SkillFactory

10. Docker Compose. Практика

В этой части мы будем практиковаться управлять сервисами, которые писали в практике по *RabbitMQ* (*features.py*, *metric.py*, *model.py*).

Перед тем, как мы приступим к изучению *Docker Compose*, проверьте, установлен ли он у вас, введя в терминале или командной строке:

```
docker-compose version
```

→ Если нет, то установите его с помощью официального руководства.

Во втором модуле нашего курса мы запускали очередь *RabbitMQ* и ещё три сервиса на *Python*. Как вы помните, мы рекомендовали использовать *docker* для запуска с помощью команды:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

11:22

Создайте файл *docker-compose.yml* с помощью любого текстового редактора и напишите в нём:

```
version: '3.7'
services:
  rabbitmq:
      image: rabbitmq:3-management
      container_name: rabbitmq
      hostname: rabbitmq
      restart: always
      ports:
        - 5672:5672
        - 15672:15672
```

Обратите внимание! В *docker-compose.yml* важны отступы.

Сравните написанное вами со строчкой выше. Много сходства, не так ли?

Теперь обсудим, что же мы написали:

- version: '3.7' — задали версию *docker-compose*;
- services — указали на начало блока, в котором будут описаны все наших сервисы;
- Rabbitmq — задали имя сервиса *rabbitmq*;
- Image — указали его образ;
- restart: always — указали, что в случае падения контейнер должен перезапускаться автоматически.

Примечание. Можно задать и другие условия перезапуска:

restart: "no" — дефолтная опция: контейнер не будет перезапускаться ни при каких обстоятельствах.

> restart: on-failure — если запуск контейнера завершился ошибкой.

И, наконец, мы пробросили порты с помощью ports.

А ЗАЧЕМ МЫ УКАЗЫВАЕМ ИМЯ ХОСТА И ИМЯ КОНТЕЙНЕРА?

Дело в том, что при работе с *docker-compose* создается собственная сеть, и чтобы постучаться из одного сервис в другой, нужно будет использовать имя хоста, а не стандартный *localhost* (хотя это тоже возможно — см. ссылку на официальную документацию).

РАБОТАЕМ С СЕРВИСАМИ

Пришло время поработать с нашими сервисами! Начнём с небольшой модернизации файлов.

→ FEATURES.PY

Заменим адрес очереди *localhost* на *'rabbitmq'* (как мы и указали в *compose*-файле).

Также добавим **бесконечный цикл**, чтобы признаки и правильные ответы отправлялись постоянно.

```
import pika
import json
import numpy as np
import time
from sklearn.datasets import load_diabetes

X, y = load_diabetes(return_X_y=True)

while True:
    try:
        random_row = np.random.randint(0, X.shape[0]-1)

        connection =
pika.BlockingConnection(pika.ConnectionParameters('rabbitmq'))
        channel = connection.channel()

        channel.queue_declare(queue='Features')
        channel.queue_declare(queue='y_true')

        channel.basic_publish(exchange='',
                              routing_key='Features',
                              body=json.dumps(list(X[random_row])))
        print('Сообщение с вектором признаков, отправлено в очередь')

        channel.basic_publish(exchange='',
                              routing_key='y_true',
                              body=json.dumps(y[random_row]))

        print('Сообщение с правильным ответом, отправлено в очередь')
        connection.close()
        time.sleep(2)
    except:
        print('Не удалось подключиться к очереди')
```

→ METRIC.PY И MODEL.PY

В файлах *metric.py* и *model.py* заменим только адрес подключения к очереди (нужно отметить, что обычно такие переменные задаются с помощью **переменных окружения**) и обернём подключение к очереди в try ... except.

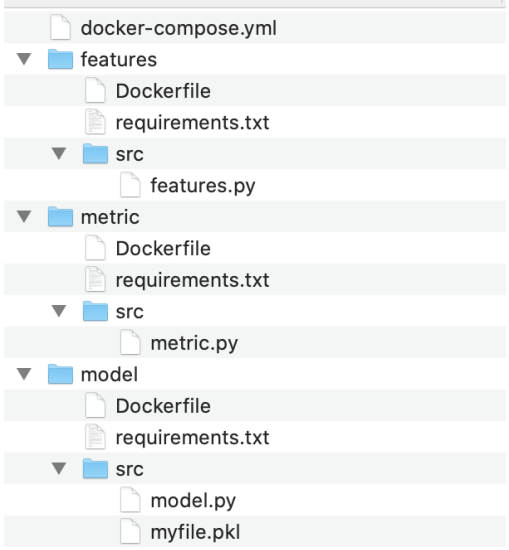
```
"""model.py"""
import pika
import json
import pickle
import numpy as np

with open('myfile.pkl', 'rb') as pkl_file:
    regressor = pickle.load(pkl_file)

try:
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host='rabbitmq'))
    channel = connection.channel()

    channel.queue_declare(queue='Features')
    channel.queue_declare(queue='y_predict')

    def callback(ch, method, properties, body):
        print(f'Получен вектор признаков {body}')
        features = json.loads(body)
        pred = regressor.predict(np.array(features).reshape(1, -1))

        channel.basic_publish(exchange='',
                              routing_key='y_predict',
                              body=json.dumps(pred[0]))
        print(f'Предсказание {pred[0]} отправлено в очередь y_predict')


    channel.basic_consume(
        queue='Features', on_message_callback=callback, auto_ack=True)

    print('...Ожидание сообщений, для выхода нажмите CTRL+C')
    channel.start_consuming()

except:
    print('Не удалось подключиться к очереди')
```

```
"""metric.py"""
import pika
import json

try:
    connection = pika.BlockingConnection(
      pika.ConnectionParameters(host='rabbitmq'))
    channel = connection.channel()


    channel.queue_declare(queue='y_true')
    channel.queue_declare(queue='y_predict')

    def callback(ch, method, properties, body):
        print(f'Из очереди {method.routing_key} получено значение
{json.loads(body)}')


    channel.basic_consume(
        queue='y_predict', on_message_callback=callback, auto_ack=True)

    channel.basic_consume(
        queue='y_true', on_message_callback=callback, auto_ack=True)

    print('...Ожидание сообщений, для выхода нажмите CTRL+C')
    channel.start_consuming()

except:
  print('Не удалось подключиться к очереди')
```

→ ОБРАЗЫ

Следующим шагом подготовьте образы для каждого из наших сервисов и сложите их в разные директории рядом с *docker-compose.yml*.

У вас должна получится примерно такая структура директорий:



Снова откроем файл *docker-compose* и добавим в него описание сервиса *features*:

```
version: '3.7'
services:
 rabbitmq:
      image: rabbitmq:3-management
      container_name: rabbitmq
      hostname: rabbitmq
      restart: always
      ports:
         - 5672:5672
         - 15672:15672

 features:
      build:
         context: ./features
      restart: always
      depends_on:
           - rabbitmq
```

Директива build указывает на то, что образ требуется собрать, при этом context указывает путь на размещение *dockerfile*. Директива depends_on указывает на зависимость от других сервисов, и означает, что *compose* не будет запускать сервис *features* без запущенного *rabbitmq*.

СДЕЛАЙТЕ САМИ!

Добавьте самостоятельно еще два сервиса: *model* и *metrics*.

Когда все будет готово, перейдите через командную строку в директорию со своим проектом и запустите docker-compose up.

Вы увидите, что все сервисы запустились. Также вы сможете наблюдать лог каждого сервиса. После закрытия терминала сервис будет выключен. Чтобы отвязать его от терминала, используйте ключ -d.

Запустите команду docker-compose up -d, а затем команду docker ps и убедитесь, что все ваши сервисы запустились и работают.

> **Ещё полезные команды**
>
> С помощью команды docker logs <ID контейнера> можно посмотреть логи каждого сервиса.
>
> Чтобы остановить запущенные сервисы, воспользуйтесь командой docker-compose down.
>
> Иногда требуется пересобирать образы, например при изменении кода. Для этого удобно воспользоваться ключом --build. В таком случае команда запуска будет выглядеть следующим образом: docker-compose up -d --build.

ЗАДАНИЕ ДЛЯ ОТЛИЧНИКОВ

В файле *metric.py* добавьте запись лога из *callback* в файл *labels_log.txt* и разместите его в директорию *logs* рядом с вашим *compose*-файлом. Чтобы файл был доступен из локальной файловой системы, необходимо примонтировать нужную папку с помощью директивы *volumes*, которая является аналогом -v для обычного запуска.

Показать решение

Решение

Эталонное решение к заданию для отличников можно посмотреть по этой ссылке.

Продолжим!