Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Владимирский государственный университет

имени Александра Григорьевича и Николая Григорьевича Столетовых»

(ВлГУ)

Кафедра информационных систем и программной инженерии

Лабораторная работа №6

по дисциплине "Технологии разработки веб приложений"

ТЕМА РАБОТЫ:

Подготовка к развертыванию приложения

Выполнил:

студент гр. ПРИ-120

Парахин К.В.

Принял:

преподаватель кафедры ИСПИ

Шамышев А.А

Владимир 2024 г

Цель работы:

Изучить принципы и технологии развертывания клиент-серверных веб приложений

Задание:

Есть клиент-серверное веб-приложение.

Оно состоит из серверного .NET приложения, которое является REST .API + signalR (поддерживает протокол для web sockets), общается с СУБД Postgres SQL

Клиентским приложением является Flutter WEB приложение, которое имеет в себе некоторый набор ресурсов и в нем настроена маршрутизация (поэтому в целом выглядит снаружи как React приложение)

Выполнение работы

Для выполнения задачи я буду использовать развертывание с помощью создания контейнеров каждого из приложений с помощью технологии Docker

По итогу, у меня получится свой docker image как для серверного приложения, так и клиентского

Dockerfile серверного приложения

# Build stage

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

WORKDIR /src

COPY ["LogisticAPI/LogisticAPI.csproj", "LogisticAPI/"]

RUN dotnet restore "LogisticAPI/LogisticAPI.csproj"

COPY . .

WORKDIR "/src/LogisticAPI"

RUN dotnet build "LogisticAPI.csproj" -c Release -o /app/build

# Publish stage

FROM build AS publish

RUN dotnet publish "LogisticAPI.csproj" -c Release -o /app/publish /p:UseAppHost=false

# Runtime stage

FROM mcr.microsoft.com/dotnet/aspnet:9.0

WORKDIR /app

COPY --from=publish /app/publish .

EXPOSE 8080

ENTRYPOINT ["dotnet", "LogisticAPI.dll"]

Docker file клиентского приложения

FROM ghcr.io/cirruslabs/flutter:3.16.9 AS buil

RUN git config --global --add safe.directory '\*' && \

    chmod -R 777 /sdks/flutter

WORKDIR /app

COPY . .

RUN flutter pub get && \

    flutter build web --release --web-renderer html --base-href /

FROM nginx:stable-alpin

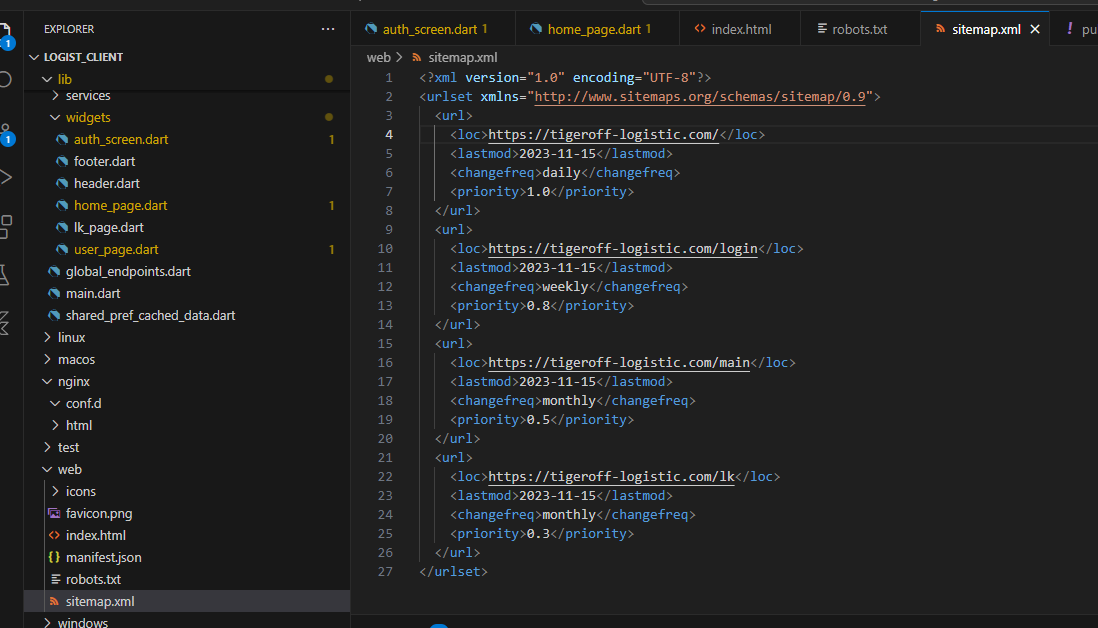
COPY --from=build /app/build/web/ /usr/share/nginx/html/

EXPOSE 80

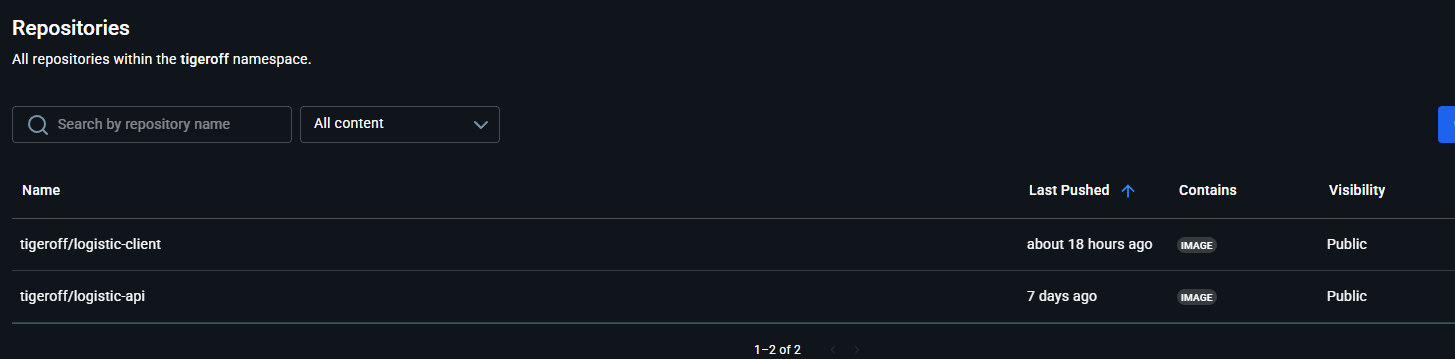
EXPOSE 443

CMD ["nginx", "-g", "daemon off;"]

При этом для веб-приложения, как и просилось в задании, я добавил дополнительно файлы для поисковых брузеров: robots.txt и sitemap.xml (карту сайта



После сборки docker образов их можно отправить на docker hub – что я и сделал, назвав их tigeroff\logistic-api и tigeroff\logistic-client, соответственно



Эти образы можно использовать для проведения CD процессов по созданию контейнеров и общей сети между ними – простейшим вариантом было бы использование docker compose – что я изначально и сделал, чтобы просто локально проверить, как сервер и клиент будет в контейнере работать вместе – все было окей – дополнительно я добавлял конфиги и образ для nginx – который выполнял роль прокси-сервера и занимался маппингом запросов и балансировкой нагрузки

Вот пример серверного docker-compose.yaml:

services:

postgres:

container\_name: postgres\_container

image: postgres:17

command:

- "postgres"

- "-c"

- "max\_connections=50"

- "-c"

- "shared\_buffers=1GB"

- "-c"

- "effective\_cache\_size=4GB"

- "-c"

- "work\_mem=16MB"

- "-c" POSTGRES\_USER: postgres

POSTGRES\_PASSWORD: root

POSTGRES\_DB: logistic\_db

POSTGRES\_HOST: postgres

ports:

- 5433:5432

volumes:

- ./Database/Migrations/:/docker-entrypoint-initdb.d/

networks:

- internal

restart: always

logistic-api:

image: tigeroff/logistic-api:latest

environment:

- ASPNETCORE\_URLS=http://+:8080

- ASPNETCORE\_ENVIRONMENT=Production

- CORS\_\_AllowedOrigins=https://tigeroff-logistic.com,http://localhost

- CORS\_\_AllowedMethods=GET,POST,PUT,DELETE

depends\_on:

- postgres

networks:

- internal

restart: always

expose:

- "8080"

nginx\_server:

image: nginx:alpine

container\_name: logistic-nginx

volumes:

- ./nginx.conf:/etc/nginx/nginx.conf

ports:

- "3500:80"

depends\_on:

- logistic-api

networks:

- internal

networks:

internal:

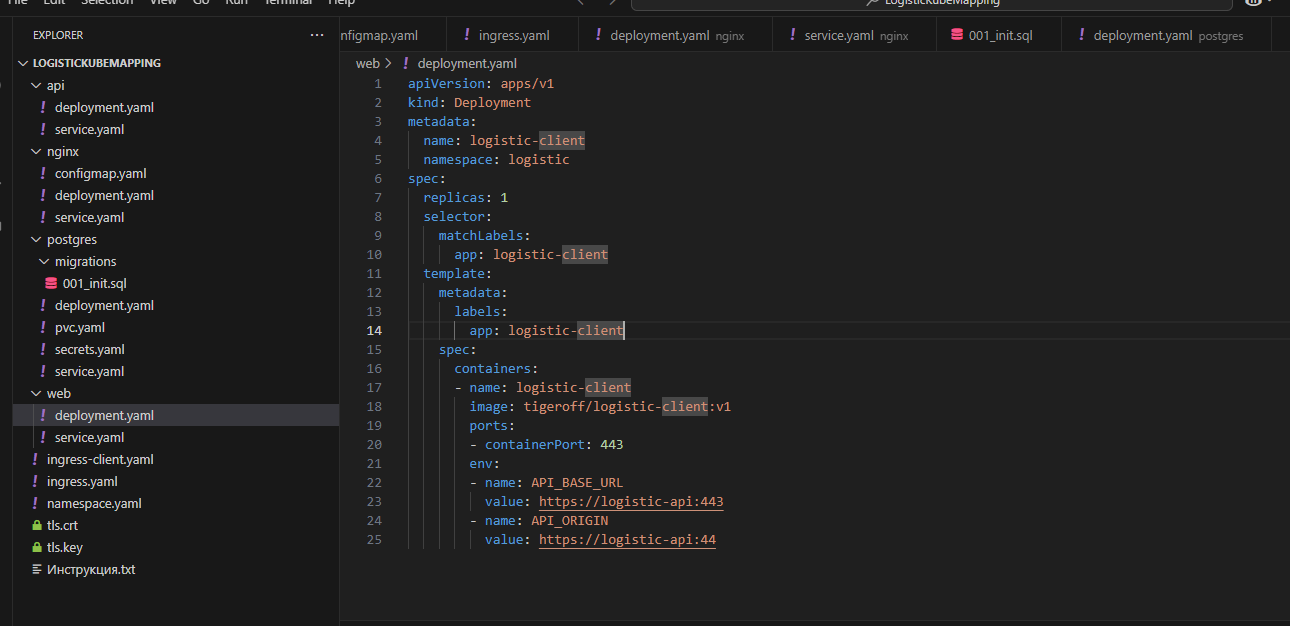
external: true

#docker network create internal

#docker-compose up --build

Но этот вариант показался мне не очень удобным и надежным + у меня были мысли на будущее добавлять новые микросервисы на стороне сервера – поэтому я решил сделать развертывание в CD с помощью Kubernetes

Я установил утилиты kubectl и helm для работы с ним – а далее перешел к настройке конфигов для работы poda postgres БД, серверного API и клиентского web приложений

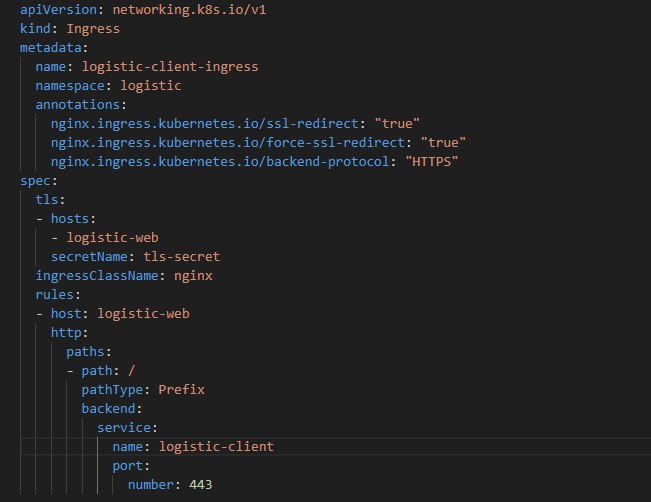


Стоит учесть, что использовалась не совсем стандартная настройка конфигов для СУБД – а именно был вынесен отдельный файл с миграцией базы – который всегда накатывается при первоначальном создании контейнера postgres

Для каждого из контейнеров было необходимо настроить deployment.yaml – файл, описывающий развертывание docker image, параметры среды (env) – с помощью которых можно переопределять какие то законфигурированные приложениями параметры, а также настройки количества реплик приложения + его открытых портов – а также service.yaml – с настройкой пода как сервиса, с которым можно как-то взаимодействовать во внутренней сети

Для сервера и клиента я открыл порты 80 (для http) и 443 (для https) – хотя дополнительно настроил в другом месте то, что базово должен использоваться именно ssl соединение с использованием https – а при обращении к http – приложение пытается выполнить автоматический редирект на https

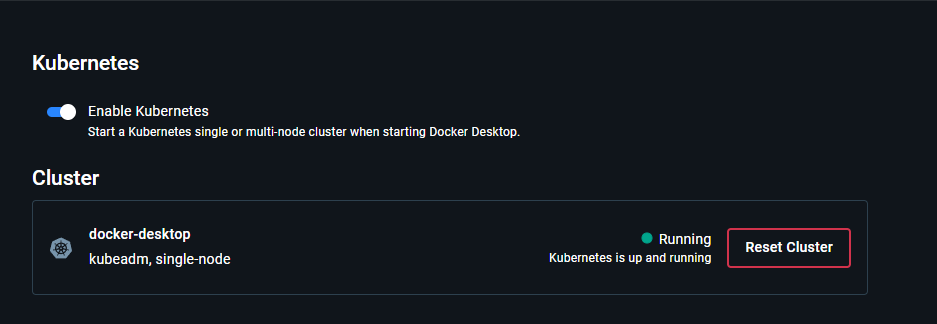
Эти действия настраиваются в специальном ingress конфиге:



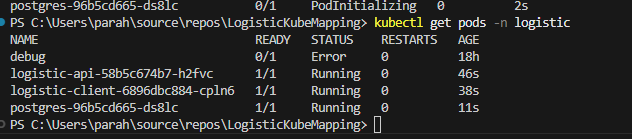
Это конфиг предназначен для корректной работы ingress контроллера – который по факту предоставляет для закрытых от внешней сети подов приложения входную точки (прокси или даже скорее gateway) – конкретно реализация использовалась от nginx

Ранее я сказал, что поды приложения и ingress контроллер были сконфигурированы под использование ssl безопасного соединения – чтобы использовать https недостаточно просто открыть порт 443, нужно еще добавить либо самоподписанные, либо подписанные lets encrypt библиотекой сертификаты безопасности

Также чтобы стартовать контейнер и его поды (все сделано в рамках одного namespace: logistic) – необходимо каким то образом запустить kubernetes кластер. Для локального тестирования я использовал кластер от docker desktop



В рамках текущего namespace можно посмотреть, что существуют и работают поды приложений

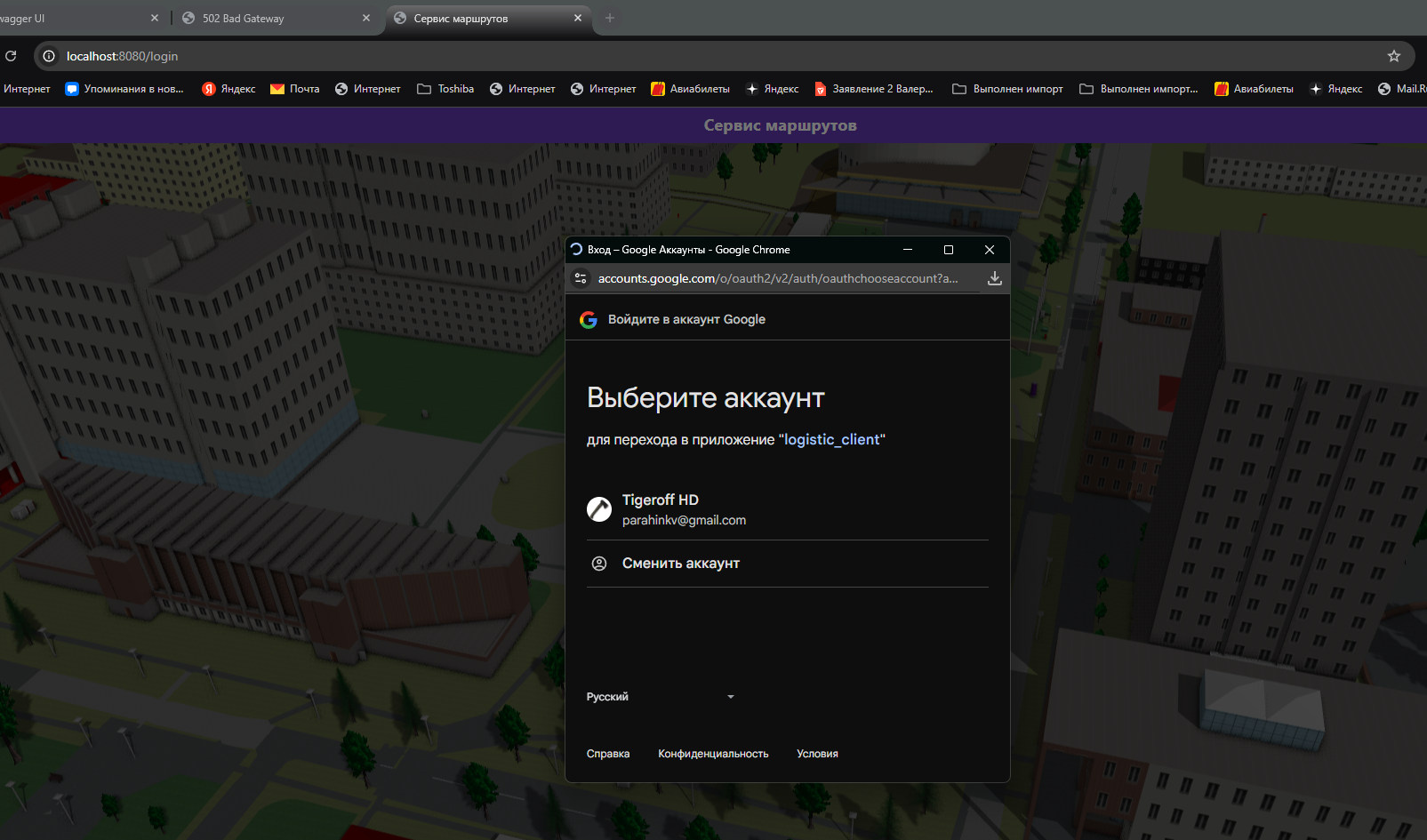


Если сделать hc запрос к приложению – видно, что оно отдает healthy



Также видно, что работает api запрос на авторизацию через swagger – сам запрос делает регистрацию нового пользователя – то есть ведет обращение к БД – что говорит о том, что взаимодействие сервера с БД корректно работает

Также работает под клиентского приложения, можно обращаться к разным его страницам и использовать google аутентификацию – которая под капотом будет делать сначала запрос к Google OAuth серверу + а потом с одобренными credentials делать запрос к серверному API для авторизации пользователя



Вывод

В результате выполнения работы я изучил технологии по развертыванию клиент-серверных веб приложений и подготовил свое приложение к публикации на удаленном домене