

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа 3-4

Выполнил:

Пономарев Константин

Группа К33402

Проверил:

Добряков Д. И.

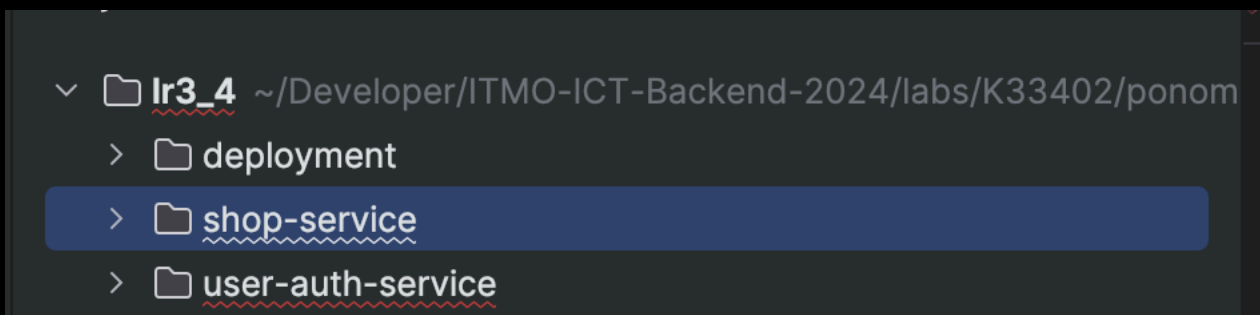
**Санкт-Петербург
2024 г.**

Задача

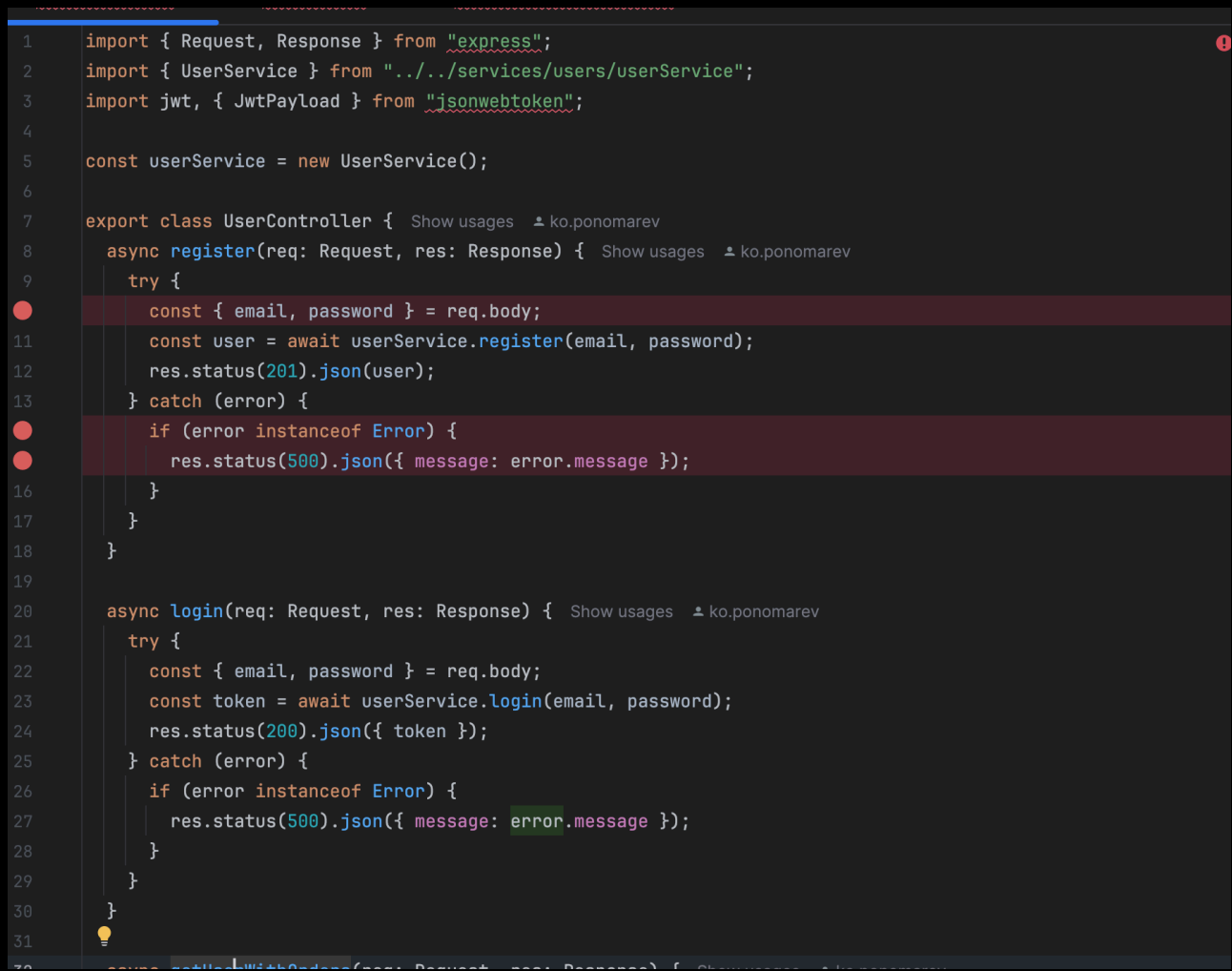
В третьей лабораторной работе необходимо было разбить один монолит на микросервисы. В четвертой необходимо было развернуть docker-compose для микросервисов, бд, подключить rabbitmq и протестировать его

Ход работы

Для начала я решил бить свой проект на auth и основной (shop) сервисы



Решил делать микросервисы в монорепозитории, поскольку это было проще, чем бить каждый микросервис в свой



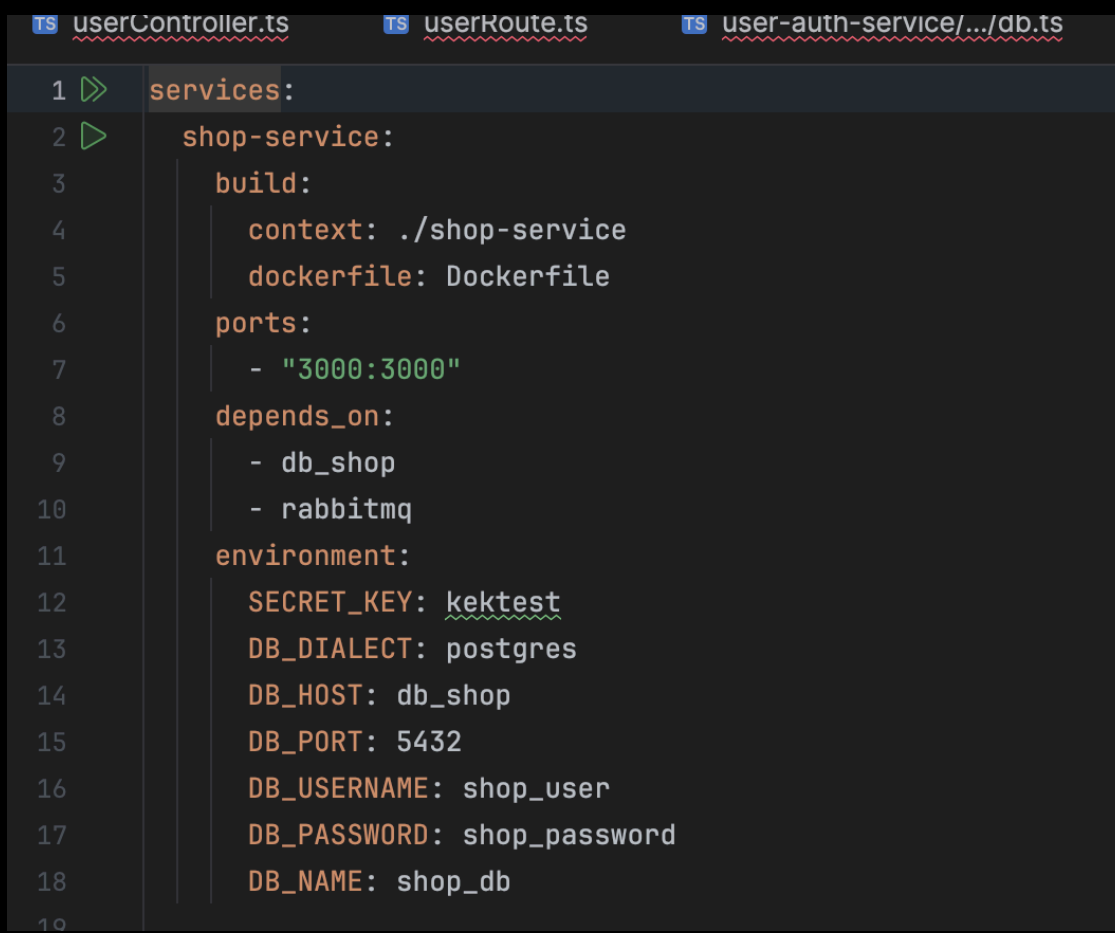
репозиторий. Да и по опыту работы знаю, что гораздо проще в монорепо поменять изменение и залить его на остальные модули.

user-auth-service теперь отвечает чисто за функционал регистрации и авторизации. shop отвечает за работу магазина с помощью токена

Далее наступили тяжелые времена, пришлось настраивать докер. Раньше же он очень сильно лагал на mac с m процессорами, но по итогу уже все стабильно работает (не может не радовать)

Давайте поговорим про это подробнее, поскольку это была наиболее интересная часть всей работы

Для начала я создал ямлик docker-compose.yml, в котором буду конфигурировать контейнеры. Собственно покажу на одном примере, остальное сделано аналогично



```
1  >> services:
2  >   shop-service:
3     build:
4       context: ./shop-service
5       dockerfile: Dockerfile
6     ports:
7       - "3000:3000"
8     depends_on:
9       - db_shop
10      - rabbitmq
11     environment:
12       SECRET_KEY: kektest
13       DB_DIALECT: postgres
14       DB_HOST: db_shop
15       DB_PORT: 5432
16       DB_USERNAME: shop_user
17       DB_PASSWORD: shop_password
18       DB_NAME: shop_db
19
```

Что здесь интересного: да особо ничего, задаем env переменные, в build указываем с откуда мы будем брать всё, что надо, ну и сам Dockerfile

```
db_shop:|
  image: postgres:13
  environment:
    POSTGRES_USER: shop_user
    POSTGRES_PASSWORD: shop_password
    POSTGRES_DB: shop_db
  ports:
    - "5432:5432"
```

Базу конфигурируем буквально копипастом с ближайшей доки, тут ничего сложного тоже нет, будем к ней биться по порту 5432

Теперь начинается веселье с двумя файлами: Dockerfile и tsconfig.json. На этом этапе нам необходимо настроить все таким образом, чтобы оно работало как часы (у меня сначала не получилось)

```
1  >> FROM node:14
2
3  WORKDIR /usr/src/app
4
5  COPY package*.json ./
6
7  RUN npm install
8
9  COPY . .
10
11 RUN npm run build
12
13 EXPOSE 3000
14
15 CMD ["npm", "start"]
```

На что обратить внимание стоит: ну наверное на то, что каждое поле делает: 1 строка – качаем nodejs 14 версии, складывать все будем в usr/src/app. Копируем файлы в контейнер и открываем 3000 порт

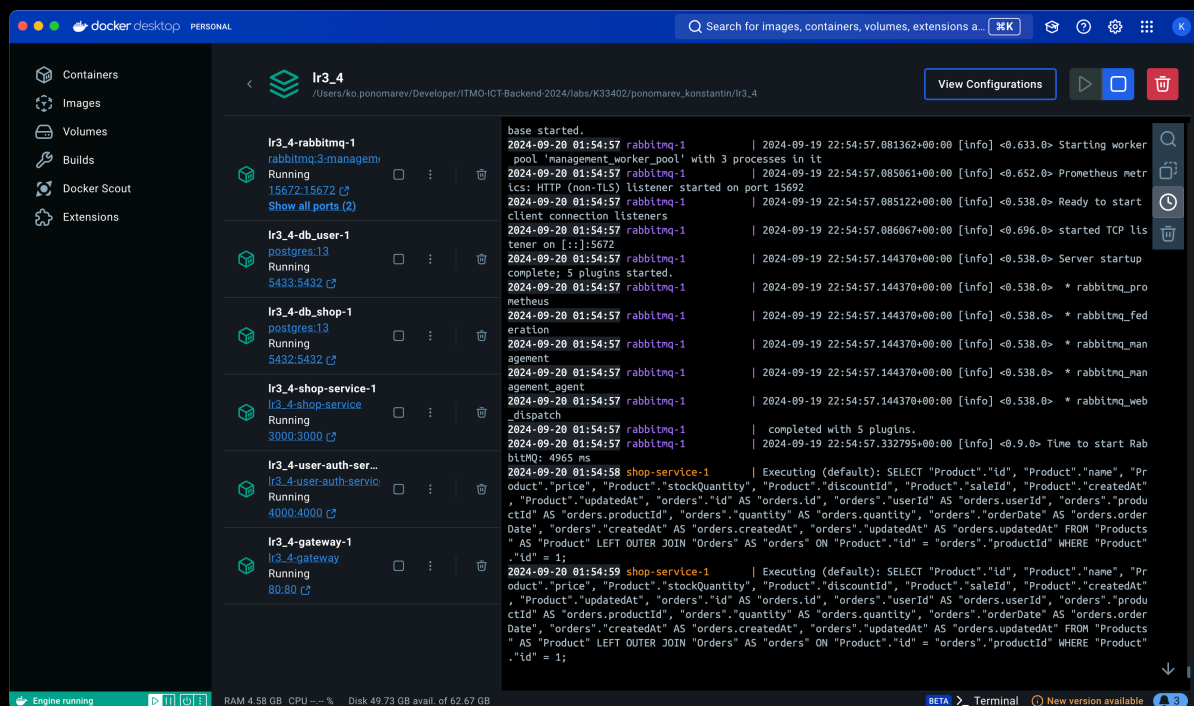
Далее tsconfig.json. Я убил 2(!!!) часа чтобы завести всё это дело. Сначала начал ресерчить, в чем проблема, все советовали просто переустановить пакеты, потому что была ошибка MODULE_NOT_FOUND. Но основная проблема крылась в том, что build файлы некорректно раскидывались в директории, поэтому мой script build не работал. Ну в общем с горем пополам настроил я всё это дело

```
"scripts": {  
  "build": "npx tsc",  
  "start": "tsc && node dist/src/index.js",  
  "dev": "npx tsc & node dist/src/index.js",
```

```
// "outFile": "./",  
"outDir": "./dist",  
  
"rootDir": "./",
```

Но это еще не всё. Далее мы запускаем докер через docker-compose build && docker-compose up (не зря учил bash на 1ом курсе, решил не писать в две отдельные команды :))

Ииии видим заветные 6 контейнеров, которые крутятся

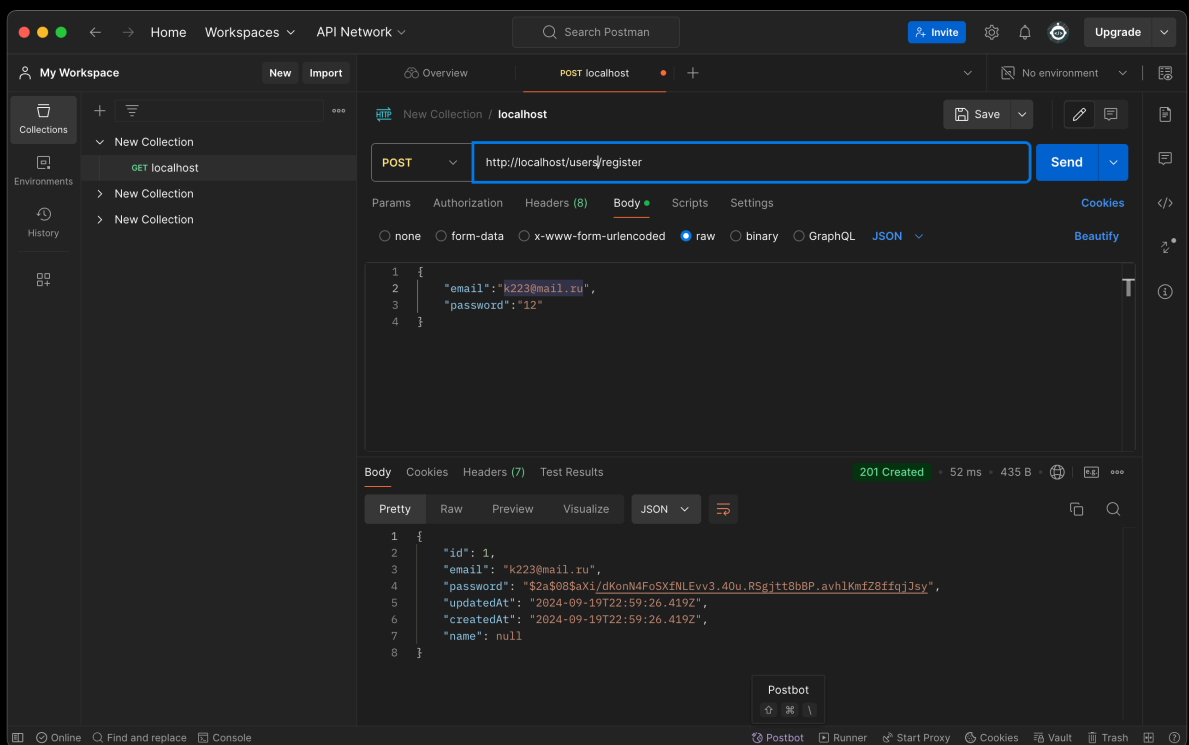
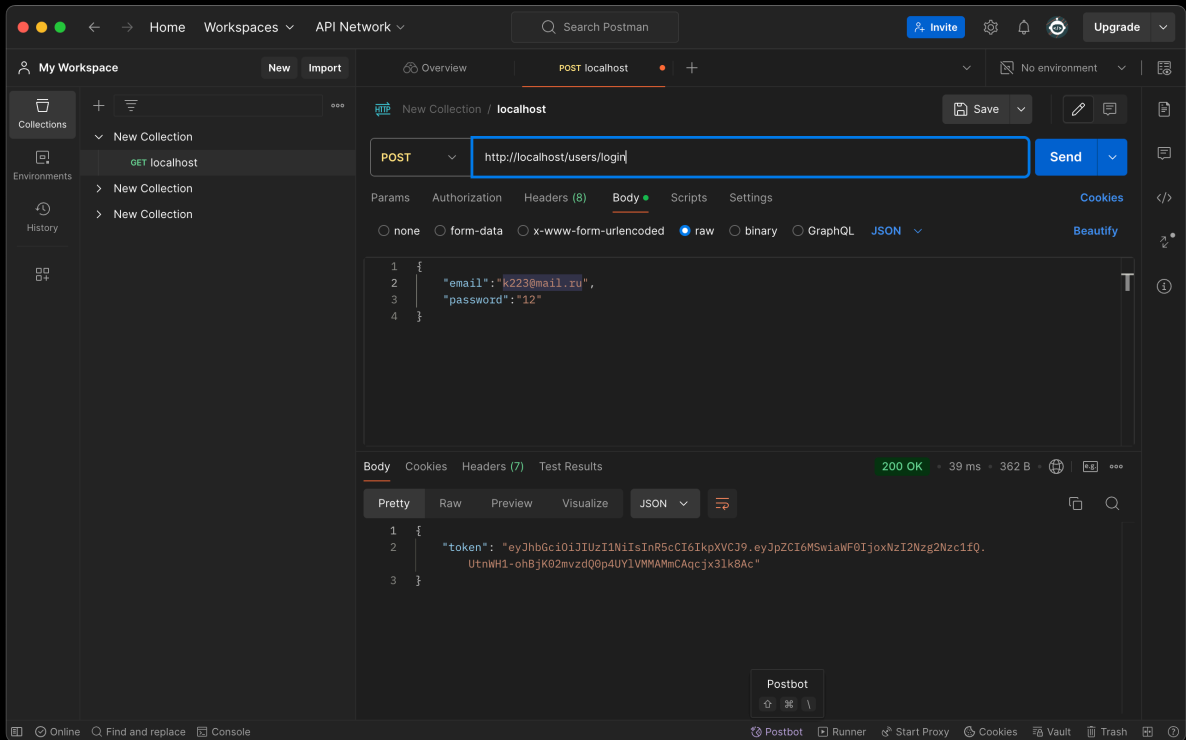


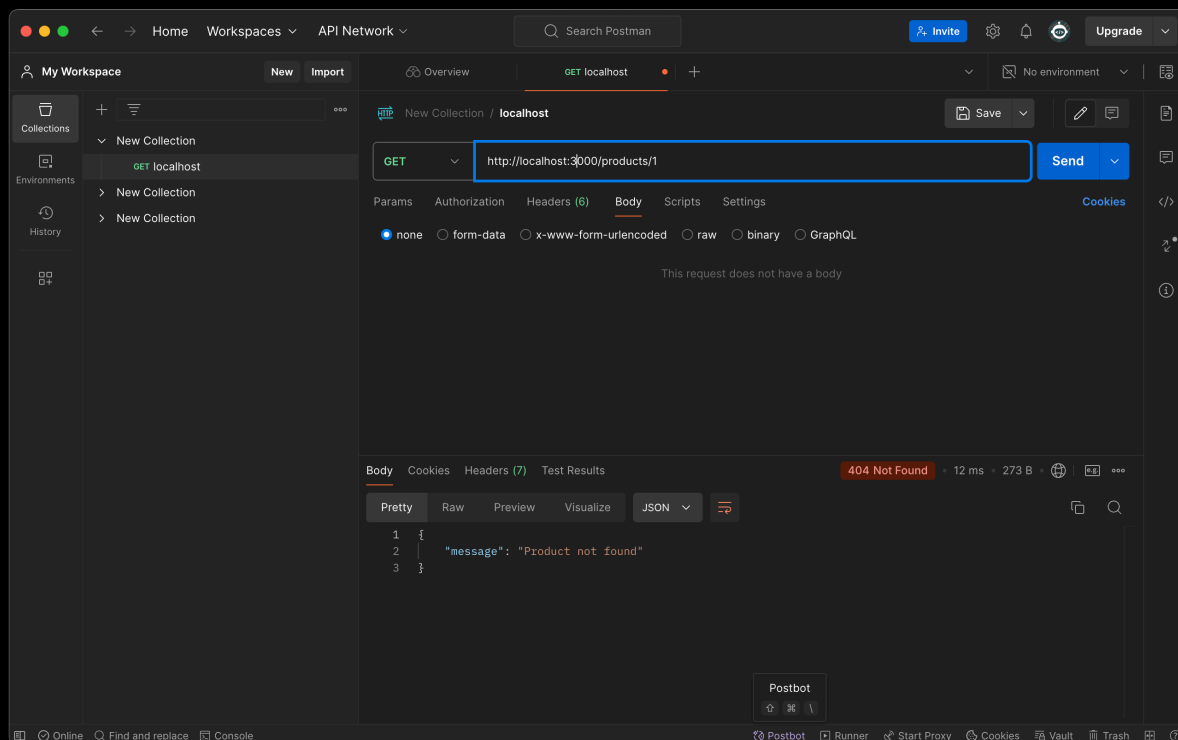
Теперь надо задуматься о том, чтобы мы не ходили каждый раз к разным хостам. В этом нам поможет либо `apache`, либо `nginx`. Я выбрал `nginx`, думал будет просто – но в итоге пришлось тоже повозиться, чтобы сделать конфиг. Давайте глядеть. На работе подсмотрел наш древний, абсолютно костыльный подход с `api-gateway` (на прод я бы так ни сделал никогда) и подумал, что это замечательная идея попробовать набросать что-то такое же. Прошу приветствовать, `nginx.conf`:

```

1 worker_processes 10;
2
3 events {
4     worker_connections 1024;
5 }
6
7 http {
8     server {
9         listen 80;
10
11         location /shop {
12             proxy_pass http://shop-service:3000;
13             proxy_set_header Host $host;
14             proxy_set_header X-Real-IP $remote_addr;
15             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
16             proxy_set_header X-Forwarded-Proto $scheme;
17         }
18
19         location /user {
20             proxy_pass http://user-auth-service:4000;
21             proxy_set_header Host $host;
22             proxy_set_header X-Real-IP $remote_addr;
23             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
24             proxy_set_header X-Forwarded-Proto $scheme;
25         }
26     }
27 }

```





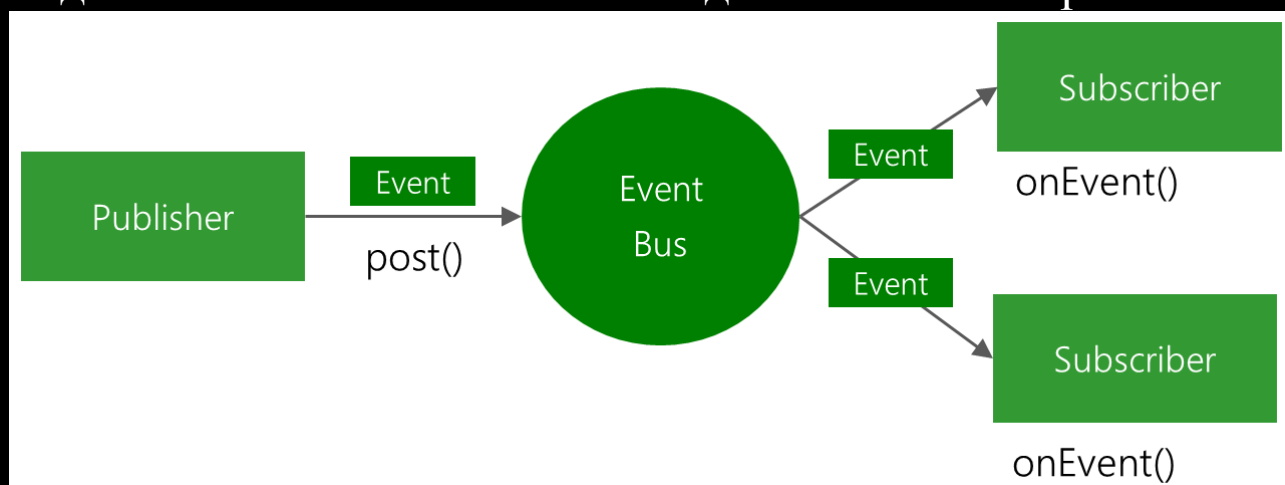
Ну в общем-то получилось как-то так, все запросы работают, можем обращаться напрямую, можем через прокси

Теперь настала очередь rabbitMQ

Наклепал простейший пример

```
8
9 export function connectToRabbitMQ(retries = 5) { Show usages ko.ponomarev
10   amqp.connect(RABBITMQ_URL, (err, connection) => {
11     if (err) {
12       console.error('Failed to connect to RabbitMQ:', err);
13       if (retries > 0) {
14         console.log("Retrying in 5 seconds... (${retries} retries left)");
15         setTimeout(() => connectToRabbitMQ(retries - 1), 5000);
16       } else {
17         throw err;
18       }
19     } else {
20       console.log('Connected to RabbitMQ');
21       receiveMessage(connection);
22     }
23   });
24 }
25
26 function receiveMessage(connection: amqp.Connection) { Show usages ko.ponomarev
27   connection.createChannel((err, channel) => {
28     if (err) {
29       throw err;
30     }
31     const queue = 'user_notifications';
32
33     channel.assertQueue(queue, { durable: false });
34
35     console.log("[*] Waiting for messages in ${queue}. To exit press CTRL+C");
36
37     channel.consume(queue, (msg) => {
38       if (msg) {
39         console.log("[x] Received ${msg.content.toString()}");
40       }
41     }, { noAck: true });
42   });
43 }
```


Сам rabbit работает по простейшей схеме, как event bus, подписались на тип события и ждем пока к нам прилетит



Вывод

Бить на микросервисы (модули) для меня не в новинку, покажу вам на примере работы, в андроид у меня примерно та же задача стоит)

Было интересно вспомнить свой 1ый курс, когда всю ночь мучался с Dockerом ради запуска Марио. Здесь ситуация та же, но запускал, к сожалению, не Марио :)

Понял, что nginx не очень, лучше лишний раз сходить к другому сервису, дернуть с него данные

Поработал с rabbitMQ, поотправлял сообщения. Но интереснее гораздо было с Docker