Практическая работа по микросервисной архитектуре

Содержание

1.Введение

Цель работы - создание двух связанных между собой микросервисов и API их взаимодействия (Рис. 1):

- **`Marketplace`** – минималистичное веб-приложение, отображающее список книг, продаваемых на сайте.
- **`Recommendations`** – микросервис, предоставляющий список книг, которые могут быть интересны конкретному пользователю.

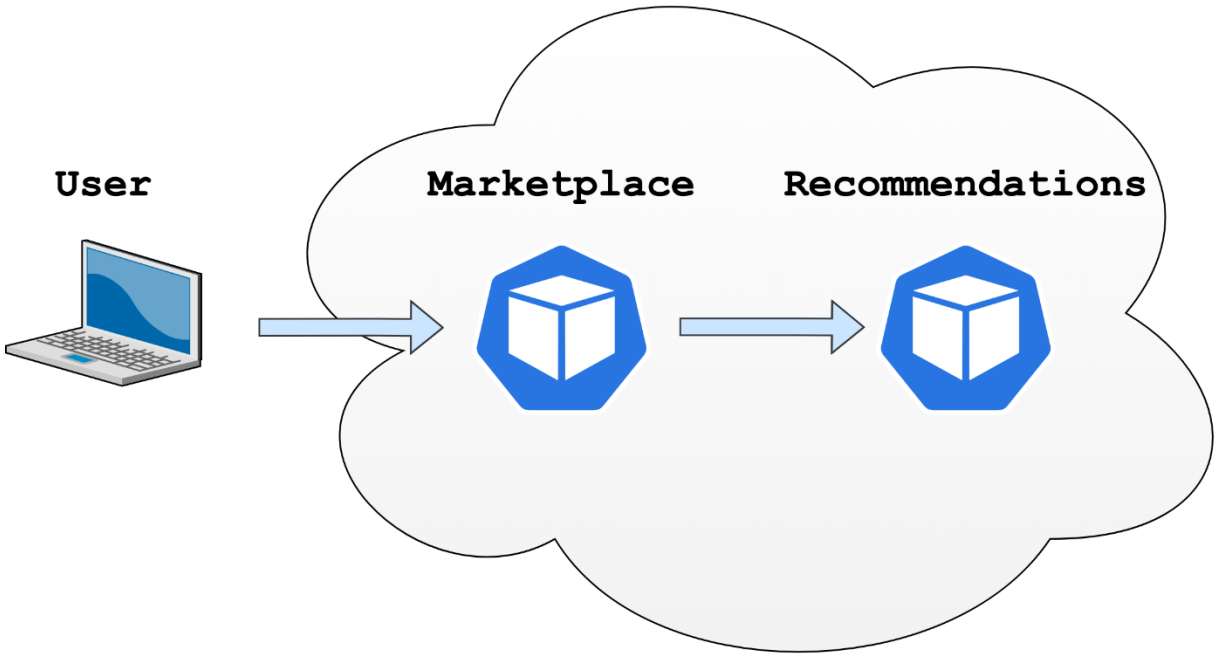


Рисунок 1.

Пользователь взаимодействует с микросервисом **`Marketplace`** через свой браузер, а уже микросервис **`Marketplace`** взаимодействует с микросервисом **`Recommendations`**.

Оба сервиса реализуем в виде сервисов Python, упакованные в контейнеры Docker и управляемые через Docker-compose.

Для реализации работы необходимо предварительно:

1) *Скачать и установить Python последней версии (не менее 3.6)*
2) *Скачать и установить Docker последней версии.*

2.Проектирование API

2.1.Спроектируем API рекомендаций.

Необходимо, чтобы запрос рекомендаций имел несколько параметров:

- **`User ID`**: мы хотим использовать id пользователя для персонализации рекомендаций.
- **`Book category`**: категории книг, по которым будет определяться рекомендация.
- **`Max results`**: ограничение на количество результатов (рекомендованных книг).

Для каждой книги будут предоставлены следующие данные:

- **`Book ID`**: уникальный числовой идентификатор книги.
- **`Book title`**: название, которое мы показываем пользователю.

Теперь определяются параметры API более формально, с помощью синтаксиса Protocol Buffers (Protobuf):

```
syntax = "proto3";

enum BookCategory {
    MYSTERY = 0;
    SCIENCE_FICTION = 1;
    SELF_HELP = 2;
}

message RecommendationRequest {
    int32 user_id = 1;
    BookCategory category = 2;
    int32 max_results = 3;
}

message BookRecommendation {
    int32 id = 1;
    string title = 2;
}

message RecommendationResponse {
    repeated BookRecommendation recommendations = 1;
}

service Recommendations {
    rpc Recommend (RecommendationRequest) returns (RecommendationResponse);
}
```

Протокол Protobuf разработан в Google и позволяет формально определять API. С помощью Protobuf специфицируется:

- Версия используемого синтаксиса **`proto3`**.
- Категории книг (**`MYSTERY`**, **`SCIENCE_FICTION`**, **`SELF_HELP`**), каждой категории присваивается числовой идентификатор.
- Определяется структура API-запроса **`RecommendationRequest`**. Используется 32-битное целое число (**`int32`**) для полей **`user_ID`** и **`max_results`**. Указывается также определенное выше перечисление **`BookCategory`**. Кроме имени каждому полю назначается числовой идентификатор поля.
- Определяется формат ответа **`BookRecommendation`** для рекомендации книги. Он имеет 32-битный целочисленный идентификатор книги и текстовое поле - название книги.
- Описывается ответ микросервиса **`RecommendationResponse`**, как несколько повторяющихся значений **`BookRecommendation`**.
- Последние строки определяют API-метода **`Recommend`**, который принимает **`RecommendationRequest`** и возвращает **`RecommendationResponse`**. Ключевое слово **`rpc`** обозначает удаленный вызов процедуры.

3.Создание микросервиса `Recommendations`

3.1. Сначала создадим рабочую папку WebService и определим структуру каталогов в ней:

```
.
├── protobufs/
│   └── recommendations.proto
└── recommendations/
```

3.2. Каталог **`protobufs/`** будет содержать файл с именем **`recommendations.proto`**. Содержимое этого файла необходимо заполнить приведенным в предыдущем разделе кодом.

3.3. По сформированному описанию API сгенерируем код Python для взаимодействия с ним в каталоге **`recommendations/`**.

3.4. Во-первых, мы должны установить **`grpcio-tools`**. Создадим файл **`recommendations/requirements.txt`** и добавим следующее:

```
grpcio-tools ~= 1.30
```

3.5. Чтобы запустить код локально запустим терминал командной строки **`cmd`**. Дальнейшие действия выполняем в нем, перейдя в рабочую папку проекта (вместо **«C:\WebService»** укажите полный путь в рабочей папке проекта).

```
C:\> cd c:\WebService
```

3.6. Активируем новую виртуальную среду

```
C:\WebService> python -m venv venv
C:\WebService> venv\Scripts\activate.bat
```

3.7. установим в нее зависимости:

```
(venv) C:\WebService> python -m pip install -r recommendations/requirements.txt
```

3.8. Сгенерируем код Python для API из **`protobufs`**, выполнив следующую команду:

```
(venv) C:\WebService> cd recommendations
python -m grpc_tools.protoc -I ../protobufs --python_out=. --grpc_python_out=.
../protobufs/recommendations.proto
```

- **python -m grpc_tools.protoc** запускает компилятор, который генерирует код Python из кода **protobuf**,
- **-I ../protobufs** сообщает компилятору, где найти файлы, которые импортирует код **protobuf**,
- **--python_out =. --grpc_python_out =.** сообщает компилятору, куда выводить файлы Python,
- **../protobufs/recommendations.proto** – это путь к файлу **protobuf**, который будет использоваться для генерации кода Python.

В результате сгенерировано два файла:

```
(venv) C:\WebService\recommendations> dir

    recommendations_pb2.py
    recommendations_pb2_grpc.py
```

Эти файлы включают типы и функции Python для взаимодействия с описанным ранее API. Компилятор сгенерирует клиентский код для вызова RPC и серверный код для реализации RPC. Сначала рассмотрим, что происходит на стороне клиента.

Создание сервера RPC.

Начнем с инструкций импорта и добавления некоторых образцов данных.

3.9. Создадим файл **recommendations.py** с кодом сервиса рекомендаций в подпапке проекта **recommendations/** и сохраним в нем следующее содержание (можно воспользоваться текстовым редактором notepad):

```
    # recommendations/recommendations.py

    from concurrent import futures
    import random


    import grpc

    from recommendations_pb2 import (
        BookCategory,
        BookRecommendation,
        RecommendationResponse,
    )

    import recommendations_pb2_grpc

    books_by_category = {
        BookCategory.MYSTERY: [
```

```
            BookRecommendation(id=1, title="Мальтийский сокол"),
            BookRecommendation(id=2, title="Убийство в Восточном экспрессе"),
            BookRecommendation(id=3, title="Собака Баскервилей"),
            BookRecommendation(id=4, title="Автостопом по галактике"),
            BookRecommendation(id=5, title="Игра Эндера"),
        ],

        BookCategory.SCIENCE_FICTION: [
            BookRecommendation(id=6, title="Дюна"),
        ],

        BookCategory.SELF_HELP: [
            BookRecommendation(id=7, title="Семь навыков высокоэффективных людей"),
            BookRecommendation(id=8, title="Как завоёвывать друзей и оказывать
влияние на людей"),
            BookRecommendation(id=9, title="Человек в поисках смысла"),
        ],
    }
```

3.10. Далее, в том же файле создадим класс, реализующий функции микросервиса:

```
class RecommendationService(recommendations_pb2_grpc.RecommendationsServicer):

    def Recommend(self, request, context):
        if request.category not in books_by_category:
            context.abort(grpc.StatusCode.NOT_FOUND, "Category not found")

        books_for_category = books_by_category[request.category]
        num_results = min(request.max_results, len(books_for_category))
        books_to_recommend = random.sample(books_for_category, num_results)

        return RecommendationResponse(recommendations=books_to_recommend)
```

Класс **RecommendationService** – это реализация нашего микросервиса. Класс наследуется от подкласса **RecommendationServicer**. Это часть интеграции с gRPC.

3.11. Далее мы определяем метод **Recommend**.

Он должен иметь то же имя, что и RPC, который мы определяем в своем файле **protobuf**. Метод также принимает **Request** и возвращает **RecommendationResponse**, как и в определении **protobuf**. Параметр **context** позволяет установить код состояния для **response**.

Метод **abort()** для завершения запроса и устанавливается код состояния **NOT_FOUND**, если вы получаете неожиданную категорию. Поскольку gRPC построен поверх HTTP/2, код состояния аналогичен стандартному коду состояния HTTP. Его установка позволяет клиенту выполнять различные действия в зависимости от полученного кода. Это также помогает промежуточному программному обеспечению (например, системам мониторинга), регистрировать, какое число запросов содержит ошибки.

В следующих строчках случайным образом выбираются книги из выбранной категории, которые можно рекомендовать. Количество рекомендаций ограничено **max_results**.

В последней строке возвращается **RecommendationResponse** со списком рекомендаций книг.

Класс **RecommendationService** уже определяет реализацию микросервиса, но нам как-то нужно запустить сам сервис.

3.12. Для этого мы определим функцию **serve()** в том же файле:

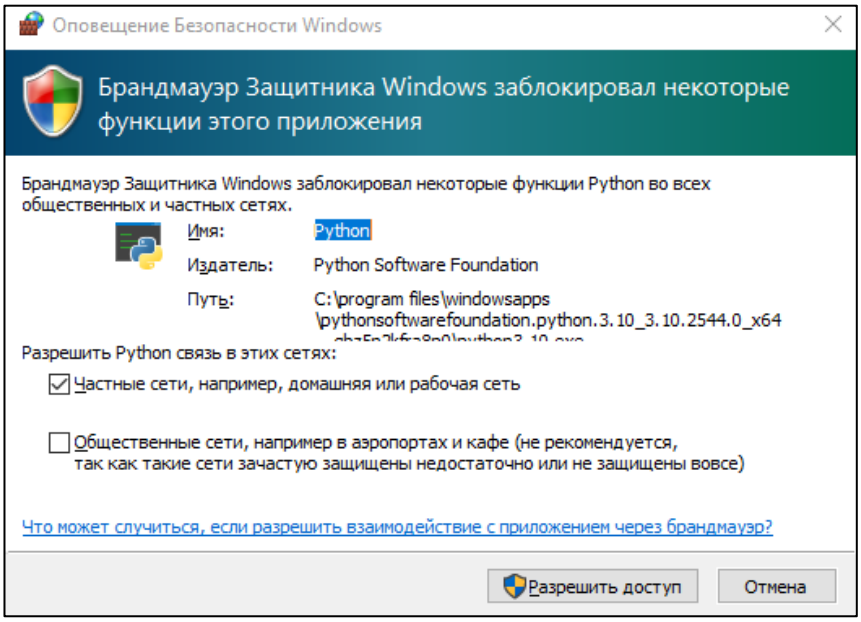
```
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    recommendations_pb2_grpc.add_RecommendationsServicer_to_server(
        RecommendationService(), server
    )

    server.add_insecure_port("[::]:50051")
    server.start()
    server.wait_for_termination()


if __name__ == "__main__":
    serve()
```

serve() запускает сетевой сервер и использует класс микросервиса для обработки запросов. В приведенном коде сначала создается сервер gRPC, которому мы указываем использовать 10 потоков для обслуживания запросов, что является хорошим дефолтным значением для реального микросервиса Python. Далее мы связываем класс с сервером. Это похоже на добавление обработчика запросов. Потом мы указываем серверу работать на порту **50051**. Далее мы запускаем микросервис и ждем, пока он не остановится. Единственный способ остановить его в этом случае – нажать в терминале комбинацию **[Ctrl + C].**

3.13. В терминале, который мы использовали ранее запустим сервис рекомендаций:

```
(venv) C:\WebService\recommendations> python recommendations.py
```

Если операционная система выдаст предупреждение о блокировке сетевых функций, выдерите «Разрешить доступ»:



Проверка сервиса

Создадим клиента для проверки работы сервиса. Ранее сгенерированный код на API основе **protobuf** является основой для его написания.

3.14. В новом терминале (не забыв активировать venv) запустим интерактивную оболочку Python, чтобы взаимодействовать с этим кодом:

```
C:\WebService> venv\Scripts\activate.bat
(venv) C:\WebService> python

>>> from recommendations_pb2 import BookCategory, RecommendationRequest
>>> request = RecommendationRequest(user_id=1,
category=BookCategory.SCIENCE_FICTION, max_results=3)
>>> request.category
>>>1
```

Компилятор Protobuf сгенерировал проверку типов в Python, соответствующую типам в коде **protobuf**. В результате **request.category** содержит одно из членов перечисления **BookCategory.**

Созданный файл **recommendations_pb2.py** содержит определения типов. Файл **recommendations_pb2_grpc.py** содержит структуру для клиента и сервера.

3.15. Инструкции импорта, необходимые для создания клиента:

```
>>> import grpc
>>> from recommendations_pb2_grpc import RecommendationsStub
```

Происходит импорт модуля **grpc**, предоставляющего функции для настройки подключений к удаленным серверам. Затем импортируем заглушку (stub) для клиента. Это заглушка, потому как сам клиент не имеет никаких функций – он лишь обращается к удаленному серверу и возвращает результат.

Если вернуться к коду **protobuf**, в конце мы увидим раздел **service Recommendations {...}**. Компилятор Protobuf берет имя микросервиса **Recommendations**, и добавляет к нему Stub, чтобы сформировать имя клиента **RecommendationsStub**.

3.16. Теперь мы можем сделать RPC-запрос:

```
>>> channel = grpc.insecure_channel("localhost:50051")
>>> client = RecommendationsStub(channel)
>>> request = RecommendationRequest(user_id=1,
category=BookCategory.SCIENCE_FICTION, max_results=1)
>>> client.Recommend(request)

recommendations {
  id: 6
  title: "The Dune Chronicles"
}
```

Мы сделали RPC-запрос к микросервису и получили ответ. Обратите внимание, что ваш результат может отличаться, потому что рекомендации выбираются случайным образом.

Теперь, когда у нас есть сервер, мы можем реализовать микросервис **Marketplace** и заставить его вызывать микросервис рекомендаций. Можно закрыть терминал с консолью Python, но оставить терминал с включенным микросервисом **Recommendations**.

4. Создание микросервиса Marketplace

4.1. Создадим новый каталог **marketplace/** и создадим в нем файл **marketplace.py** для микросервиса **Marketplace**. Дерево каталогов должно теперь выглядеть так:

```
.
├── marketplace/
│   └── marketplace.py
├── protobufs/
│   └── recommendations.proto
└── recommendations/
    ├── recommendations.py
    ├── recommendations_pb2.py
    ├── recommendations_pb2_grpc.py
    └── requirements.txt
```

Микросервис **Marketplace** будет приложением Flask для отображения пользователю веб-страницы. Он будет вызывать микросервис **Recommendations**, чтобы получить рекомендации по книгам.

4.2. Откроем файл **marketplace/marketplace.py** и добавим следующее:

```
#marketplace/marketplace.py
import os

from flask import Flask, render_template
import grpc


from recommendations_pb2 import BookCategory, RecommendationRequest
from recommendations_pb2_grpc import RecommendationsStub


app = Flask(__name__)

recommendations_host = os.getenv("RECOMMENDATIONS_HOST", "localhost")
recommendations_channel = grpc.insecure_channel(
    f"{recommendations_host}:50051"
)
recommendations_client = RecommendationsStub(recommendations_channel)


@app.route("/")
def render_homepage():
    recommendations_request = RecommendationRequest(
        user_id=1, category=BookCategory.MYSTERY, max_results=3
    )
    recommendations_response = recommendations_client.Recommend(
        recommendations_request
    )
    return render_template(
        "homepage.html",
        recommendations=recommendations_response.recommendations,
    )

if __name__ == '__main__':

    app.run(host='0.0.0.0', port=5001)
```

Здесь мы настраиваем Flask, создаем клиент gRPC и добавляем функцию для рендеринга домашней страницы. В этом файле создается канал и заглушка gRPC для обращения к сервису рекомендаций.

4.3. Создадим файл **homepage.html** в каталоге **marketplace/templates/** и добавим следующий HTML-код:

```
<!-- homepage.html -->
<!doctype html>
<html lang="en">
<head>
    <title>Книги для вас</title>
</head>
<body>
    <h1>Книги для вас</h1>
    <ul>
    {% for book in recommendations %}
        <li>{{ book.title }}</li>
    {% endfor %}
    </ul>
</body>
```

Это прототип домашней страницы приложения. После окончания разработки она будет отображать список рекомендуемых книг, которые возвращает сервис рекомендаций.

4.4. Для запуска кода потребуются следующие зависимости, которые необходимо добавить в **marketplace/requirements.txt**:

```
flask ~= 1.1
grpcio-tools ~= 1.30
Jinja2 ~= 2.11
markupsafe==2.0.1
pytest ~= 5.4
```

Каждый микросервис будут иметь свой файл **requirements.txt**, но при желании можно использовать одну и ту же виртуальную среду для обоих. Выполним обновление виртуальной среды:

```
(venv) C:\WebService> python -m pip install -r marketplace/requirements.txt
```

4.5. Теперь, после установки зависимости, нужно также сгенерировать код для **protobufs** в каталоге **marketplace/**. Для этого выполним в консоли следующие. команды:

```
(venv) C:\WebService> cd marketplace
(venv) C:\WebService> python -m grpc_tools.protoc -I ../protobufs --python_out=. --
grpc_python_out=. ../protobufs/recommendations.proto
```

Дерево каталогов должно теперь выглядеть так:

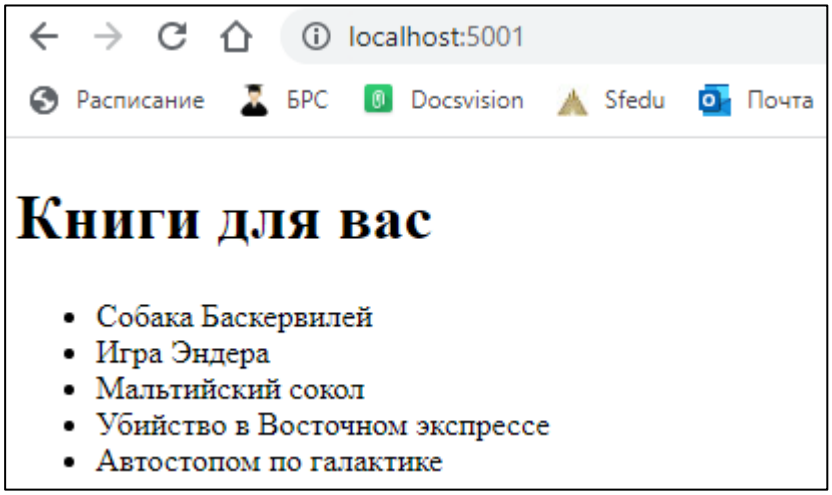
```
.
├── marketplace/
│   ├── marketplace.py
│   ├── recommendations_pb2.py
│   ├── recommendations_pb2_grpc.py
│   ├── requirements.txt
│   └── templates/
│       └── homepage.html
├── protobufs/
│   └── recommendations.proto
└── recommendations/
    ├── recommendations.py
    ├── recommendations_pb2.py
    ├── recommendations_pb2_grpc.py
    └── requirements.txt
```

4.6. Для запуска микросервиса **Marketplace**, введем в консоль следующее:

```
(venv) C:\WebService\marketplace> python marketplace.py
```

Теперь микросервисы **Recommendations** и **Marketplace** работают в двух отдельных терминалах.

В результате наших действий запущено приложение Flask, которое по умолчанию работает на порту 5001. Проверьте результат в браузере:



Создано два микросервиса, которые общаются друг с другом. Но они все еще находятся и выполняются непосредственно на компьютере. Далее рассмотрим, как внедрить их в производственную среду.

4.7. Можно остановить микросервисы Python, нажав в терминале **Ctrl + C**. Далее мы создадим контейнеры для каждого сервиса и будем запускать микросервисы в Docker.

5. Упаковка микросервисов Python в Docker

Docker – это технология, которая позволяет на одном компьютере изолировать одни процессы от других. Docker идеально подходит для развертывания микросервисов Python, поскольку можно упаковать все зависимости и запустить микросервис в изолированной среде.

Чтобы продолжить работу, необходимо убедиться, что у вас установлен Docker (его можно скачать с официального сайта).

Создадим два образа Docker: по одному для каждого из микросервисов. Образ – это по сути файловая система плюс некоторые метаданные. Каждый микросервис может записывать файлы, не затрагивая файловой системы, на которой запущен Docker, и открывать порты без конфликтов с другими процессами.

Образы создаются с помощью описания в виде **`Dockerfile`**. Весь процесс всегда строится от некоторого базового образа. В нашем случае базовый образ будет включать интерпретатор Python. Далее мы скопируем файлы из своей файловой системы в образ Docker и запустим команду для установки зависимостей.

Dockerfile для сервиса Recommendations

5.1. Начнем с создания образа Docker для микросервиса рекомендаций. Создадим файл **recommendations/Dockerfile** и добавим в него следующее:

```
FROM python:3.9-slim

RUN mkdir /service
COPY protobufs/ /service/protobufs/
COPY recommendations/ /service/recommendations/
WORKDIR /service/recommendations
RUN python -m pip install --upgrade pip
RUN python -m pip install -r requirements.txt
RUN python -m grpc_tools.protoc -I ../protobufs --python_out=. \
           --grpc_python_out=. ../protobufs/recommendations.proto

EXPOSE 50051
ENTRYPOINT [ "python", "recommendations.py" ]
```

В первой строчке мы инициализируем образ с помощью базовой среды Linux и последней версии Python. Образ **slim** на этот момент имеет типичную структуру файловой системы Linux и включает минимальный набор модулей, достаточный для работы Python. Далее создается новый каталог в **/service** для хранения кода микросервиса. Потом мы копируем каталоги **protobufs/** и **recommendations** в **/service**.

Затем мы создаем каталог **/service/recommendations** и делаем его текущей рабочей директорией. После этого мы устанавливаем зависимости для библиотек Python и генерируем Python-файлы из кода **Protobuf**. Т.е. повторяются действия, которые выполнялись для создания сервиса вручную.

В итоге папка **/service/** внутри Docker-образа будет иметь следующую структуру:

```
/service/
├── protobufs/
│   └── recommendations.proto
└── recommendations/
    ├── recommendations.py
    └── requirements.txt
```

Наконец, мы открываем микросервис наружу Docker-образа на порте 50051 и говорим Docker запустить наш микросервис в Python, передав название файла для запуска.

Мы описали всю процедуру в **Dockerfile**.

5.2. Чтобы сгенерировать Docker-образ, перейдем в терминале на уровень выше относительно **Dockerfile** и запустим команду **build**:

```
C:\WebService\recommendations>cd ..
C:\WebService>docker build . -f recommendations/Dockerfile -t recommendations
```

5.3. Когда Docker создаст образ, мы сможем его запустить:

```
C:\WebService>docker run -p 127.0.0.1:50051:50051/tcp recommendations
```

Никаких дополнительных сообщения не отображается, но микросервис рекомендаций теперь работает в Docker-контейнере. Когда мы запускаем образ, на его основе создается контейнер. Мы можем запустить образ несколько раз, чтобы развернуть несколько контейнеров.

Параметр **`-p 127.0.0.1:50051:50051/tcp`** указывает Docker перенаправлять TCP-соединения с порта **`50051`** на вашем компьютере на порт **`50051`** внутри контейнера.

Dockerfile для Marketplace

5.4. Теперь создадим образ для сервиса **Marketplace**. Создается аналогичный файл **Marketplace/Dockerfile** с содержимым:

```
FROM python:3.9-slim

RUN mkdir /service
COPY protobufs/ /service/protobufs/
COPY marketplace/ /service/marketplace/
WORKDIR /service/marketplace
RUN python -m pip install --upgrade pip
RUN python -m pip install -r requirements.txt
RUN python -m grpc_tools.protoc -I ../protobufs --python_out=. \
           --grpc_python_out=. ../protobufs/recommendations.proto

EXPOSE 5001
ENV FLASK_APP=marketplace.py
ENTRYPOINT [ "python", "marketplace.py"]
```

Этот **Dockerfile** очень похож на тот, что мы подготовили для микросервиса **Recommendations**, лишь с некоторыми отличиями в конце.

5.5. Собираем образ и запускаем контейнер:

```
C:\WebService>docker build . -f marketplace/Dockerfile -t marketplace
C:\WebService>docker run -p 127.0.0.1:5001:5001/tcp marketplace
```

Настройка сети в контейнерах

Хотя контейнеры **Recommendations** и **Marketplace** уже работают, если перейти в браузере по адресу **http://localhost:5001**, то будет выдано сообщение об ошибке. Дело в том, что контейнеры изолированы.

К счастью, Docker предлагает решение этой проблемы. Можно создать виртуальную сеть и добавить в нее оба контейнера, а также назначить им DNS-имена, чтобы они могли найти друг друга.

Ниже создается сеть под названием **microservices** и запускаем в ней микросервис рекомендаций.

5.6. Сначала остановите запущенные в данный момент контейнеры с помощью **Ctrl + C**.

5.7. Создадим сеть **microservices** и запустим контейнер :

```
C:\WebService>docker network create microservices
C:\WebService>docker run -p 127.0.0.1:50051:50051/tcp --network microservices --nam
recommendations recommendations
```

Команда **docker network create** создает сеть. Достаточно это сделать только один раз – далее к этой сети можно подключить несколько контейнеров. Далее мы добавляем сеть микросервисов в команду **docker run**, чтобы запустить контейнер в этой сети.

Параметр **--name recommendations** указывает имя контейнера в сети.

В файле **marketplace.py** ранее было предусмотрено использование адреса, который должен быть сохранен в переменной среды **RECOMMENDATIONS_HOST:**

```
    recommendations_host = os.getenv("RECOMMENDATIONS_HOST", "localhost")
    recommendations_channel = grpc.insecure_channel(
        f"{recommendations_host}:50051"
)
```

Эти строчки в **marketplace.py** связывают имя хоста микросервиса рекомендаций в переменную среды **RECOMMENDATIONS_HOST**. Если переменная среды отсутствует, то по умолчанию будет использоваться значение **localhost**. Это помогает запускать один и тот же код и непосредственно на компьютере, и внутри контейнера.

5.8. Запускаем котейнер сервиса **marketplace**:

```
C:\WebService>docker run -p 127.0.0.1:5001:5001/tcp --network microservices -e
RECOMMENDATIONS_HOST=recommendations marketplace
```

5.9. На этом этапе можно снова попробовать открыть **localhost:5001** в браузере, и теперь уже страница должна загрузиться.

6.Насктройка Docker Compose

Несмотря на преимущества, такая работа с Docker выглядит немного утомительно, т.к. необходимо создавать, запускать и управлять каждым контейнером отдельно. Было бы неплохо иметь одну команду, которая бы выполнила все необходимое для запуска всех контейнеров. Соответствующее решение называется **`docker-compose`** и является частью проекта Docker.

Вместо того, чтобы запускать кучу команд для создания образов, создания сетей и запуска контейнеров, можно объявить микросервисы в одном YAML-файле.

6.1. Создадим файл **`docker-compose.yaml`**, который обращается к отдельным докерфайлам в соответствующих папках:

```
version: "3.8"
services:

    marketplace:
        build:
            context: .
            dockerfile: marketplace/Dockerfile
        environment:
            RECOMMENDATIONS_HOST: recommendations
        image: marketplace
        networks:
            - microservices
        ports:
            - 5001:5001
    recommendations:
        build:
            context: .
            dockerfile: recommendations/Dockerfile
        image: recommendations
        networks:
            - microservices


networks:
    microservices:
```

Этот файл расположен в корне проекта, а сам проект сейчас имеет следующую структуру:

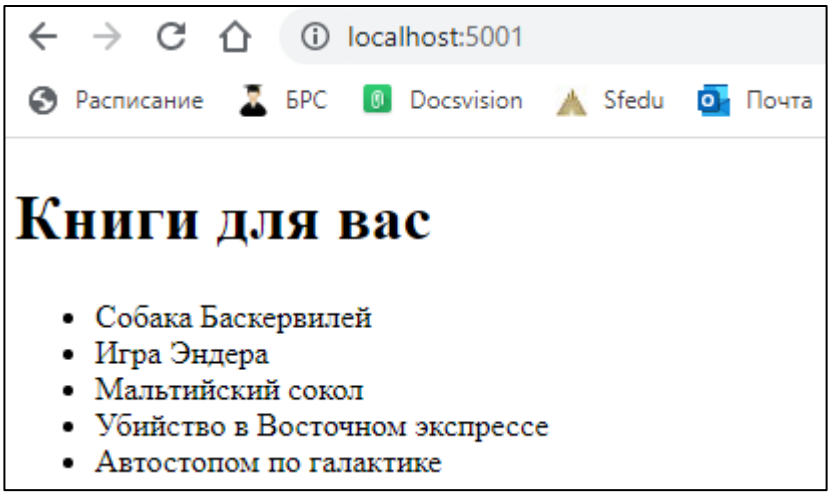
```
.
├── marketplace/
│   ├── Dockerfile
│   ├── marketplace.py
│   ├── requirements.txt
│   └── templates/
│       └── homepage.html
├── protobufs/
│   └── recommendations.proto
├── recommendations/
│   ├── Dockerfile
│   ├── recommendations.py
│   └── requirements.txt
```

```
└── docker-compose.yaml
```

6.2. Теперь для запуска (и первичного создания образов) достаточно набрать лишь одну команду:

```
C:\WebService>docker-compose up
```

6.3. На этом этапе можно открыть **`localhost:5001`** в браузере, и страница должна загрузиться с перечнем рекомендованных книг.



7.Завершение работы

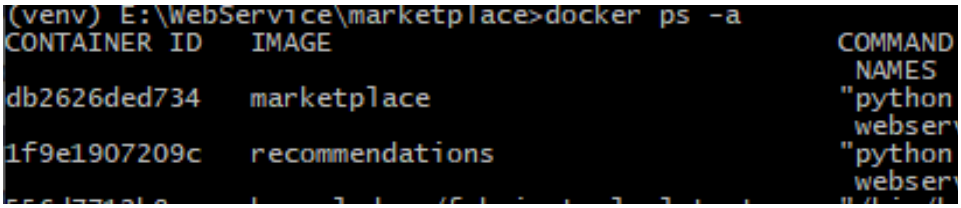
После выполнения работы и проверки работоспособности сервисов можно очистить неиспользуемые ресурсы, т.к. при необходимости всегда можно развернуть и запустить контейнеры из файлов конфигурации проекта командой **`docker-compose up`**.

Для очистки ресурсов сперва необходимо удалить созданные и запущенные контейнеры. Для этого получаем список контейнеров в системе и удаляем контейнеры **`marketplace`** и **`recommendations`**.

7.1. Находим нужные контейнеры и получаем их идентификаторы.

```
C:\WebService>docker ps -a
```

7.2. Получаем список такого вида и в первой колонке определяем **`CONTAINER ID`**



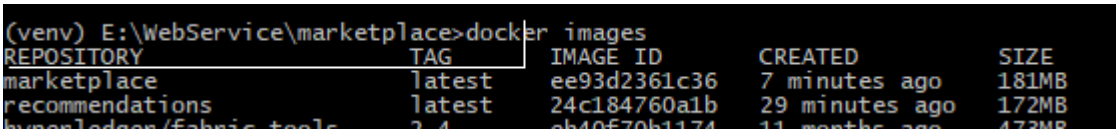
7.3. Удаляем оба контейнера по их ID командой **`docker rm:`**

```
C:\WebService>docker rm db2626ded734
C:\WebService>docker rm 1f9e1907209c
```

7.4. Далее необходимо удалить образы Docker для обоих сервисов. Для этого необходимо получить список образов Docker в системе командой **`docker images`**:

```
C:\WebService>docker images
```

Отобразится список образов Docker в системе.



7.5. Находим образы созданных ранее сервисов и удаляем их по имени командой **`docker image rm`**:

```
C:\WebService>docker image rm -f marketplace
C:\WebService>docker image rm -f recommendations
```

Теперь все ресурсы очищены, на диске остались файлы исходного кода двух сервисов и конфигурационные файлы Docker и Docker-compose. Данные файлы достаточны для разворачивания созданного продукта на любой целевой системе, на которой установлен Docker.

```
.
├── marketplace/
│   ├── Dockerfile
│   ├── marketplace.py
│   ├── requirements.txt
│   └── templates/
│       └── homepage.html
├── protobufs/
│   └── recommendations.proto
├── recommendations/
│   ├── Dockerfile
│   ├── recommendations.py
│   └── requirements.txt
└── docker-compose.yaml
```

Этот набор файлов и является результатом работы. Его можно передать в виде архива или загрузить в репозиторий системы контроля версий (например GitHub).