Команды:

Просмотр всех установленных компонентов:

Kubectl get all

Обновление конфигурации:

Kubectl apply -f <имя файла конфигурации.yaml> <- для применения конкретного файла

Kubectl apply -f <путь к папке с файлами \*.yaml> <- для применения всех файлов \*.yaml в указанной папке

Развертывания:

kubectl delete --all deployments <- удалить все развертывания

kubectl get deployments <- просмотреть все развертывания

Сервисы:

kubectl delete --all services <- удалить все сервисы

kubectl get servives <- просмотреть все сервисы

kubectl describe svc <- просмотреть определения сервисов

kubectl get endpointslices <- просмотреть

Поды:

kubectl delete --all pods <- удалить все поды

kubectl get pods <- просмотреть все поды

Сеты:

kubectl delete statefulsets –all <- удалить все сеты

kubectl get statefulsets <- просмотреть все сеты

Ingress:

Kubectl get ingress <- просмотреть ingress-сервисы

Kubectl delete ingress -all <-удалить ingress-сервисы

Локальные тома и локальные тома клаймсы:

kubectl get pv <- просмотреть локальные тома

kubectl get pvс <- просмотреть тома-клаймсы

kubectl delete –all pv <- удалить локальные тома

kubectl delete –all pvc <- удалить локальные тома-клаймсы

kubectl delete pv <наименование тома>

kubectl delete pvc <наименование тома-клаймса>

Секреты:

kubectl create secret generic <наименование секрета> --from-literal <наименование ключа>=<значение> [--from literal ...] <- создать секрет

kubectl get secrets <- просмотреть секреты

kubectl delete secrets –all <- удалить секреты

Просмотреть лог нужного пода:

kubectl logs <наименование пода>

Подключиться к поду postgres:

kubectl exec -it <наименование пода> -- psql -h localhost -U postgres --password -p 5432 postgres

Просмотр установок кластера:

kubectl cluster-info <- это краткая аннотация

kubectl cluster-info dump <- это куча всей инфы по кластеру, поэтому лучше выводить в файл:

kubectl cluster-info dump > dump.txt

Установка балансера запросов (если не используем Ingress):

kubectl expose deployments <наименование деплоя> --type=LoadBalancer

**Разворачиваем сервер БД PostgreSql и его данные в кластере**

1. **Ставим БД в кластер**

- загружаем образ postgres в докер -> docker pull postgres;

- делаем файл подключения тома между кластером и файловой системой - postgres-volume.yaml;

* делаем файл разворачивания экземпляра докера в кластере - postgres-deployment.yaml;

Разворачиваем сначала том:

kubectl apply -f postgres-volume.yaml

Далее разворачиваем экземпляр сервера postgres:

kubectl apply -f postgres-deployment.yaml

Проверяем:

kubectl get pv <-это должен быть смонтирован том Volume, должен быть Bounded

kubectl get pvc <- это должен быть смонтирован том VolumeClaim, должен быть Bounded

kubectl get pods <- должен быть запущен экземпляр postgre-0, статус Running

1. **Восстанавливаем БД**

- должен быть бекап БД в формате tar;

- открываем каталог - [\\wsl$\docker-desktop\mnt\host\D\Docker\Volumes\Postgres](file:///\\wsl$\docker-desktop\mnt\host\D\Docker\Volumes\Postgres), там должны быть данные – здесь спроецирован каталог с данными работающего экземпляра сервера postgres. Копируем в этот каталог наш бекап.

- открываем в DockerDesctop в запущенных контейнерах контейнер с именем k8s\_postgre\_postgre-0\_....

Это наш экземпляр.

Или его можно открыть из консоли:

kubectl exec –stdin –tty postgre-0 -- /bin/bash

Далее устанавливаем файловый менеджер:

apt update

apt install mc

mc

переходим -> /var/lib/postgresql/data

там должен находится наш бекап, проверяем так ли это

выходим из mc -> exit

также можно было переходить командами cd (переход) и ls (просмотр).

**Восстанавливаем нужные БД из бекапов.**

- подключаемся к БД и создаем нужную БД:

psql -U postgres -d postgres

create database <имя БД>;

выходим в командную консоль – exit

восстанавливаем БД:

pg\_restore -h localhost -p 5432 -U postgres -d <имя БД> -v <имя файла бекапа БД>

должно все восстановится. Проверяем:

psql -U postgres -d <имя БД>

select \* from “public”.”<наименование любой не пустой таблицы БД>”;

должны отобразится данные, важно – наименование таблиц – casesensitive! Так что нужно указывать точно.

1. **Восстанавливаем pgAdmin в кластере.**

- нужный файл конфигурации -> pgadmin.yaml;

Там важен порт – 30200 – по нему потом будем коннектиться через браузер, порт может быть любой

Сохраняем скрипт в файл – pgadmin.yaml.

* Скачиваем образ pgadmin -> docker pull dpage/pgadmin4
* Разворачиваем под в кластере -> kubectl apply -f pgadmin.yaml
* Проверяем что все работает – должен быть сервис и deployment -> kubectl get all
* Подключаемся к pgadmin, он работает через браузер по порту 30200 – это видно в определении сервиса:



* Во входных данных логин – то, что указали в файле конфигурации pgadmin -> [admin@admin.com](mailto:admin@admin.com)

Пароль - admin, должно все открыться.

* Подключаемся к серверу БД ->

Адрес сервера – свой локальный IP-адрес, смотрим по команде ipconfig

Порт – то что указали в файле конфигурации определения сервера БД -> 30432

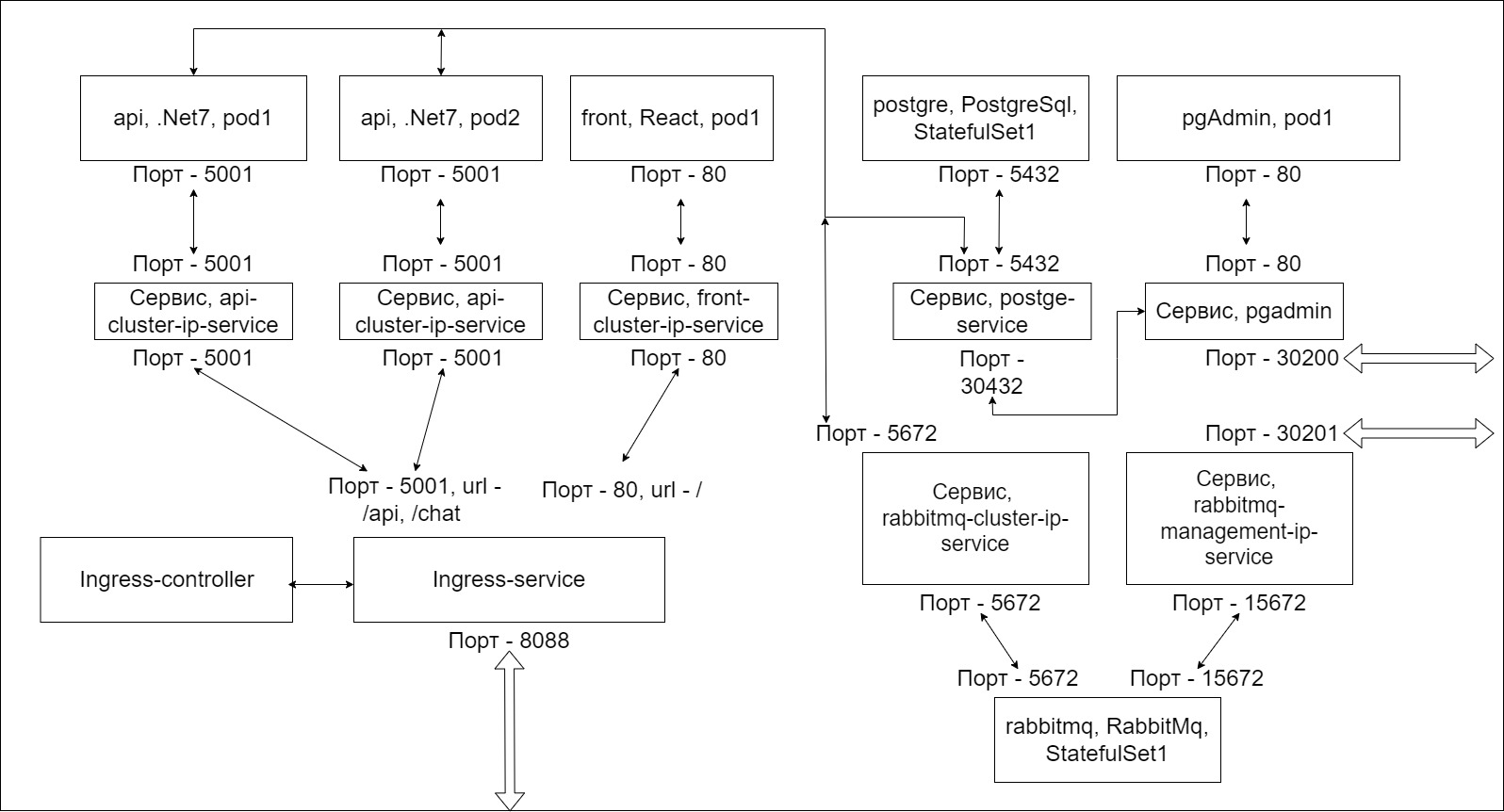
Логин -> стандартный админ для сервера Postgres -> postgres

Пароль -> то, что указали в конфигурации сервера БД -> admin

Теперь должно подключиться и отобразиться ранее восстановленная БД.

**Разворачиваем приложение на React + 2 экземпляра web-api на NetCore + сетевой балансер в кластере.**

Общая схема кластера:



1. Сервер БД уже развернут, внутренний порт (тип ports: port) – 5432, стандартный. Тип – StatefulSet, т.е. при создании нода (в случае дауна) создастся экземпляр с тем же наименованием – в нашем случае это postgre-0.
2. Разворачиваем сетевой балансер – компонент Ingress. По умолчанию – у него внешний порт – 80, но можно поставить свой, для этого в файле конфигурирования ingress-desktop.yaml в разделе type: LoadBalancer указываем внешний интерфейс -> ports: port: 8088

Запускаем : kubectl apply -f ingress-desktop.yaml

1. Конфигурируем сервис для сетевого балансера – ingress-service, файл ingress-service.yaml.

В нем есть секция с конфигурированием affinity через куки. Смысл этого механизма – обеспечить работоспособность соединения через веб-сокеты при работе нескольких экземпляров web-api. В этом случае в каждый запрос автоматически добавляется кука “reactivity\_socket”, которая перенаправляет запрос на тот под, с которым работает эта сессия фронта. Иначе, если фронт будет перенаправляться на разные поды – будет потеря коннекта через веб-сокеты.

Далее идет раздел правил. Он отвечает за перенаправление запроса на поды в зависимости от структуры url. В нашем случае -–все корневые запросы и запросы, не включающие в себя «/api\*» и «/chat\*» будут перенаправлены на под фронта, а запросы с «/api\*» и «/chat\*» - перенаправлены на поды web-api.

1. Конфигурируем web-api.

Создаем docker-образ с помощью файла Dockerfile, в нем важно указать порт работы веб-апи, в нашем случае – 5001. Остальное – стандартное. Должен быть готов репозиторий на DockerHub для хранения образа.

Команда для создания образа:

docker build -f D:\Projects\Reactivities\API\Dockerfile -t kozlovas/reactivities-api "D:\Projects\Reactivities"

- D:\Projects\Reactivities\API\Dockerfile – путь к Docker-файлу;

- kozlovas/reactivities-api – наименование репозитория в DockerHub;

- D:\Projects\Reactivities – путь к исходникам проекта для компиляции.

После создания образа – пушим его в DockerHub:

docker push kozlovas/reactivities-api

если будет ошибка доступа – заходим через cmd – docker login, в качестве пароля можно указать хеш (в самом верху файла описания докера)

Создаем поды web-api в кластере файлом api-deployment.yaml:

- количество подов – параметр replicas;

- наименование образа – параметр image;

- порт – параметр ports: containerPort, также – важно указать открытый порт в созданном образе – параметр ASPNETCORE\_URLS, иначе не заведется!

- подключение к БД – обычная строка ConnectionString, но в качестве наименования сервера – можно указать или его IP, или наименование сервиса, который относится к БД.

Также, развертываем сервис для апи в том же файле api-deployment.yaml, в сервисе только порты указываем.

Разворачиваем : kubectl apply -f api-deployment.yaml

1. Конфигурируем SPA, в нашем случае – реакт.

Создаем docker-образ с помощью Dockerfile.

Важно:

* Если в исходниках используются переменные (подключаются с помощью process.env.<наименование переменной>), то нужно включить эти переменные в описание Dockerfile.
* Если нужно иметь в образах какой-то дополнительный функционал – можно указать директиву RUN с параметрами (например, чтобы в образ был включен файловый менеджер mc -> можно выполнить run apt install mc).
* Если нужно кастомно сконфигурировать что-то, например – веб-сервер nginx, то можно заранее сделать нужный исходник и потом просто скопировать его в нужное место в файловую систему образа, например -> COPY default.conf /etc/nginx/conf.d/

В случае SPA-приложения нужно изменить исходный файл конфигурации, иначе возникает проблема с react-router-dom компонентом – при обновлении странички по не рутовому адресу – выскакивает 404 страница не найдена. Причина – nginx пытается найти эту страничку физически, а вместо этого – нужно вернуть ту же самую index.html, поэтому правим default.conf – удаляем статические маршруты и добавляем try\_files

Команда для создания образа:

docker build -f D:\Projects\Reactivities\client-app\Dockerfile -t kozlovas/reactivities-front "D:\Projects\Reactivities"

Пушим образ в DockerHub:

docker push kozlovas/reactivities-front

Создаем под для фронта с помощью имеющегося файла – front-deployment.yaml:

kubectl apply -f front-deployment.yaml

В этом файле Под и соответствующий сервис.

1. Конфигурируем сервисный брокер.

В качестве сервисного брокера используется RabbitMq.

* Загружаем образ из докера – rabbitmq:management
* Разворачиваем 3 компонента – один StatefulSet – это собственно брокер, и два сервиса – один сервис для полезной работы с апи, второй – для работы админки. Это файл rabbitmq-deployment.yaml.

Вобщем все, должно заработать, иначе смотрим логи подов и правим проблемы.