

## Практическая работа №6. Создание микросервисов

### Цель работы

Цель данной работы приобретение практических навыков разработки backend-приложений для системы микросервисов, а также обеспечения межсервисного взаимодействия на примере созданных приложений.

### Теоретическое введение

Микросервисная архитектура – это распределённая система, состоящая из набора небольших backend-приложений. Чтобы работать с микросервисной архитектурой необходимо сначала создать эти приложения. В данной работе будет рассматриваться пример того, как может быть построен процесс создания сервиса после его проектирования.

### Описание сервиса

В данной работе будет рассмотрено создание сервиса управления доставкой, схема которого представлена на Рисунке 6.1.



Рисунок 6.1 — Схема сервиса управления доставкой

На схеме сервиса управления доставкой отображены самые основные вещи, касающиеся его внутренней работы: команды, которые будет выполнять сервис; события, которыми будет заканчиваться выполнение команд; очереди, с которыми будет взаимодействовать сервис.

Рассмотрим более подробно сервис. Стандартный сценарий работы с сервисом выглядит так:

1. Создание доставки. Пользователь оформляет заказ в сервисе управлением заказами. Этот сервис публикует в очередь событие о том, что создан новый заказ. Данное событие считывается сервисом управления доставкой, после чего информация о доставке вносится в систему. Её назначение – генерация персональных вариантов заданий для студентов, а также автоматическая проверка предоставленных решений.
2. Активация доставки. Сервису управления доставкой необходимо отслеживать, был ли оплачен заказ после оформления, чтоб не получилось так, что клиент не оплатил товар, но заказ всё равно доставили.
3. Назначение курьера. Менеджер магазина с помощью веб-сайта прикрепляет доставку заказа к определённому курьеру.
4. Доставить заказ. Когда курьер передал заказ покупателю, он должен отметить в системе, что заказ был успешно доставлен.

Кроме того, в системе имеются дополнительные функции: получить список доставок, которой могут пользоваться разные типы пользователей для разных целей, и отменить доставку, если пользователь решил отменить заказ.

Наша спроектированная модель несовершенна и не может быть использована в реальной жизни, однако она отлично подходит в качестве демонстрационного примера.

Сервис работы со студентами будет создан с помощью фреймворка FastAPI на языке программирования Python.

## Подготовка проекта

В рамках работы над проектом могут понадобиться различные сторонние модули и системы.

В первую очередь необходимо создать виртуальное окружение проекта, куда будут устанавливаться дополнительные модули, необходимые для работы сервиса.

```
python -m venv venv  
./venv/Scripts/Activate.ps1 # Для Windows
```

Затем после активации виртуального окружения надо установить основные модули, необходимые для создания сервиса на FastAPI.

```
pip install fastapi           # Установка фреймворка FastAPI  
pip install "uvicorn[standard]" # Установка ASGI-сервера  
pip install alembic           # Средство для миграций БД  
pip install pydantic-settings  # Модуль для управления конфигурацией сервиса  
pip install aio_pika           # Модуль для асинхронной работы с RabbitMQ
```

## Бизнес-логика

Основой любого backend-приложения является бизнес-логика, которую оно реализует. К бизнес-логике относятся объекты, с которыми мы работаем, а также функции, которые должно выполнять приложение. Создание интерфейсов внешнего взаимодействия, например, пользователя с приложением или приложения с БД – это вторичный процесс.

Основными объектами, с которыми будет происходить работа в системе являются доставки (delivery) и курьеры (deliveryman). В работе с данным сервисом для описания объектов данных используются анемичные модели. В таком подходе модель представляет собой лишь структуру объекта, а его поведение отделено и вынесено в другие части кода.

Так как в данный сервис создаётся только в качестве примера, функционал, связанный с управлением курьерами представлен достаточно

скучно. Для работы нам будет достаточно иметь лишь id курьера и его ФИО, чтобы проще различать их между собой.

```
# /app/models/deliveryman.py

from uuid import UUID
from pydantic import BaseModel, ConfigDict

class Deliveryman(BaseModel):
    model_config = ConfigDict(from_attributes=True)

    id: UUID
    name: str
```

Модель доставки заказа представлена в системе более богато. Она содержит в себе все поля необходимые для выполнения основного пайплайна доставки товара.

```
# /app/models/delivery.py

import enum
from uuid import UUID
from datetime import datetime
from pydantic import BaseModel, ConfigDict

from app.models.deliveryman import Deliveryman

class DeliveryStatuses(enum.Enum):
    CREATED = 'created'
    ACTIVATED = 'activated'
    DONE = 'done'
    CANCELED = 'canceled'

class Delivery(BaseModel):
    model_config = ConfigDict(from_attributes=True)

    id: UUID
    address: str
    date: datetime
    status: DeliveryStatuses
    deliveryman: Deliveryman | None = None
```

После описания бизнес-объектов, существующих в системе, можно перейти к бизнес-функциям, которые будут помещены в сервисы приложения.

В данном приложении будет достаточно одного сервиса, отвечающего за управление доставкой.

```
# /app/services/delivery_service.py

from uuid import UUID
from datetime import datetime
from app.models.delivery import Delivery, DeliveryStatuses
from app.repositories.local_delivery_repo import DeliveryRepo
from app.repositories.local_deliveryman_repo import DeliverymenRepo

class DeliveryService():
    delivery_repo: DeliveryRepo
    deliveryman_repo: DeliverymenRepo

    def __init__(self) -> None:
        self.delivery_repo = DeliveryRepo()
        self.deliveryman_repo = DeliverymenRepo()

    def get_deliveries(self) -> list[Delivery]:
        return self.delivery_repo.get_deliveries()

    def create_delivery(self, order_id: UUID, date: datetime, address: str) -> Delivery:
        delivery = Delivery(id=order_id, address=address, date=date, status=DeliveryStatuses.CREATED)
        return self.delivery_repo.create_delivery(delivery)

    def activate_delivery(self, id: UUID) -> Delivery:
        delivery = self.delivery_repo.get_delivery_by_id(id)
        if delivery.status != DeliveryStatuses.CREATED:
            raise ValueError
        delivery.status = DeliveryStatuses.ACTIVATED
        return self.delivery_repo.set_status(delivery)

    def set_deliveryman(self, delivery_id, deliveryman_id) -> Delivery:
        delivery = self.delivery_repo.get_delivery_by_id(delivery_id)
        try:
            deliveryman = self.deliveryman_repo.get_deliveryman_by_id(deliveryman_id)
        except KeyError:
            raise ValueError
        if delivery.status != DeliveryStatuses.ACTIVATED:
            raise ValueError
        delivery.deliveryman = deliveryman
        return self.delivery_repo.set_deliveryman(delivery)

    def finish_delivery(self, id: UUID) -> Delivery:
        delivery = self.delivery_repo.get_delivery_by_id(id)
        if delivery.status != DeliveryStatuses.ACTIVATED:
            raise ValueError
        delivery.status = DeliveryStatuses.DONE
        return self.delivery_repo.set_status(delivery)

    def cancel_delivery(self, id: UUID) -> Delivery:
        delivery = self.delivery_repo.get_delivery_by_id(id)
        if delivery.status == DeliveryStatuses.DONE:
```

```
        raise ValueError
    delivery.status = DeliveryStatuses.CANCELED
    return self.delivery_repo.set_status(delivery)
```

## Управление данными. Локальное хранилище

В коде сервиса можно заметить повсеместное использование объектов с названием `*_repo`. Это репозитории, предназначенные для создания, получения и изменения объектов определённого типа без привязки к конкретному источнику данных. За счёт этих объектов бизнес-логика не будет ничего знать о том, где и как хранятся его данные. В данный момент приложению достаточно хранить данные в виде списков объектов. В дальнейшем данный способ частично заменится на работу с базой данных.

Для работы с курьерами необходимо описать весьма простой репозиторий, содержащий в себе всего два метода: получение списка курьеров и получение курьера по его `id`. В нашем приложении будет по умолчанию существовать 3 курьера, заданных заранее.

```
# /app/repositories/local_deliveryman_repo.py

from uuid import UUID

from app.models.deliveryman import Deliveryman

deliverymen: list[Deliveryman] = [
    Deliveryman(id=UUID('85db966c-67f1-411e-95c0-f02edfa5464a'),
                name='Лаптев Иван Алексаендрович'),
    Deliveryman(id=UUID('31babbb3-5541-4a2a-8201-537cdff25fed'),
                name='Зуев Андрей Сергеевич'),
    Deliveryman(id=UUID('45309954-8e3c-4635-8066-b342f634252c'),
                name='Кудж Станислав Алексеевич')
]

class DeliverymenRepo():
    def get_deliverymen() -> list[Deliveryman]:
        return deliverymen

    def get_deliveryman_by_id(self, id: UUID) -> Deliveryman:
        for d in deliverymen:
            if d.id == id:
                return d
```

```
raise KeyError
```

Для работы с доставками понадобится немного более крупный репозиторий, который также создаёт новые доставки и изменяет созданные.

```
# /app/repositories/local_delivery_repo.py

from uuid import UUID

from app.models.delivery import Delivery

deliveries: list[Delivery] = []

class DeliveryRepo():
    def get_deliveries(self) -> list[Delivery]:
        return deliveries

    def get_delivery_by_id(self, id: UUID) -> Delivery:
        for d in deliveries:
            if d.id == id:
                return d

        raise KeyError

    def create_delivery(self, delivery: Delivery) -> Delivery:
        if len([d for d in deliveries if d.id == delivery.id]) > 0:
            raise KeyError

        deliveries.append(delivery)
        return delivery

    def set_status(self, delivery: Delivery) -> Delivery:
        for d in deliveries:
            if d.id == delivery.id:
                d.status = delivery.status
                break

        return delivery

    def set_deliveryman(self, delivery: Delivery) -> Delivery:
        for d in deliveries:
            if d.id == delivery.id:
                d.deliveryman = delivery.deliveryman
                break

        return delivery
```

Полученное на данном этапе решение представляет собой полноценное приложение, которое выполняет все необходимые функции. Однако, чтобы из этого получился полноценный backend, необходимо добавить интерфейсы, с помощью которых с сервисом будут взаимодействовать пользователи и другие системы.

## Интерфейсы взаимодействия

С сервисом управления доставкой могут взаимодействовать разные пользователи, что изображено на Рисунке 6.1. Часть взаимодействий представляет собой синхронные запросы, а часть обработку асинхронных сообщений. На схеме к командам, которые будут выполняться по получению сообщения, подведены стрелки от очередей, из которых эти сообщения будут приходить. Также необходимо помнить о том, что сервис управления доставкой тоже отправляет сообщение в очередь после выполнения команды «Доставить заказ».

Для начала необходимо разобраться с синхронным взаимодействием, которое будет реализовано с помощью HTTP-запросов. Как правило HTTP-интерфейс реализуется в виде REST API. В данном случае это не лучший вариант, так как система подразумевает выполнение определённых действий над объектами, продиктованные предметной областью. В такой ситуации будет достаточно проблематично упаковать все методы с соблюдением принципов REST.

Для реализации HTTP-интерфейса был создан контроллер, вызывающий необходимые методы при получении запросов.

```
# /app/endpoints/delivery_router.py
from uuid import UUID
from fastapi import APIRouter, Depends, HTTPException, Body

from app.services.delivery_service import DeliveryService
from app.models.delivery import Delivery, CreateDeliveryRequest

delivery_router = APIRouter(prefix='/delivery', tags=['Delivery'])
```



```

@delivery_router.get('/')
def get_deliveries(delivery_service: DeliveryService = Depends(DeliveryService))
-> list[Delivery]:
    return delivery_service.get_deliveries()

@delivery_router.post('/{id}/finish')
def finish_delivery(id: UUID, delivery_service: DeliveryService =
Depends(DeliveryService)) -> Delivery:
    try:
        delivery = delivery_service.finish_delivery(id)
        return delivery.dict()
    except KeyError:
        raise HTTPException(404, f'Delivery with id={id} not found')
    except ValueError:
        raise HTTPException(400, f'Delivery with id={id} can\'t be finished')

@delivery_router.post('/{id}/cancel')
def cancel_delivery(id: UUID, delivery_service: DeliveryService =
Depends(DeliveryService)) -> Delivery:
    try:
        delivery = delivery_service.cancel_delivery(id)
        return delivery.dict()
    except KeyError:
        raise HTTPException(404, f'Delivery with id={id} not found')
    except ValueError:
        raise HTTPException(400, f'Delivery with id={id} can\'t be canceled')

@delivery_router.post('/{id}/appoint')
def set_deliveryman(
    id: UUID,
    deliveryman_id: UUID = Body(embed=True),
    delivery_service: DeliveryService = Depends(DeliveryService)
) -> Delivery:
    try:
        delivery = delivery_service.set_deliveryman(id, deliveryman_id)
        return delivery.dict()
    except KeyError:
        raise HTTPException(404, f'Delivery with id={id} not found')
    except ValueError:
        raise HTTPException(400)

```

Теперь существует контроллер, готовый обрабатывать запросы пользователя. Осталось только создать FastAPI приложение и подключить к нему данный контроллер.

```

# /app/main.py

from fastapi import FastAPI

```

```

from app.endpoints.delivery_router import delivery_router

app = FastAPI(title='Delivery Service')

app.include_router(delivery_router, prefix='/api')

```

Теперь после выполнения команды `uvicorn app.main:app` запустится приложение, обрабатывающее все запросы, описанные в контроллере `delivery_router`.

Для работы с асинхронными сообщениями будет использоваться развёрнутый глобально RabbitMQ. Фактически, необходимо написать ещё одну оболочку, которая будет вызывать различные методы из сервиса управления доставками.

```

# /app/rabbitmq.py

import json
import traceback
from asyncio import AbstractEventLoop
from aio_pika.abc import AbstractRobustConnection
from aio_pika import connect_robust, IncomingMessage

from app.settings import settings
from app.services.delivery_service import DeliveryService

async def process_created_order(msg: IncomingMessage):
    try:
        data = json.loads(msg.body.decode())
        DeliveryService().create_delivery(data['order_id'], data['date'],
data['address'])
    except:
        traceback.print_exc()
        await msg.ack()

async def process_paid_order(msg: IncomingMessage):
    try:
        data = json.loads(msg.body.decode())
        DeliveryService().activate_delivery(data['id'])
    except:
        await msg.ack()
    pass

async def consume(loop: AbstractEventLoop) -> AbstractRobustConnection:

```

```

connection = await connect_robust(settings.amqp_url, loop=loop)
channel = await connection.channel()

order_created_queue = await
channel.declare_queue('laptev_order_created_queue', durable=True)
order_paid_queue = await channel.declare_queue('laptev_order_paid_queue',
durable=True)

await order_created_queue.consume(process_created_order)
await order_paid_queue.consume(process_paid_order)
print('Started RabbitMQ consuming...')

return connection

```

Чтобы сконфигурировать ссылку для подключения к RabbitMQ, используется модуль `pydantic_settings`, позволяющий задать все настройки приложения в виде единого объекта, который может быть проинициализирован из различных источников. В данной работе таким источником будет файл `.env`.

```

# /app/settings.py

from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    amqp_url: str
    postgres_url: str

    model_config = SettingsConfigDict(env_file='.env')

settings = Settings()

```

Указав правила, по которым будет происходить чтение сообщений, необходимо подключить новый обработчик к основному циклу приложения, чтобы оно запускалось одной командой. Для этого необходимо немного видоизменить код в файле `/app/main.py`.

```

# /app/main.py

import asyncio
from fastapi import FastAPI

from app import rabbitmq

```

```
from app.settings import settings
from app.endpoints.delivery_router import delivery_router

app = FastAPI(title='Delivery Service')

@app.on_event('startup')
def startup():
    loop = asyncio.get_event_loop()
    asyncio.ensure_future(rabbitmq.consume(loop))

app.include_router(delivery_router, prefix='/api')
```

Теперь при запуске сервиса автоматически запускается чтение и обработка сообщений из очереди.

### Управление данными. Подключение БД

Сейчас все данные хранятся в памяти приложения, поэтому при каждом его перезапуске они удаляются. Чтобы такого не происходило, необходимо подключить внешнюю базу данных, и отправлять данные на хранение туда. В данной работе в качестве СУБД будет использоваться PostgreSQL.

Чтобы работа с СУБД была наиболее удобной, необходимо описать сущности, которые будут храниться в базе кодом. Для этого необходимо создать файл с базовой конфигурацией, которая будет использоваться в каждой сущности.

```
# /app/schemas/base_schema.py

from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

Далее при создании любой из сущностей в БД необходимо использовать созданный объект `Base`, в который будет помещаться вся информация о необходимом конечном состоянии БД.

В данной практической работе необходимо хранить в базе всего один тип объектов – доставки. Для начала необходимо описать схему.

```
# /app/schemas/delivery.py

from sqlalchemy import Column, String, DateTime, Enum
from sqlalchemy.dialects.postgresql import UUID

from app.schemas.base_schema import Base
from app.models.delivery import DeliveryStatuses

class Delivery(Base):
    __tablename__ = 'deliveries'

    id = Column(UUID(as_uuid=True), primary_key=True)
    address = Column(String, nullable=False)
    date = Column(DateTime, nullable=False)
    status = Column(Enum(DeliveryStatuses), nullable=False)
    deliveryman_id = Column(UUID(as_uuid=True), nullable=True)
```

После этого необходимо добавить новый репозиторий, который будет работать не с локальным хранилищем, а с глобальной базой данных.

```
# /app/repositories/bd_delivery_repo.py

from uuid import UUID
from sqlalchemy.orm import Session

from app.database import get_db
from app.models.delivery import Delivery
from app.schemas.delivery import Delivery as DBDelivery

class DeliveryRepo():
    db: Session

    def __init__(self) -> None:
        self.db = next(get_db)

    def get_deliveries(self) -> list[Delivery]:
        deliveries = []
        for d in self.db.query(DBDelivery).all():
            deliveries = Delivery.from_orm(d)
        return deliveries

    def get_delivery_by_id(self, id: UUID) -> Delivery:
        delivery = self.db \
```

```
.query(DBDelivery) \
.filter(DBDelivery.id == id) \
.first()

if delivery == None:
    raise KeyError

return Delivery.from_orm(delivery)
...
```

## Контейнеризация приложения

Как правило для простоты запуска и стабильности работы приложения запускают в виде docker-контейнеров. За счёт этого приложение достаточно легко запускается, просто масштабируется и одинаково работает на разных машинах.

Однако, перед тем как производить сборку контейнера с приложением необходимо зафиксировать все модули, которые были установлены в наше виртуальное окружение в процессе работы с приложением. Как правило зависимости проектов на языке Python записываются в файл `requirements.txt`.

```
pip freeze > requirements.txt
```

После выполнения команды появится файл, в котором будут перечислены все модули, от которых зависит проект.

Теперь можно перейти к созданию docker-образа. Для этого необходимо создать файл `Dockerfile`, где будут перечислены все шаги, необходимые для успешного создания образа.

```
FROM python:3.10

# Выбор папки, в которой будет вестись работа
WORKDIR /code

# Установка зависимостей проекта
COPY ./requirements.txt /code/
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

# Перенос проекта в образ
COPY ./app /code/app
```

```
# Копирование файлов alembic
COPY ./alembic.ini /code/alembic.ini

EXPOSE 8000

CMD ["/bin/sh", "-c", \
    alembic upgrade head && \
    uvicorn student_service.api.main:app --host 0.0.0.0 --port 80"]
```

Для удобного хранения конфигураций, необходимых для запуска контейнеров, а также их оркестрации можно воспользоваться плагином `docker compose`. Для этого понадобится создать ещё один файл: `docker-compose.yaml`. Данный файл не должен находиться ни в одном из запускаемых сервисов. Здесь будет необходимо описать все сервисы, которые мы собираемся запускать, параметры их запуска, значения, которые будем передавать в сервисы. В общем виде схема весьма простая: надо переписать корректную команду для запуска docker-контейнера в другом формате в файл.

```
version: '3.9'

# Секция, где перечислены все сервисы, с которые относятся к системе
services:
  delivery_service:
    # Путь до Dockerfile сервиса
    build: .
    # Даём доступ к 80 порту внутри контейнера через 8000 на машине
    ports:
      - 8000:80
    # Передаём переменные окружения, которые будут использованы приложением
    environment:
      - POSTGRES_URL=postgresql://myuser:pgpwd@student_db:5432/postgres
      - AMQP_URL=amqp://guest:guest123@51.250.26.59:5672/
    # Перед запуском этого сервиса ждём пока запустится БД
    ...
```

Благодаря Docker Compose можно поднять все описанные сервисы одной командой:

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

Ссылка на проект: [https://github.com/IvLaptev/MA\\_6](https://github.com/IvLaptev/MA_6).

### **Задание на самостоятельную работу**

Для выполнения практической работы необходимо реализовать небольшую микросервисную систему, которая будет содержать в себе минимум два собственных микросервиса.

Общие требования к работе:

- каждый микросервис должен содержать от двух пяти;
- в одном из микросервисов для одного из эндпоинтов должны вызываться вложенные функции (например, при обработке запроса вызывается функция из какого-то сервиса);
- все микросервисы должны относиться к одной предметной области;
- хотя бы один из микросервисов должен работать с БД.

В конце выполнения данной работы необходимо упаковать каждый микросервис в отдельный docker-контейнер, а также настроить запуск приложений с помощью Docker Compose.