

EASS שיעור 9-SQL, DATACLASS, אלמנטים נוספים בפייתון, מבוא

docker compose

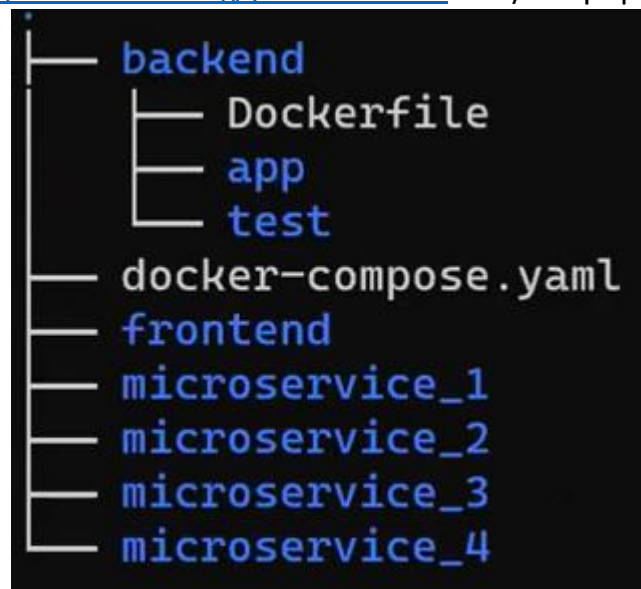
SQL (Structured Query Language) - שפה לניהול בסיסי נתונים יחסיים.
 מנועים שממשים את השפה - PostgreSQL, MySQL, SQLite.
 כדי לפנות לבסיס הנתונים נשלח למנוע בקשות HTTP שיכילו queries.
SQLAlchemy - ספרייה (toolkit) בפייתון שמתרגמת פקודות בשפת פייתון לשאילתות ב-SQL, שאותן שולחת למנוע SQL ואת התשובה מתרגמת לפייתון. מתקשר עם כל אחד מהמנועים של SQL (שחלקם הוזכרו מעלה). דוגמה:

```
SQL :
SELECT * FROM census
WHERE sex = F

SQLAlchemy :
db.select([census]).where(census.columns.sex == 'F')
```

SQLite3 - ספרייה בפייתון שמאפשרת עבודה עם קובץ DB ללא שרת נפרד (microservice במקרה שלנו). <https://docs.python.org/3/library/sqlite3.html>

תצורת פרוייקט - כל מיקרוסרוויס הוא תיקייה, כמו גם ה-frontend וה-backend. לכל אחד מהם צריך ליצור קונטיינר משלו לכן בכל אחת מהתיקיות יהיה Dockerfile וגם טסטים. את הכל יחבר docker-compose שהוא מסוג yaml. (ניתן להיעזר באתרים לביצוע format לקובץ yaml כמו <https://jsonformatter.org/yaml-formatter>)



בREADME יש להוסיף תמונה של דיאגרמת בלוקים המתארת את כל חלקי הפרוייקט וכיצד הם מתקשרים (הייתה דוגמה בשיעור וגם בשיעור 7)

:Dataclass

Dataclass הוא class שמכוון לאחסון מידע, כלומר יותר פשוט ונוח למתכנת ליצור בו משתנים.

יצירת Dataclass דורשת ייבוא מהספרייה dataclasses. כל משתנה שניצור בו יהיה לו בנאי (__init__) אוטומטי:

```
from dataclasses import dataclass

@dataclass
class Employee:
    employeeID : int
    name: str
    designation: str
```

לעומת זאת, ב-class רגיל צריך מתודת בנאי __init__:

```
class Employee:

    def __init__(self, employeeID, name, designation):
        self.employeeID = employeeID
        self.name = name
        self.designation = designation
```

Pydantic מזכיר בצורתו Dataclass.

:Decorator

מתודה שמקבלת מתודה אחרת, "עוטפת" אותה, מוסיפה לה פונקציונליות ומחזירה אותה. נשתמש בסימן @ ואחריו שם ה-decorator כדי לגרום למתודה להיות עטופה על ידי.

דוגמה:

```
>>> def make_pretty(func): #This is the decorator
>>>     def inner():
>>>         print("I got decorated")
>>>         func()
>>>     return inner

>>> @make_pretty
>>> def ordinary(): #This is a regular function that's going to be decorated
>>>     print("I am ordinary")

>>> ordinary()
I got decorated
I am ordinary
```

כל הפונקציות שהשתמשנו בהן ב-fastAPI עטופות ע"י decorators (@app.get וכו')

ניתן גם ליצור decorator מסוג class.

מתודת __call__: מתודה ב-class שמגדירה מה יקרה כשיקראו לו, לדוגמה MyClass()

:Generator

פונקציה שמחזירה אובייקט שניתן לבצע עליו איטרציות, כלומר אובייקט שניתן לקחת כל פעם ערך אחד שלו ולבצע עליו פעולה כלשהי. לדוגמה- לולאה על רשימה של 10 מספרים.

כל פונקציה שמכילה את הפעולה yield לפחות פעם אחת היא generator. yield היא פעולה כמו return, רק שהיא לא מסיימת את פעילות הפונקציה (כמו return) אלא רק עוצרת אותו ושומרת את המצב הקיים עד שניתן להמשיך.

דוגמה:

```
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n

for item in my_gen():
    print(item)
```

ומה שמוחזר:

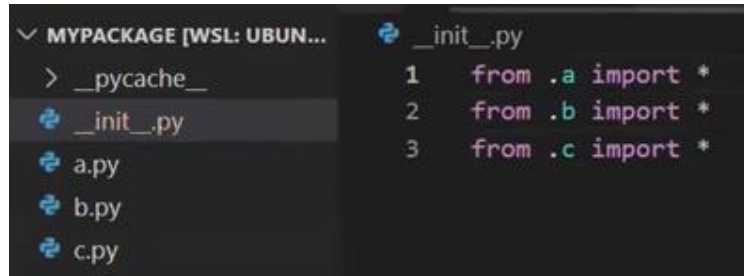
```
This is printed first
1
This is printed second
2
This is printed at last
3
```

ניתן לראות את פעולת השמירה של my_gen בין הדפסה להדפסה גם בעזרת פונקציית next בדוגמה הבאה:

```
>>> a = my_gen()
>>> next(a)
This is printed first
1
>>> next(a) #Value 1 is saved while generator is paused.
This is printed second
2
```

Package - מודל פייתוני שיכול להכיל מודלים אחרים ואף packages אחרים. את ה-package צריך להתקין לוקלית (עם pip install) ואז לייבא בעזרת import בקובץ פייתון שבו נרצה אותו.

על מנת ליצור package עליו להכיל קובץ `__init__.py`, קובץ זה יכול להיות ריק או להכיל קוד. בכל תיקייה נוספת שניצור בתוך ה-package צריך להיות עוד קובץ `__init__`. ניתן לייבא את פונקציונליות שיש בקבצי `py` אחרים שכתבנו בpackages לקובץ ה-`__init__` (ניתן לעשות את זה באופן כללי- לייבא פונקציונליות מקובץ `py` אחד לאחר) בצורה הבאה:



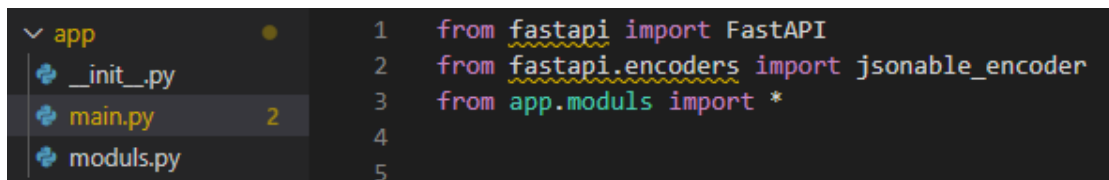
```

MYPACKAGE [WSL: UBUN...
  > __pycache__
  > __init__.py
  > a.py
  > b.py
  > c.py
__init__.py
1  from .a import *
2  from .b import *
3  from .c import *

```

הכוכבית מסמנת שאנו רוצים לייבא את כל הפונקציונליות של הקובץ. ניתן לבחור גם משתנים, פונקציות או class ספציפיים במקום כוכבית.

דוגמה נוספת:



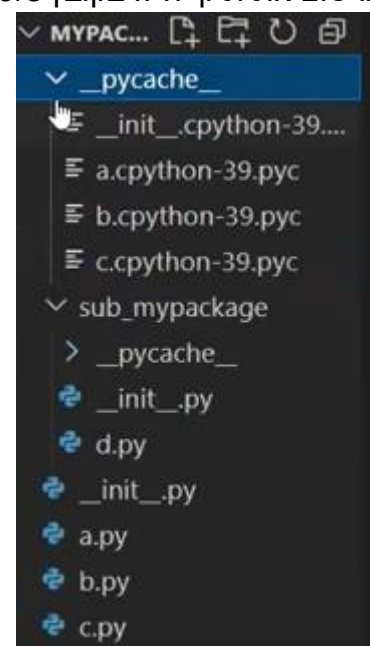
```

app
  > __init__.py
  > main.py
  > modules.py
main.py
1  from fastapi import FastAPI
2  from fastapi.encoders import jsonable_encoder
3  from app.modules import *
4
5

```

בקובץ modules יצרנו את ה-classes של pydantic. לאחר מכן ייבאנו אותן לקובץ main כדי שנוכל להשתמש בהם. (ניתן גם היה לרשום modules. במקום app.modules מכיוון שהmain הוא גם בתיקיית app)

__pycache__ - תיקייה שנוצרת שבה נשמרים דברים שנמצאים בcache. לדוגמה- אם נבצע import למשהו הוא יישמר שם כחלק משימוש אוטומטי יעיל בקוד. נרשום את תיקייה זו בקובץ gitignore כשנעלה את הקוד ל-github.



```

MYPAC...
  > __pycache__
  > __init__.cpython-39...
  > a.cpython-39.pyc
  > b.cpython-39.pyc
  > c.cpython-39.pyc
  > sub_mypackage
    > __pycache__
    > __init__.py
    > d.py
    > __init__.py
    > a.py
    > b.py
    > c.py

```

Context manager - אמצעי לניהול משאבים כמו קבצים ו-DB בתוך הקוד. לדוגמה - ניתן באמצעותו לקרוא קובץ txt. בתוך הקוד. כאשר נממש את class FileManager חייב לממש בה את שתי הפונקציות: `__enter__` ואת `__exit__`. נשתמש במילה השמורה `with` בקריאה לclass הזה פונקציה `__enter__` תופעל אוטומטית ובסיום השימוש פונקציה `__exit__` תופעל אוטומטית.

דוגמה למימוש פשוט -

```
class FileManager:
    def __init__(self, filename):
        self.filename = filename

    def __enter__(self):
        self.file = open(self.filename)
        return self.file

    def __exit__(self, type, value, traceback):
        self.file.close()

with FileManager('sample.txt') as f:
    for line in f:
        print(line)
```

תכנות פונקציונלי - סגנון כתיבת קוד בדומה לצורת כתיבת ביטוי מתמטי, כלומר הקוד מוצג כהרכבה של פונקציות. כדי ליישם זאת נשתמש בפונקציות `high-order` ו-`Lambda`. פונקציות `high-order` הן פונקציות שמקבלות כקלט או מחזירות כפלט פונקציה אחרת (או פונקציות אחרות). פונקציית `Lambda` היא פונקציה אנונימית שמועברת כביטוי לפונקציה אחרת. דוגמה לבניית מערך של ערכים ריבועיים ללא תכנות פונקציונלי -

```
numbers = [1, 2, 3, 4]
squared_numbers = []
for n in numbers:
    squared_numbers.append(n**2)
print(squared_numbers)
```

ומימוש של הפעולה באמצעות תכנות פונקציונלי -

```
numbers = [1, 2, 3, 4]
squared_numbers = map(lambda x: x**2, numbers)
print(list(squared_numbers))
```

Map - פונקציה שמורה שמקבלת פונקציה ומבנה איטרבילי (כמו רשימה, dictionary, מערך וכו'), ואז מבצעת את הפונקציה על כל אחד מהערכים במבנה האיטרבילי.

```
def myfunc(n):
    return len(n)
```

```
x = map(myfunc, ('apple', 'banana', 'cherry'))
```

ערכו של `x` בסוף הפעולה `[5, 6, 6]`.

Pandas - ספרייה שנועדה לעיבוד וניתוח נתונים. מאפשרת עבודה עם מבני נתונים כמו `dataframe` (טבלה) ו-`series` (סדרה).

Docker compose - מאפשר חיבור של כמה microservices באמצעות קובץ אחד מסוג .yaml.

נבצע את הבנייה של imagen באמצעות פקודה docker-compose build
(במקום docker build שעשינו עד כה)

נבצע את יצירת הקונטיינר באמצעות פקודה docker-compose up
(במקום docker run שעשינו עד כה)

דוגמה לקובץ .yaml :

```
version: "3.9" #The docker-compose version
services:
  backend:
    build: . # File location. One dot means the location is current file.
    ports:
      - "8888:80" #8888 is the port in localhost.
  redis:
    image: "redis:alpine"
```