name: Optional[str] = Query( #El valor opcional debe cumplir los requisitos establecidos

None, # El valor por defecto

min\_length=1,

max\_length=50,

title="Name",

description="Enter Person Name"),

age: int = Query(

..., #En este caso pasar el parametro age en el path es oligatorio ”?age=50”.

ge = 18,

lt =100)

):

return {name: age}

**Crear operaciones con PATH PARAMETERS y Validarlos.**

**Estos son obligatorios y el usuario los debe incluirlo en el PATH.**

**En la definición de parámetros de la función se especifican los path parameters y sus propiedads junto con los Bodys. Separados por coma.**

Ejemplo de ENDPOINT: api/person/detail/**100**

Necesito importar el tipo Path de: **from fastapi import Path**

Propiedades para validar un path parameter:

… indica que es obligatorio

max\_length =

min\_length =

regex =

Para validar números

ge = greater or equal than >= (Mayor igual que)

le = less or equal than <= (Menor igual que)

gt = greater than > (Mayor)

lt = less than < (Menor)

title =

description =

Los títulos pueden aparecer o no según swagger, ver con redoc.

Si existe dos path operations iguales con el mismo path, fastAPI toma el primero.

**@app.get("/person/detail/{person\_id}") Path decorator**

def show\_person(

person\_id: int **= Path**( # Indicación que sera un path parameter obligatorio, mayor que cero y un título con descripción.

...,

gt = 0,

title = "Person id",

description = "Enter a person id > 0"

)):

return {

person\_id:'The person id '+str(person\_id)+' exists'

}

"""

**CREACIÓN DE MODELOS Y VALIDACIÓN DE REQUEST BODY:**

"""

Validación de Modelos y Request Body:

#Creación de un Modelo: Sirven para definir que tipo de schema debe tener el JSON que recibe la función. Validar es establecer el tipo de dato que debe tener el JSON, si el usuario envía otro dato la app no permitirá que se ejecute un proceso y le retornará una respuesta de error.

Necesito las librerías:

from pydantic import BaseModel para crear clases que lo heredan.

from pydantic import Field para validar campos.

A cada atributo se define el tipo : y luego se declara que es un tipo Field con sus propiedades.

Field permite un valor default.

El tipo Optional define el tipo de dato que espera dentro de los []

En caso de no recibir valores se declara seguido del signo =.

"""

"""

La clase enum se importa from enum import Enum.

Permite crear un tipo de dato que solo acepta los valores indicados en esta clase. Hereda de (Enum).

"""

class HairColor(Enum):

    brown = "brown"

    red = "red"

    blonde = "blonde"

    black = "black"

    white = "white"

class Person(BaseModel):

    first\_name: str = Field(

        ...,

        min\_length = 1,

        max\_length = 50

        )

    last\_name: str = Field(

        ...,

        min\_length = 1,

        max\_length = 50

        )

    age: int = Field(

        ...,

        ge = 18,

        le = 100

        )

credit\_card\_number : Optional[PaymentCardNumber] = Field(default=None)

color: Color = Field(default="blue")

hair\_color: Optional[HairColor] = Field( default= None )

    hair\_color: Optional[HairColor] = Field( default= None )

    is\_married: Optional[bool] = Field( default = False )

class Location(BaseModel):

    city : str = Field(

        ...,

        min\_length = 1,

        max\_length = 50

        )

    country : str = Field(

        ...,

        min\_length = 1,

        max\_length = 50

        )

Método PUT para actualizar un elemento. Validación de path parameter y request body.

Recibir dos Json.

En el path decorator se define el path parameter con el id

Dentro de los parámetros que define la función se define:

+ el path parameter y su validación,

+ los Body que se deben enviar.

La función convierte de Json a diccionario con el método .dict()

Luego al diccionario creado le añade las llave valor de otro con el método .update()

FastApi permite pasar cuando se necesita más de un Body un dos llaves {} con los diccionarios en su interior separados por comas.

"""

@app.put("/person/{person\_id}")

def update\_person(

    person\_id : int = Path(

        ...,

        title = "Person ID",

        description = "This is the person ID",

        ge = 1

    ),

    person : Person = Body(...),

    location : Location = Body(...)

    ):

    result = person.dict()

    result.update(location.dict())

    return result