29.3.22

EASS שיעור 7- חזרה ודגשים, המשך Redis, המשך Async IO

חזרה והערות כלליות לקראת הגשת המטלה הראשונה:

- ההגשה נדחתה לתאריך 26.04.2022
- המטלה תוגש באמצעות github וצריכה להכיל את כל הרכיבים הבסיסיים הנדרשים ובינהם:
  - ✓ README מפורט ומסודר
  - ✓ סידור של כל הקבצים בתיקיות שונות בהתאם לתפקידם
  - ✓ קובץ דרישות requirements.txt - מה צריך המשתמש להתקין כדי שיוכל להריץ את המערכת
  - ✓ קובץ בדיקות unit וסקריפט בדיקת אינטגרציה/ הסבר כיצד לבצע בדיקת אינטגרציה למערכת ב-README
  - ✓ שימוש בfastapi שמגדיר את התשובות לבקשות HTTP שונות ו-pydantic להמרה של JSON לclass פייתוני ולהפך.
- בדיקות אינטגרציה-המלצת המרצה לבנות shell_script.sh שמבצע את ההרצה של כל ה-dockerfile ושולח בקשות HTTP כדי לבחון שאכן מוחזרת התשובה הנכונה. הסקריפט בוחן שהמערכת באופן כללי עובדת כמו שצריך (לדוגמה אם אחד מה-microservices הוא בסיס נתונים אז הבקשה צריכה לגרום ל-backend לגשת לבסיס הנתונים, כדי לבחון ששניהם עובדים כראוי ביחד)
- OCR- מבצע המרה של תמונה לטקסט (deep learning)
- כיצד להשתמש בקובץ requirements.txt :

בקובץ טקסט requirements.txt נרשום את השמות של הספריות אותן נרצה שהמשתמש יתקין עם pip install. ניתן לרשום עם גרסאות או בלי. לדוגמה-

requirements.txt

```
1  ###### Requirements without Version Specifiers ######
2  fastapi
3  uvicorn[standard]
4
5  ###### Requirements with Version Specifiers ######
6  numpy==2.2.2
7  pandas>= 1.4.0
```

הפקודה בdockerfile שמריצה התקנה לכל מה שיש בקובץ requirements.txt-

RUN pip install -r requirements.txt

```
RUN pip install -r requirements.txt
```

הערה- לכל microservice יהיה קובץ requirements.txt משלו.

29.3.22

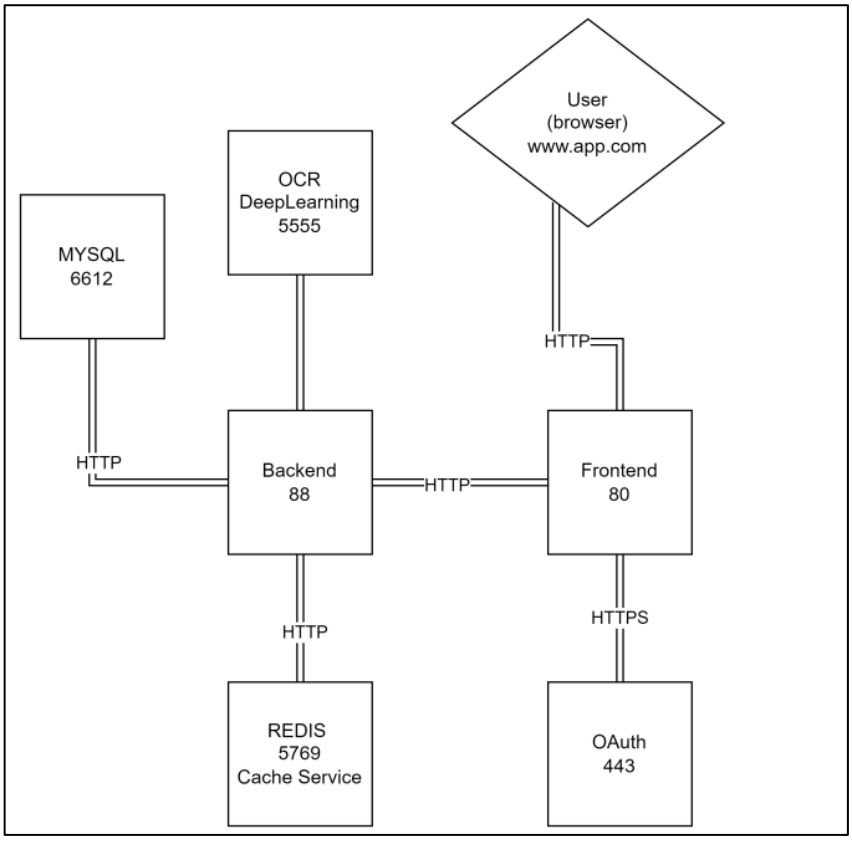
<u>הדגמה למבנה מערכת בעזרת דיאגרמת בלוקים:</u>

כל ריבוע מהווה microservice שעבורו נייצור Dockerfile נפרד שעובד עם port שונה. נציין שרק ה-port של frontend הוא חשוף ללקוח, כל השאר הם פנימיים למערכת.

ל- Dockerfile לא בהכרח יש קובץ קוד.
אם ב- Dockerfile יש קטע קוד fastapi , צריך שיהיה לו גם תיקייה משלו עם main, app, requirements.txt לפחות.

ה-microservices מתקשרים אחד עם השני באמצעות פרוטוקול HTTP, כלומר שולחים JSON בבקשות ובתשובות.

בפרוייקט של קורס זה עלינו לממש לפחות 3 microservices, כש- backend,frontend הם הכרחיים.



29.3.22

Redis המשך:

1. כדי שהמידע מהcontainer של redis יישמר בדיסק שלנו (host) ולא יימחק, נשתמש בvolume (-v) ונקשר תיקייה לוקלית לתיקיית data בקונטיינר (בתיעוד documentation של אימג' redis אמרו לנו מהי התיקייה שאליה אנחנו צריכים לפנות):

```
docker run --rm --name redis-container -v data_local:/data -p1234:6379 -d redis
```

2. בשלב הבא ניצור את התקשורת עבור redis:

```
docker network create --subnet 172.20.0.0/16 --ip-range 172.20.240.0/20 redis-demo-network
```

3. נחבר בין הקונטיינר של redis לתקשורת שיצרנו:

```
docker network connect --ip=172.20.0.1 redis-demo-network redis-container
```

4. נריץ קונטיינר פייתון שמחובר לתקשורת של redis , וכל מידע שנגדיר יישמר גם אם הקונטיינר יסתיים:

```
docker run -it --network redis-demo-network --rm python:3.9 bash
```

לדוגמה, אם בטרמינל פייתון שייפתח נרשום:

```
>>> import redis
>>> r = redis.Redis(host='172.20.0.1', port=6379, db=0)
>>> r.set('foo','bar')
```

(כלומר, אם נבצע r.get('foo') נקבל b'bar')

אז גם אם הקונטיינר יסתיים ואז נפתח אותו שוב ונבצע רק r.get('foo') נקבל תשובה b'bar' מבלי שעשינו שוב set, כי המידע שהוגדר בset לפניכן כבר שמור.

Async IO המשך:

- Event loop – מבנה שדוגם את האירועים שממתינים לתשובה מהם. בfastapi יש event loop מובנה אוטומטית.

אנו נתייחס למושגים tread, process, GIL לפי המשמעות שלהם בסביבת פייתון.

מאמר שמבצע השוואה והסבר על דרכי הפעולה של process, multithreading, asyncio:
https://medium.com/analytics-vidhya/asyncio-threading-and-multiprocessing-in-python-4f5ff6ca75e8

- GIL- Global Interpreter Lock, מערכת שמאפשרת רק ל-tread אחד לרוץ בזמן מסוים (כלומר, לא מאפשר פעילות מקבילה של כמה threads). מערכת זו משפיעה כשמריצים קוד שהוא multi-threaded.
- Process- תהליך, קטע קוד שרץ. בסביבת פייתון ייתכן מצב של multi-processing.
- Thread- תהליכון, חלק מ-process מסוים. אם יש כמה threads בתהליך אחד, יש לכל אחד מהם זיכרון פנימי אישי וזיכרון משותף לכולם. בסביבת פייתון לא ניתן שיותר מ-thread אחד ירוץ במקביל.