Федеральное государственное автономное образовательное учреждение

высшего образования

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ

ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Практическая работа на тему:

«Разработка микросервисных приложений на FastAPI в docker»

Студент:

Гизбрехт В.Д

Преподаватель:

Афанасьев Максим Яковлевич

Санкт-Петербург

2025 г.

Содержание

1 Введение

В данной практической работе была исследована разработка микросервисных приложений на языке программирования Python с помощью FastAPI в docker. Полный проект можно просмотреть на моем гите: `https://github.com/WALTANN/fastapi-docker`. Разберемся с каждым термином по-отдельности:

1. Микросервис — это подход к разбиению большого монолитного приложения на отдельные приложения, специализирующиеся на конкретной функции.

2. FastAPI — это современная высокопроизводительная веб-инфраструктура, которая обладает множеством функций, таких как автоматическое документирование на основе OpenAPI и встроенная библиотека сериализации и проверки. Имеет асинхронность

3. Docker — это программное обеспечение, позволяющее упаковать приложение и все его зависимости в единый модуль.

4. Python — идеальный инструмент для создания микросервисов, поскольку у него отличное сообщество, простота обучения и множество библиотек.

Использование микросервисной архитектуры позволяет достичь гибкости и масштабируемости при разработке программных систем. Каждый микросервис работает независимо от других, что упрощает тестирование, обновление и развертывание отдельных частей приложения. Это особенно актуально для крупных проектов, где важно минимизировать время на внесение изменений и обеспечить высокую отказоустойчивость.

FastAPI, как современный фреймворк для создания веб-приложений, предоставляет удобный инструментарий для построения API с поддержкой асинхронности, что положительно сказывается на производительности. Его способность автоматически генерировать документацию на основе OpenAPI и JSON Schema значительно упрощает взаимодействие между фронтендом и бэкендом, а также работу с внешними разработчиками.

Docker играет ключевую роль в процессе разработки и развертывания микросервисов. Он позволяет изолировать каждый сервис в контейнер, обеспечивая одинаковое поведение приложения на разных платформах. Это решает проблему "у меня на машине работало так как окружение полностью определяется конфигурацией

Docker-образа. Кроме того, Docker упрощает управление зависимостями и совместное использование сервисов внутри распределенной системы.

Python, благодаря своей простоте и читаемости кода, а также богатой экосистеме библиотек, становится отличным выбором для реализации микросервисов. Быстрая разработка, наличие мощных инструментов для работы с сетью и данными, а также активное сообщество делают его одним из самых популярных языков в сфере backend-разработки.

В рамках данной практической работы были рассмотрены основные принципы проектирования микросервисной архитектуры, реализованы примеры сервисов на Python с использованием FastAPI, а также выполнено их контейнеризирование с помощью Docker. Полученные навыки позволяют эффективно применять современные подходы к разработке распределенных приложений и готовят к решению реальных задач в профессиональной деятельности.

2 Создание простого микросервиса

В этой главе разберем создание простого микросервиса на python + FastAPI и соберем его в docker. Рассмотрим простейший проект, структура которого выглядит примерно так:

```
project/
 main.py
 requirements.txt
 Dockerfile
```

main.py — FastAPI приложение

Это основной файл приложения на Python, реализующий простой REST API с двумя маршрутами.

```python
from fastapi import FastAPI

app = FastAPI()

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

@app.get("/ping")
def ping():
    return {"message": "pong"}
```

Что делает этот код?

1. Создаёт экземпляр FastAPI (‘app = FastAPI()‘).
2. Определяет два маршрута:
   (a) GET ‘/‘ — возвращает JSON "Hello": "World".
   (b) GET ‘/ping‘ — возвращает JSON "message": "pong".

requirements.txt — зависимости

Файл содержит список библиотек, необходимых для работы приложения.

```
fastapi
uvicorn
```

Dockerfile — инструкция по сборке образа

Dockerfile описывает шаги по созданию Docker-образа, включающего Python-приложение.

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . /app

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Команды для сборки и запуска контейнера

Переходим в папку проекта и выполняем следующие команды:

```
docker build -t project .
docker run -p 8000:8000 project
```

Доступ

После этого приложение будет доступно по адресам:

- `http://localhost:8000`
- `http://localhost:8000/ping`
- `http://localhost:8000/docs` (Swagger UI)

Пояснение

1. **FastAPI** — фреймворк для создания REST API на Python. Он автоматически генерирует документацию (Swagger и ReDoc), поддерживает асинхронность и имеет мощную систему валидации данных.

2. **Маршруты (routes)** — это точки входа в ваше API. Они позволяют клиентам выполнять запросы и получать данные или выполнять действия.

3. **main.py** — точка входа в приложение, где создаётся экземпляр FastAPI и регистрируются маршруты.

4. **Dockerfile** — текстовый файл, содержащий инструкции для сборки Docker-образа. Это позволяет упаковать всё приложение со всеми зависимостями в один контейнер.

5. **docker build** — команда создаёт Docker-образ на основе Dockerfile.

6. **docker run** — запускает контейнер из созданного образа.

Такой подход удобен для тестирования, разработки и дальнейшего развёртывания (деплоя) в production.

3 Проделанная работа

Теперь разберемся с проектом, архитектура которого была представлена в методическом пособии к практической работе. Проект состоит из нескольких микросервисов, каждый из которых реализован как отдельное FastAPI-приложение внутри своего контейнера.

Дерево каталогов

```
my_project/
 docker-compose.yml
 auth/
  app/
   main.py
   routers/
    example.py
  requirements.txt
  .env
 users/
  app/
   main.py
   routers/
    example.py
  requirements.txt
  .env
```

Каждый сервис:

- Имеет собственную точку входа ‘main.py‘.
- Содержит маршруты в папке ‘routers/‘.
- Определяет зависимости в ‘requirements.txt‘.
- Хранит настройки порта в ‘.env‘.

При работе с docker-compose.yml возникла проблема, связанная с использованием параметра inline в секции build. В предоставленном шаблоне docker-compose.yml предполагалось описывать инструкции сборки Docker-образов прямо внутри файла с помощью ключа inline, чтобы не создавать отдельные Dockerfile для каждого сервиса.

Однако при попытке запуска проекта с помощью команды ‘docker-compose up –build‘ была получена ошибка:

```
ERROR: The Compose file './docker-compose.yml' is invalid because:
services.auth.build contains unsupported option: 'inline'
```

Это произошло из-за того, что поддержка inline появилась только в новых версиях Docker Compose (начиная с версии 2.23+), а на моей системе установлена более старая версия 1.29.2, в которой такой способ описания Dockerfile не поддерживается. У меня не получилось установить более новую версию docker, в связи с чем было принято решение уйти от inline путем написания dockerfile отдельно для auth и users, при этом оставив docker-compose, в который были направлены dockerfile’ы через указания путей к ним.

Дальнейшнее развитие 1

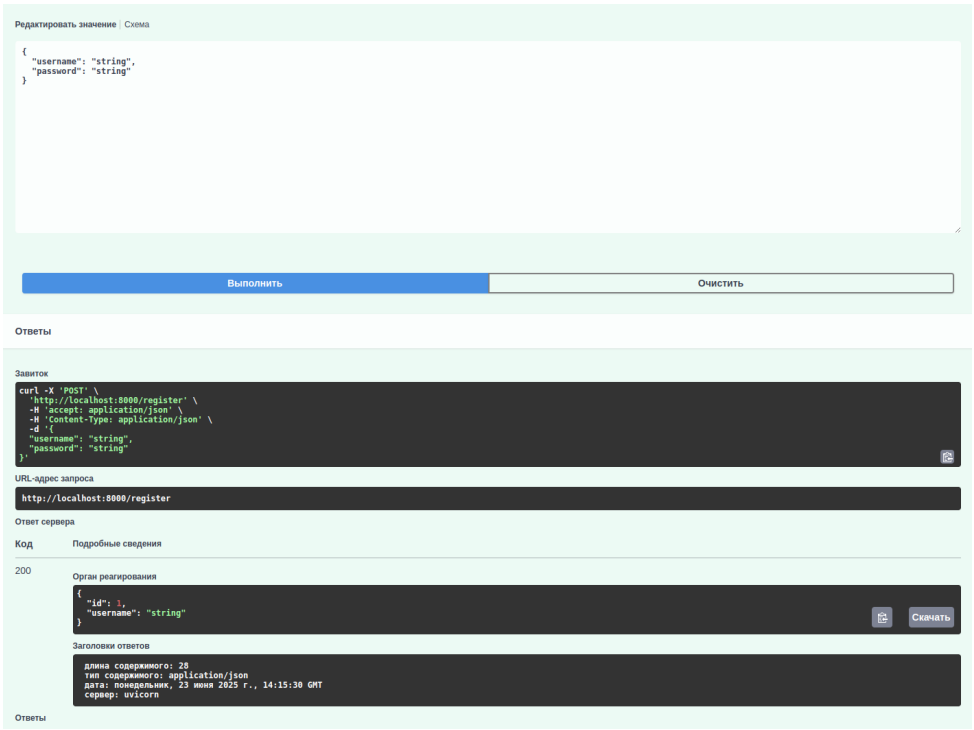
Были добавлены models и проверены POST-запросы, далее прикреплен скриншот, на котором видно рабочий POST-запрос в виде "регистрации"через маршрут `http://localhost:8000/register`
Тело запроса:

```
{
  "username": "string",
  "password": "string"
}
```

Тело ответа:

```
{
  "id": 1,
  "username": "string"
}
```

Это означает, что сервис обрабатывает POST-запрос корректно по коду.



Дальнейшнее развитие 2

Теперь добавим еще один сервис, который хранит в себе просто какие-то данные. Кроме того был добавлен volumes, так как он даёт возможность монтировать локальные файлы внутрь контейнера , что особенно удобно на этапе разработки. Это позволяет редактировать код в привычном редакторе (например, VS Code), и сразу видеть изменения внутри запущенного контейнера — без необходимости пересборки образа после каждого правки. Конечная архитектура проекта выглядит так:

```
my_project/
 docker-compose.yml
 auth/
  app/
   main.py
   routers/
    auth_router.py
  requirements.txt
  .env
 users/
  app/
   main.py
```

```
   routers/
    users_router.py
  requirements.txt
  .env
 zprod/
  app/
   main.py
   routers/
    zprod_router.py
  requirements.txt
  .env
```

Как и говорилось ранее, весь проект можно скачать и посмотреть на гитхаб, для того чтобы его запустить, стоит сделать билд:

```
docker-compose up --build
```

После билда будут доступны такие основные руты:

1. ‘Auth‘ - `http://localhost:8000/docs` (swagger)
2. `http://localhost:8000/ring`
3. Users‘ - `http://localhost:8001/docs` (swagger)
4. `http://localhost:8001/king`
5. ‘Prod‘ - `http://localhost:8002/docs` (swagger)
6. `http://localhost:8002/hello`

С помощью swagger (рут docs) можно будет опробовать POST-запросы и увидеть рабочее состояние сервисов

4 Вывод

В данном отчете приведены мои действия при выполнении практической работы на тему "Разработка микросервисных приложений на FastAPI в docker". При выполнении работы было изучено написание микросервисных приложений на языке программирования Python с использованием FastAPI с дальнейшим контейнезированием в docker-образ и запуском из образа.

Изначально во 2 главе было приведено написание простейшего микросервиса, учитывая все вышесказанные действия, с которого началось мое изучение такой структуры программирования. В данном разделе отчета приведен полный код для python, dockerfile, а также описаные действия и структура проекта для попытки запуска для понимания работы на простом примере.

В следующей части проектной работы был представлен конечный проект, имеющий 3 сервиса, а также скриншот с работой сервиса и ответом на POST-запрос.

В ходе выполнения практической работы я научился писать микросервисные приложения на языке программирования Python с использованием фреймворка FastAPI, а также упаковывать их в Docker-образы и запускать в контейнерах с помощью docker-compose.

Эти навыки являются чрезвычайно полезными как для робототехника, так и для разработчика, поскольку позволяют создавать масштабируемые, изолированные сервисы, которые легко тестируются, переносятся между средами и интегрируются в более сложные системы.

Контейнеризация обеспечивает стабильность окружения, что особенно важно при разработке распределённых систем — теперь я понимаю, как можно разделять функционал приложения на отдельные модули, взаимодействующие между собой через API, а также как организовать удобную и повторяемую среду разработки.

Кроме того, я освоил работу с volumes, что позволяет значительно ускорить процесс разработки за счёт автоматического обновления кода внутри контейнера без его пересборки.

Также я получил опыт работы с базовыми средствами микросервисной архитектуры: маршрутизацией, зависимостями, переменными окружения и управлением несколькими сервисами через docker-compose.yml. Эта практика дала мне прочный фундамент для создания более сложных и реальных проектов, имеющих более серьезные масштабы. К сожалению, у меня не получилось уйти от dockerfile'ов в связи

с версией докера, но использование docker-compose присутствует

Полный проект можно просмотреть на моем гите: `https://github.com/WALTANN/fastapi-docker`