01076105, 01075106

Object Oriented Programming

Object Oriented Programming Project

Communication

JSON

- JSON ย่อมาจาก JavaScript Object Notation เป็นมาตรฐานการแลกเปลี่ยนข้อมูลระหว่าง Server และ Client ที่ได้รับความนิยมในปัจจุบัน
- อักขระมาตรฐานของ JSON
  - เครื่องหมาย “:” ใช้สำหรับแยกค่า name และ value
  - เครื่องหมาย “,” ใช้สำหรับแยกข้อมูล name-value ในแต่ละคู่
  - เครื่องหมาย “{” และ “}” ระบุว่าข้อมูลเป็น Object
  - เครื่องหมาย “[” และ “]” ระบุว่าข้อมูลเป็นอาเรย์
  - เครื่องหมาย “ ” (double quotes) ใช้เขียนค่า name-value ใน JSON

JSON

- ข้อมูล 1 คู่

  "name" : "value"

- ข้อมูล 2 คู่ ใช้ เครื่องหมายคอมมา , (comma) ในการแยกเป็นคู่

  "name" : "value", "name" : "value", "name": "value"

- ใช้เครื่องหมาย { } ในการระบุว่าเป็น Object

```
{
  "name" : "Dwayne Johnson",
  "email" : “johnson@email.com",
}
```

JSON

- ชนิดข้อมูลของ JSON มี 6 ชนิด คือ

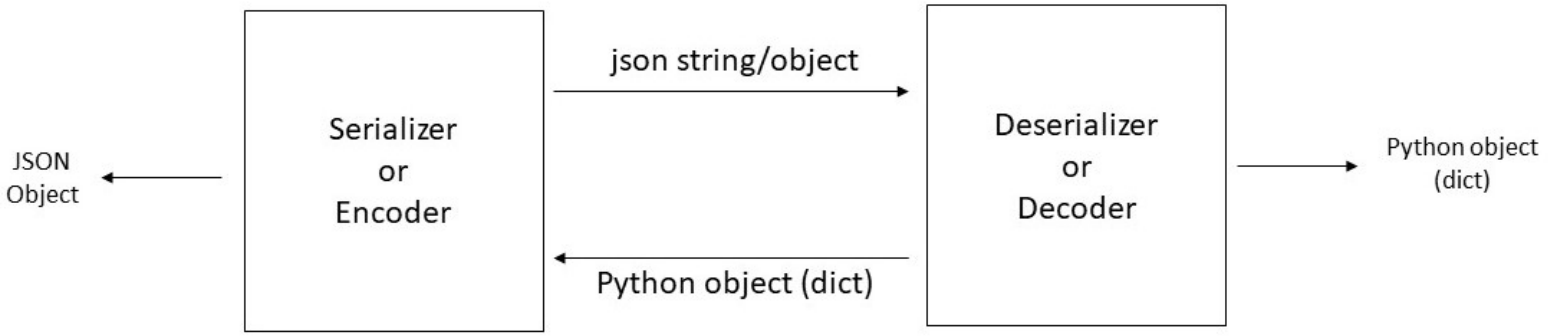
1. strings
2. numbers
3. objects
4. arrays
5. Boolean
6. null or empty

```
{
   "text" : "This is Sting",
   "number" : 210,
   "object" :  {
                    "name" : "abc",
                    "popularity" : "immense"
               },
   "arary" : ["1","2","3"],
   "empty" : ,
   "booleans" : true
}
```

JSON

- เนื่องจาก JSON มีลักษณะเป็นคู่ key:value คล้ายกับกับ dictionary ใน python จึงมักใช้ dictionary ในการแปลงเป็น JSON
- การแปลง Python Object ไปเป็น JSON จะเรียกว่า Serialization
- การแปลง JSON มาเป็น Python Object จะเรียกว่า Deserialization

กระบวนการ Serialization และ Deserialization



JSON

- การแปลง Python Object (Dict) ไปเป็น JSON String หรือ Object ทำได้โดยเรียกใช้ json.dumps()

```python
import json

test_dict = {
    "name": "Python",
    "author": "Guido Van Rossum",
    "year": 1990,
    "frameworks": ["Flask", "Django"],
    "libraries": ["Pandas", "Numpy", "Matplotlib", "Requests"]
}

print(type(test_dict))
j_string = json.dumps(test_dict)
print(j_string)
print(type(j_string))
```

JSON

- การแปลง JSON String ไปเป็น Python Object (Dict) ทำได้โดยเรียกใช้ json.loads()

```
import json

prog_string = '''
    {
        "name": "Python",
        "author": "Guido Van Rossum",
        "year": 1990,
        "frameworks": ["Flask", "Django"],
        "libraries": ["Pandas", "Numpy", "Matplotlib", "Requests"]
    }'''

print(type(prog_string))
prog_dict = json.loads(prog_string)
print(prog_dict)
print(type(prog_dict))
print(prog_dict["name"])
```

Socket

- Socket ในความหมายภาษาอังกฤษ อาจมองว่าคล้ายๆ กับ ช่องเสียบที่สามารถเสียบเข้าหากันเพื่อสื่อสารระหว่างโปรแกรม หรือ ระหว่างเครื่อง
- Socket ประกอบด้วย IP Address และ Port Number

> Socket Address = IP Address + Port Number

- ใน Python มี Library ชื่อ socket ซึ่งต้อง import แล้วจึงใช้ได้

```
import socket

TCP_IP = "127.0.0.1"
TCP_PORT = 8081
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #init tcp
s.connect((TCP_IP,TCP_PORT)) #เชื่อมต่อ
```

Socket

- โปรแกรมฝั่ง Client

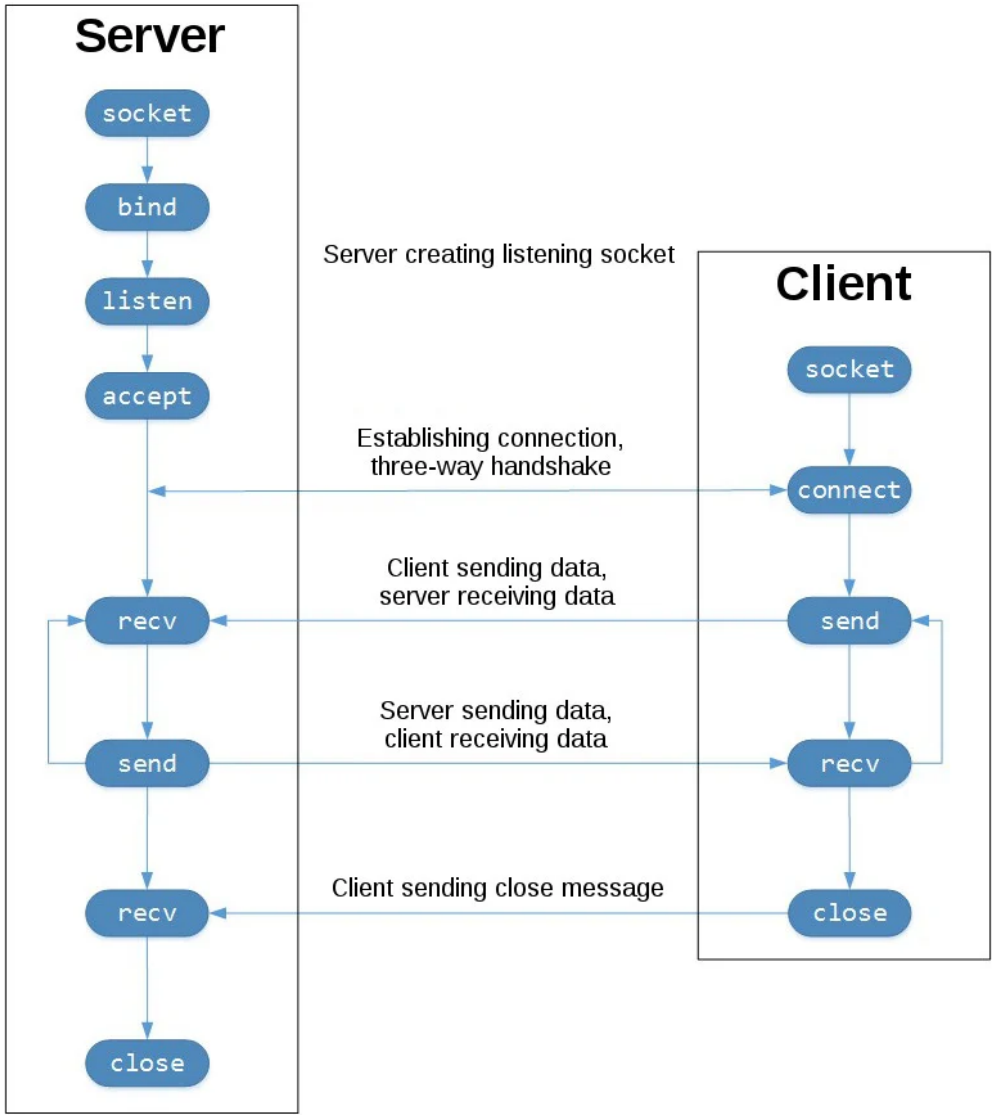
```python
import socket

TCP_IP = "127.0.0.1"
TCP_PORT = 8081
BUFFER_SIZE = 1024
MESSAGE = "Hello, World!"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((TCP_IP,TCP_PORT)) #เชื่อมต่อ
s.send(MESSAGE.encode('utf-8')) # ส่ง / เข้ารหัส utf-8
data = s.recv(BUFFER_SIZE) # รับข้อมูล
s.close()

print("received data:", data)
```

Socket

- ฟังก์ชัน connect จะใช้ในการเชื่อมต่อกับ Socket ใน TCP/IP Stack เป็นการขอการเชื่อมต่อกับ Network Layer ของ OS
- send ใช้ในการส่งข้อความไปยังฝั่งตรงข้าม แต่จะส่งได้จะต้องผ่านการ connect สำเร็จก่อน มิฉะนั้นจะส่งไม่ได้
- recv ใช้ในการรับข้อมูลจากฝั่งตรงข้าม โดยในการรับจะต้องมีการกำหนดขนาดสูงสุดที่จะรับในแต่ละครั้ง เรียกว่า Buffer Size
- เมื่อจะสิ้นสุด connection ต้องมีการ close

Socket

- โปรแกรมฝั่ง Server จะซับซ้อนกว่าเมื่อเทียบกับ Client
- คำสั่ง bind จะเชื่อมโยง IP กับ Port เข้ากับ Socket
- คำสั่ง listen จะสั่งให้ server คอย monitor การเชื่อมต่อที่เรียกเข้ามา
- คำสั่ง accept จะรับการเชื่อมต่อและติดต่อกันได้



Socket

```python
import socket

TCP_IP = "127.0.0.1"
TCP_PORT = 8081
BUFFER_SIZE = 1024

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((TCP_IP,TCP_PORT)) # bind with socket
s.listen(1)
conn, addr = s.accept() # receive data from client.py
print('Connection address:', addr) #ip address client ที่เชื่อมต่อมา

while True:
    data = conn.recv(BUFFER_SIZE)
    if not data:
        break
    print("received data:", data)
    conn.send(data) # echo

conn.close()
```

Socket

- การใช้ Socket ค่อนข้างเป็นเรื่องยุ่งยาก
- จึงไม่นิยมใช้ในการติดต่อมากนัก
- มีการใช้งานใน Low Level เท่านั้น
- นอกจากนั้นยังต้องบริหารจัดการเรื่องของ Thread เองอีกด้วย

Requests Library

- เป็น Library สำหรับเรียกใช้ API จากที่ต่างๆ เพื่อนำข้อมูลมาใช้งาน
- ให้ติดตั้ง Library ชื่อ requests และรันโปรแกรมนี้

```
import requests

r = requests.get('https://covid19.ddc.moph.go.th/api/Cases/today-cases-all')
print(r)
print(dir(r))
```

- จะพบว่าได้ผลดังนี้

<Response [200]>
['__attrs__', '__bool__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__nonzero__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_content', '_content_consumed', '_next', 'apparent_encoding', 'close', 'connection', 'content', 'cookies', 'elapsed', 'encoding', 'headers', 'history', 'is_permanent_redirect', 'is_redirect', 'iter_content', 'iter_lines', 'json', 'links', 'next', 'ok', 'raise_for_status', 'raw', 'reason', 'request', 'status_code', 'text', 'url']

Requests Library

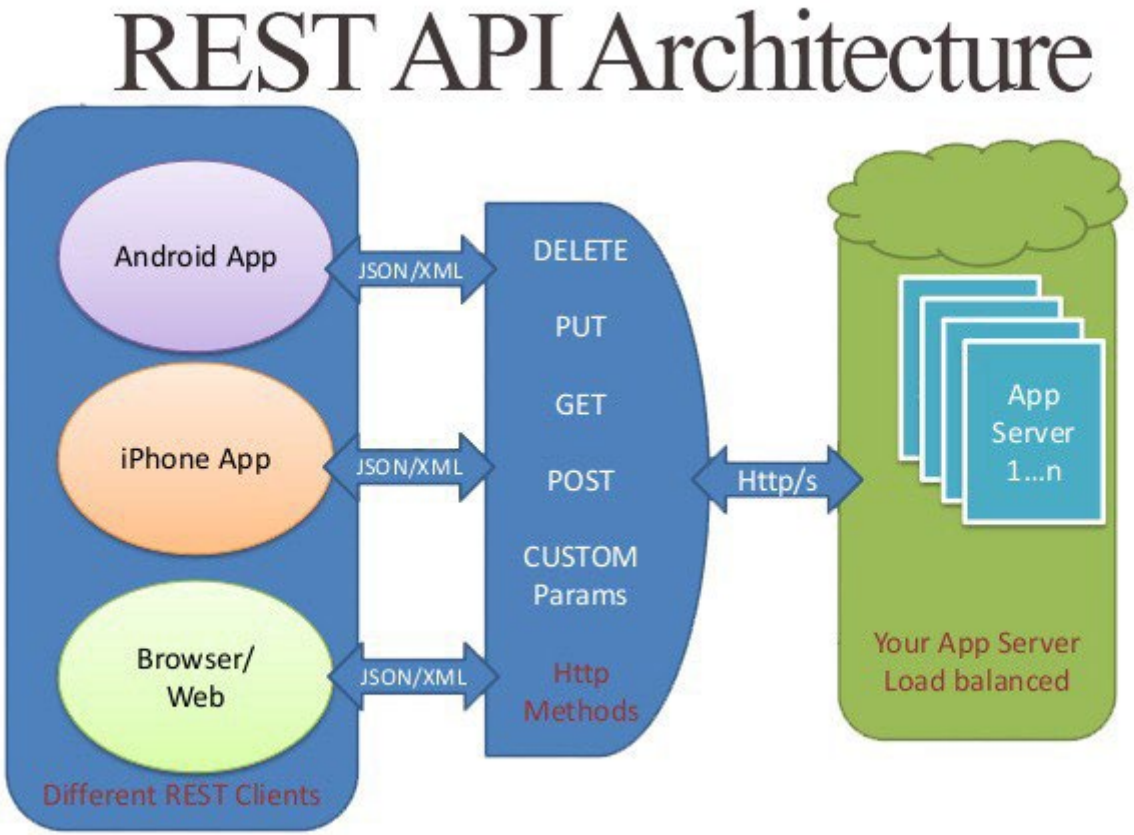
- ตัว Object ที่ส่งกลับมาเรียกว่า Response Object โดยสามารถนำมาใช้งานตามตัวอย่าง

```
import requests
import json

url = 'https://covid19.ddc.moph.go.th/api/Cases/today-cases-all'

req = requests.get(url)
data = req.json()

print("ผู้ป่วยรายวัน:",data[0]['new_case'])
print("ผู้ป่วยสะสม:",data[0]['total_case'])
print("ผู้เสียชีวิต:",data[0]['new_death'])
print("หายป่วย:",data[0]['new_recovered'])
```

FastAPI

- FastAPI เป็น Library ที่ช่วยในการสร้าง Rest API
- REST ย่อมาจาก Representational State Transfer
- REST เป็นรูปแบบการส่งข้อมูลระหว่าง Server-Client รูปแบบหนึ่ง บนพื้นฐานของ HTTP Protocol
- เป็นการสร้าง Web Service เพื่อแลกเปลี่ยนข้อมูลกันผ่าน Application วิธีหนึ่ง
- ส่งข้อมูลได้หลายชนิด ไม่ว่าจะเป็น Text, XML, JSON หรือส่งมาเป็นหน้า HTML เลยก็ได้

FastAPI


REST API Architecture
Android App
iPhone App
Browser/
Web
Different REST Clients
JSON/XML
JSON/XML
JSON/XML
DELETE
PUT
GET
POST
CUSTOM
Params
Http
Methods
Http/s
App
Server
1...n
Your App Server
Load balanced

FastAPI

- การใช้ Rest API นั้นจะอยู่บนพื้นฐาน HTTP Method เช่น GET (เรียกดูข้อมูล), POST(เพิ่มข้อมูล), PUT(แก้ไขข้อมูล), DELETE(ลบข้อมูล) เราจะใช้ Method ไหน เมื่อไร ก็ขึ้นอยู่กับว่าเราจะทำอะไรกับข้อมูล
- ในการใช้ Fast API เราต้องติดตั้ง Library ก่อน โดยต้องให้ใช้คำสั่ง
  - pip install fastapi
  - pip install uvicorn
- โดย uvicorn จะทำหน้าที่เป็น Mini Web Server
- สำหรับผู้ที่ใช้ PyCharm จะมีติดตั้งมาให้แล้ว

FastAPI

- หลังจากติดตั้งเรียบร้อย ให้ copy code จาก https://fastapi.tiangolo.com/

```python
from typing import Optional
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

FastAPI

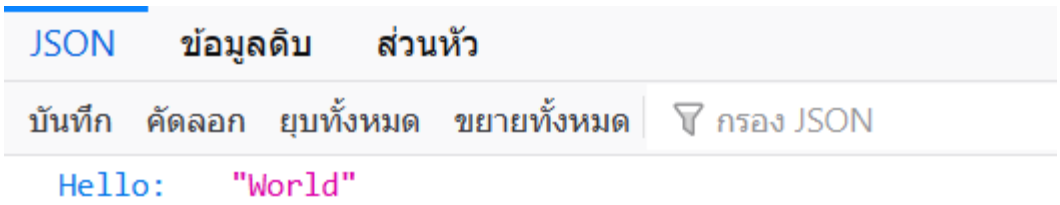
- จากนั้น ให้ไปที่ terminal แล้วเรียกคำสั่ง ตามรูป (สมมติโปรแกรมชื่อ main.py)

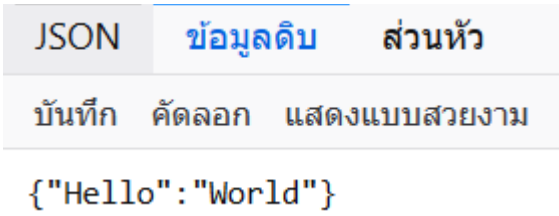
```
$ uvicorn main:app --reload

INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [28720]
INFO:     Started server process [28722]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

- แล้วเปิด Browser แล้วป้อน http://127.0.0.1:8000

FastAPI

- จะแสดงหน้าจอดังนี้ (การแสดงผลอาจจะต่างกันไปในแต่ละเครื่อง)





- หน้าจอข้างต้นมาจาก code ดังนี้ (“/” เรียกว่า endpoint)

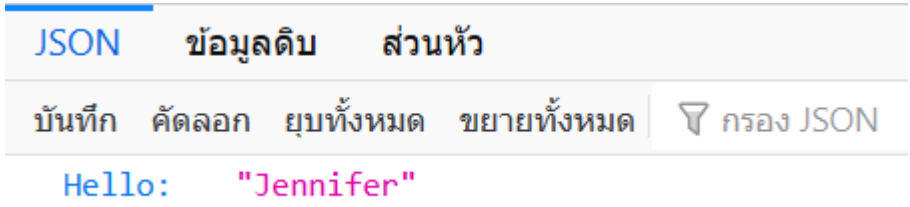
```python
@app.get("/")
def read_root():
    return {"Hello": "World"}
```

FastAPI

- สมมติว่าแก้โปรแกรมเป็น (เปลี่ยน endpoint และเพิ่ม parameter)

```python
@app.get("/hello")
def read_root(name:str):
    return {"Hello": name}
```

- แล้วป้อน URL เป็น http://127.0.0.1:8000/hello?name=Jennifer
- หน้าจอจะแสดง ซึ่งแสดงว่าเราสามารถ pass parameter เข้าไปได้

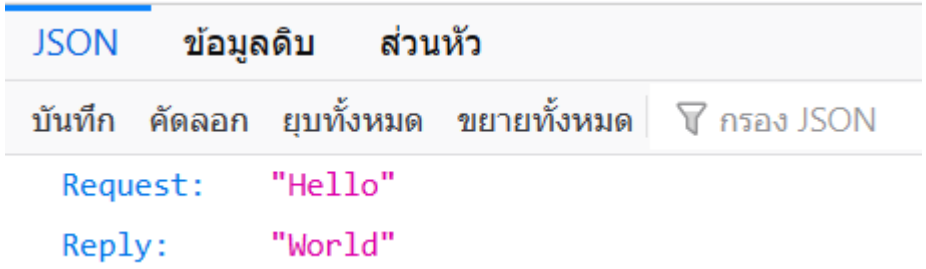


FastAPI

- สมมติว่าแก้โปรแกรมเป็น

```python
@app.get("/test")
def read_root(request:str, reply:str):
    return {"Request": request, "Reply": reply}
```

- แล้วป้อน URL เป็น http://127.0.0.1:8000/test?request=Hello&reply=World
- จะเห็นว่าส่ง parameter ได้หลายตัว

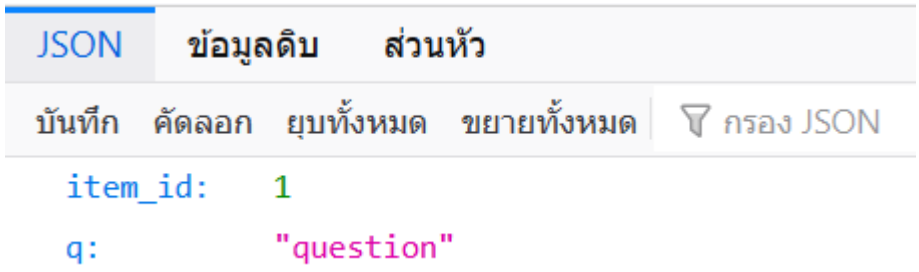


FastAPI

- ในโปรแกรมตัวอย่างจะมีอีกส่วน คือ

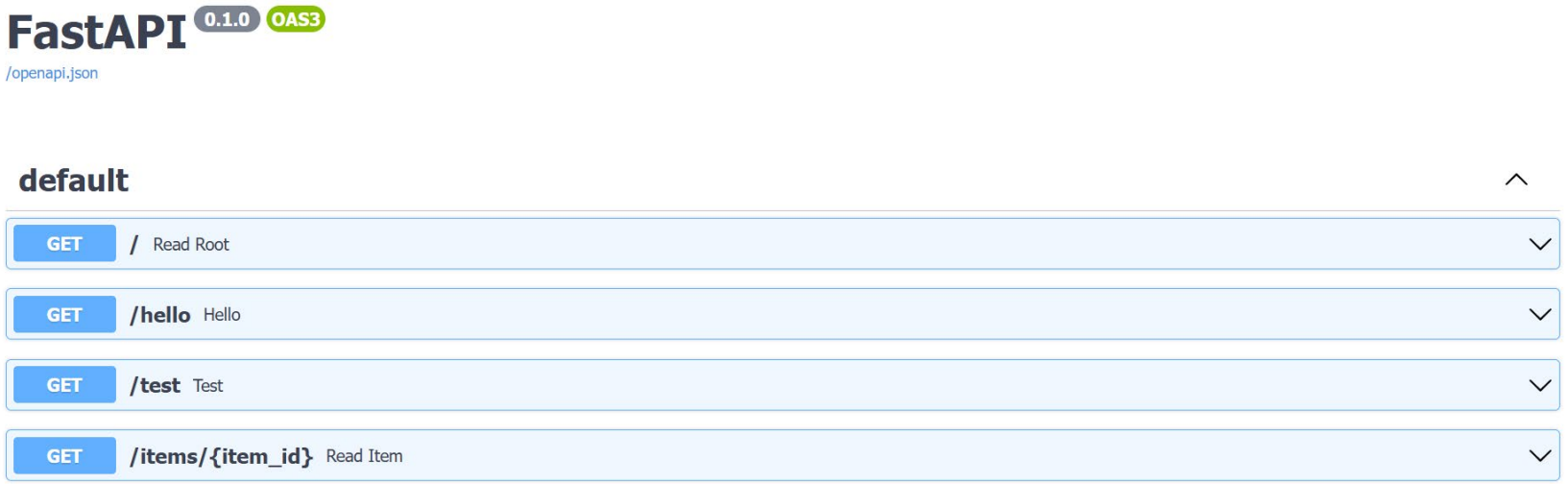
```python
@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

- จะเห็นว่าเราสามารถเอาส่วนหนึ่งของ URL มาใช้เป็น parameter ได้
- ส่วนที่เขียนว่า Optional คือ ถ้าไม่ได้ใส่ จะถือว่าเป็น None
- สมมติป้อน URL เป็น http://127.0.0.1:8000/items/1?q=question



FastAPI

- ให้ป้อน URL http://127.0.0.1:8000/docs จะแสดงหน้าจอดังนี้



- เรียกว่า Swagger UI ใช้สำหรับทดสอบ API
- มีความสามารถในการสร้างฟอร์มเพื่อ Input สำหรับแต่ละ API

FastAPI

- ที่ผ่านมาเป็น HTTP GET ทั้งหมด แต่ HTTP Method จะมีด้วยกัน 4 วิธี
  - GET : ขอดึงข้อมูล เมื่อได้รับคำสั่ง Server API
  - POST : ขอเพิ่มข้อมูล
  - PUT : ขอแก้ไขข้อมูล
  - DELETE : ขอลบข้อมูล

FastAPI

- จะสมมติ App ง่ายๆ ขึ้นมา 1 ตัว คือ todo list โดยจะเก็บข้อมูลใน List

```
todos = [
    {
        "id": "1",
        "Activity": "Jogging for 2 hours at 7:00 AM."
    },
    {
        "id": "2",
        "Activity": "Writing 3 pages of my new book at 2:00 PM."
    }
]
```

FastAPI

- จะทำ Method GET ก่อน เพื่อดึงข้อมูลมาแสดงโดยทำ function ชื่อ get_todos โดยจะ return ค่ากลับเป็น dict โดย ประกอบด้วย ค่า key คือ “Data” และ value เป็น List todos

```python
# A minimal app to demonstrate the get request
@app.get("/", tags=['root'])
async def root() -> dict:
    return {"Ping": "Pong"}


# GET -- > Read Todo
@app.get("/todo", tags=['Todos'])
async def get_todos() -> dict:
    return {"Data": todos}
```

FastAPI

- เมื่อเปิด Swagger แล้วเรียก Execute จะพบว่าสามารถส่ง list กลับมาได้ครบ

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/todo
```

Server response

Code	Details
200	Response body

```
{
  "Data": [
    {
      "id": "1",
      "Activity": "Jogging for 2 hours at 7:00 AM."
    },
    {
      "id": "2",
      "Activity": "Writing 3 pages of my new book at 2:00 PM."
    }
  ]
}
```

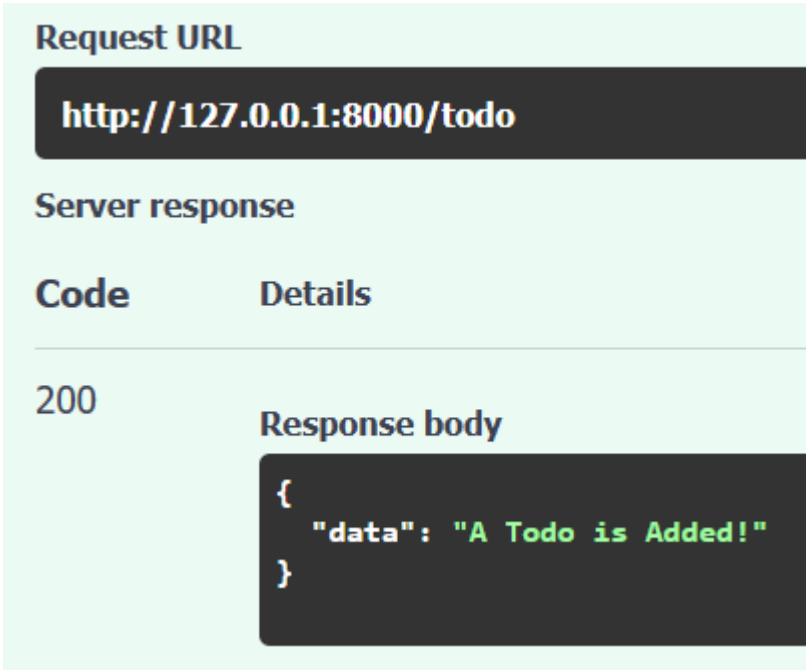
FastAPI

- จากนั้นจะทำ Method POST โดยรับข้อมูลเป็น dict ในชื่อ todo จากนั้นก็นำ todo ไป append ใน todos อีกที จากนั้นก็ส่งค่ากลับเป็น dict ประกอบด้วย ค่า key คือ “data” และ value เป็นข้อความ “A Todo is Added!”

```python
# Post -- > Create Todo
@app.post("/todo", tags=["Todos"])
async def add_todo(todo: dict) -> dict:
    todos.append(todo)
    return {
        "data": "A Todo is Added!"
    }
```

FastAPI

- คราวนี้จะทดลองเพิ่มข้อมูลเข้าไป โดยใช้ POST โดยป้อนข้อมูลตามรูป เมื่อกด Execute จะส่งข้อมูลกลับมา



FastAPI

- จากนั้นจะทดลอง GET ดูอีกครั้ง ว่าข้อมูลมีการเปลี่ยนหรือไม่ ก็พบว่าได้ผลลัพธ์

Code **Details**

200

Response body


```
{
  "Data": [
    {
      "id": "1",
      "Activity": "Jogging for 2 hours at 7:00 AM."
    },
    {
      "id": "2",
      "Activity": "Writing 3 pages of my new book at 2:00 PM."
    },
    {
      "id": "3",
      "Activity": "Play football"
    }
  ]
}
```


Download

FastAPI

- ต่อไปจะทำ Method PUT คือ การแก้ไขข้อมูล จะมีการรับข้อมูลผ่าน Path และ Request Body โดยโปรแกรม คือ ค้นหา id และแก้ไข Activity ของ id นั้น

```python
# PUT  -- > Update Todo
@app.put("/todo/{id}", tags=["Todos"])
async def update_todo(id: int, body: dict) -> dict:
    for todo in todos:
        if (int(todo["id"])) == id:
            todo["Activity"] = body["Activity"]
            return {
                "data": f"Todo with id {id} has been updated"
            }
    return {
        "data": f"This Todo with id {id} is not found!"
    }
```

FastAPI

- จะทดลองใช้งานผ่าน Swagger โดยมีหน้าตาดังนี้

FastAPI

- จากนั้นจะทดลอง GET ดูอีกครั้ง ว่าข้อมูลมีการเปลี่ยนหรือไม่ ก็พบว่าได้ผลลัพธ์

200

Response body


```
{
  "Data": [
    {
      "id": "1",
      "Activity": "Jogging for 2 hours at 7:00 AM."
    },
    {
      "id": "2",
      "Activity": "Writing 3 pages of my new book at 2:00 PM."
    },
    {
      "id": "3",
      "Activity": "Play games"
    }
  ]
}
```

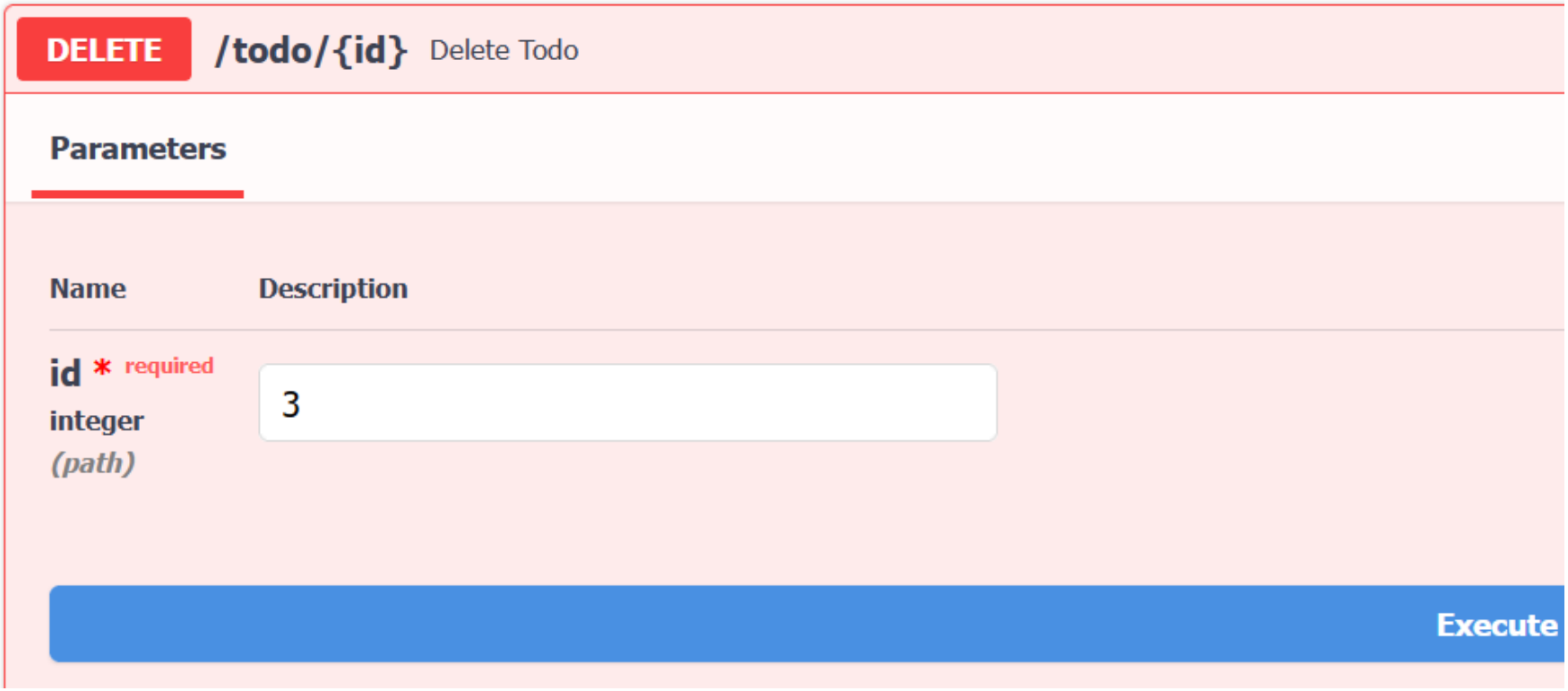
FastAPI

- ต่อไปจะทำ Method DELETE โดยจะรับ Path parameter และค้นหาใน todos ถ้าพบก็ remove ออกไป

```python
# DELETE --> Delete Todo
@app.delete("/todo/{id}", tags=["Todos"])
async def delete_todo(id: int) -> dict:
    for todo in todos:
        if int(todo["id"]) == id:
            todos.remove(todo)
            return{
                "data": f"Todo with id {id} has been deleted!"
            }
    return {
        "data": f"Todo with id {id} was not found!"
    }
```

FastAPI

- จะทดลองใช้งานผ่าน Swagger โดยมีหน้าตาดังนี้



FastAPI

- จากนั้นจะทดลอง GET ดูอีกครั้ง ว่าข้อมูลมีการเปลี่ยนหรือไม่ ก็พบว่าได้ผลลัพธ์ตามรูป

Code	Details
200	Response body


```
{
  "Data": [
    {
      "id": "1",
      "Activity": "Jogging for 2 hours at 7:00 AM."
    },
    {
      "id": "2",
      "Activity": "Writing 3 pages of my new book at 2:00 PM."
    }
  ]
}
```

Requests

- ต่อไปจะทดลองใช้ Library Requests เพื่อทดสอบการทำงานของ API แทนการใช้ Swagger โดยเริ่มจาก GET

```python
import requests
import json

r = requests.get("http://127.0.0.1:8000/todo")
print(r)
print(r.json())
```

```
C:\Users\khtha\Desktop\Python OOP\Source\oop_class_2021\FastAPI>python request.py
<Response [200]>
{'Data': [{'id': '1', 'Activity': 'Jogging for 2 hours at 7:00 AM.'}, {'id': '2',
'Activity': 'Writing 3 pages of my new book at 2:00 PM.'}]}
```

Requests

- ทดสอบ POST

```
import requests
import json

act = {"id" : "3","Activity" : "Play football"}
r = requests.post("http://127.0.0.1:8000/todo",data=json.dumps(act))
print(r)
print(r.json())

r = requests.get("http://127.0.0.1:8000/todo")
print(r)
print(r.json())
```

```
<Response [200]>
{'data': 'A Todo is Added!'}
<Response [200]>
{'Data': [{'id': '1', 'Activity': 'Jogging for 2 hours at 7:00 AM.'}, {'id': '2',
'Activity': 'Writing 3 pages of my new book at 2:00 PM.'}, {'id': '3', 'Activity':
 'Play football'}, {'id': '3', 'Activity': 'Play football'}]}
```

Requests

- ทดสอบ PUT

```
import requests
import json

act = {"id" : "3","Activity" : "Play football"}
r = requests.post("http://127.0.0.1:8000/todo",data=json.dumps(act))
print(r)
print(r.json())

act = {"id" : "3","Activity" : "Play games"}
r = requests.put("http://127.0.0.1:8000/todo/3",data=json.dumps(act))
print(r)
print(r.json())

r = requests.get("http://127.0.0.1:8000/todo")
print(r)
print(r.json())
```

Requests

- ทดสอบ PUT
- จะเห็นว่าข้อมูลถูกแก้ไข

```
C:\Users\khtha\Desktop\Python OOP\Source\oop_class_2021\FastAPI>python request.py
<Response [200]>
{'data': 'A Todo is Added!'}
<Response [200]>
{'Data': [{'id': '1', 'Activity': 'Jogging for 2 hours at 7:00 AM.'}, {'id': '2',
'Activity': 'Writing 3 pages of my new book at 2:00 PM.'}, {'id': '3', 'Activity':
 'Play football'}]}
<Response [200]>
{'data': 'Todo with id 3 has been updated'}
<Response [200]>
{'Data': [{'id': '1', 'Activity': 'Jogging for 2 hours at 7:00 AM.'}, {'id': '2',
'Activity': 'Writing 3 pages of my new book at 2:00 PM.'}, {'id': '3', 'Activity':
 'Play games'}]}
```

Requests

- ทดสอบ DELETE

```
import requests
import json

act = {"id" : "3","Activity" : "Play football"}
r = requests.post("http://127.0.0.1:8000/todo",data=json.dumps(act))
print(r.json())

act = {"id" : "3","Activity" : "Play games"}
r = requests.put("http://127.0.0.1:8000/todo/3",data=json.dumps(act))
print(r.json())

r = requests.get("http://127.0.0.1:8000/todo")
print(r.json())

r = requests.delete("http://127.0.0.1:8000/todo/3")
print(r)
print(r.json())
```

Requests

- ทดสอบ DELETE

```
C:\Users\khtha\Desktop\Python OOP\Source\oop_class_2021\FastAPI>python request.py
{'data': 'A Todo is Added!'}
{'data': 'Todo with id 3 has been updated'}
{'Data': [{'id': '1', 'Activity': 'Jogging for 2 hours at 7:00 AM.'}, {'id': '2',
'Activity': 'Writing 3 pages of my new book at 2:00 PM.'}, {'id': '3', 'Activity':
 'Play games'}]}
<Response [200]>
{'data': 'Todo with id 3 has been deleted!'}
```

FastAPI

- คราวนี้จะเขียนเป็น Class todo โดยเก็บข้อมูลใน List เช่นเดิม แต่จะกำหนด id ให้

```python
class ToDoList:
    ID = 1

    def __init__(self, name):
        self.__name = name
        self.__tasks = []

    def add_task(self, todo):
        id = ToDoList.ID
        todo["id"] = id
        self.__tasks.append(todo)
        ToDoList.ID += 1
        return id
```

FastAPI

- ในการเพิ่มข้อมูล (method POST) แม้จะรับข้อมูลมาเป็น dictionary แต่จะกำหนดหมายเลขของ id ใหม่ โดยให้ Class ส่งคืนมาให้ จากนั้นก็ส่งหมายเลข id กลับคืนไป

```
my_list = ToDoList("My")

# Post -- > Create Todo
@app.post("/todo", tags=["Todos"])
async def add_todo(task: dict) -> dict:
    id = my_list.add_task(task)
    todo = "A Todo "+str(id)+" is added!"
    return {
        "data": todo
    }
```

FastAPI

- ในส่วนของ method GET จะส่งทั้ง list คืนไป

```python
# GET -- > Read Todo
@app.get("/todo", tags=['Todos'])
async def get_todos() -> dict:
    return {"Data": my_list.get_task()}
```

FastAPI

- ในส่วนของ method PUT จะสร้าง method

```
def modify_task(self, id, body):
    for todo in self.__tasks:
        if (int(todo["id"])) == id:
            todo["Activity"] = body["Activity"]
            return {
                "data": f"Todo with id {id} has been updated"
            }
    return {
        "data": f"This Todo with id {id} is not found!"
    }
```

FastAPI

- และเขียน API ดังนี้

```python
# PUT  -- > Update Todo
@app.put("/todo/{id}", tags=["Todos"])
async def update_todo(id: int, body: dict) -> dict:
    return my_list.modify_task(id, body)
```

FastAPI

- ในส่วนของ method DELETE จะสร้าง method

```
    def delete_task(self, id):
        for todo in self.__tasks:
            if int(todo["id"]) == id:
                self.__tasks.remove(todo)
                return {
                    "data": f"Todo with id {id} has been deleted!"
                }
        return {
            "data": f"Todo with id {id} was not found!"
        }
```

FastAPI

- และเขียน API ดังนี้

```python
# DELETE --> Delete Todo
@app.delete("/todo/{id}", tags=["Todos"])
async def delete_todo(id: int) -> dict:
    return my_list.delete_task(id)
```

For your attention