Дата последней сборки документа:
15 февраля 2024 г. в 19:30

Сборник заметок
по использованию Kubernetes в контексте машинного обучения

Содержание

1. Основные термины

под (pod) – группа *контейнеров* (один или несколько); минимальная сущность, управляемая Kubernetes; у всех контейнеров внутри одного пода общие network, IPC, UTS, PID*, namespace; pod нельзя делить между узлами кластера рис. 1; на рис. 2 приведены основные паттерны использования подов

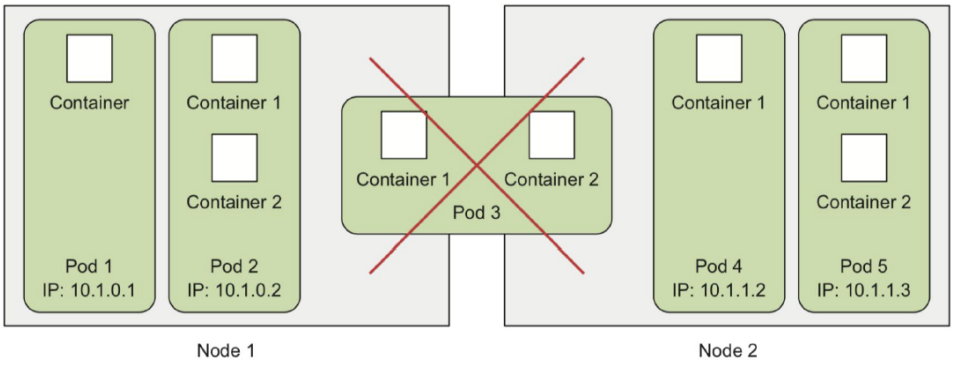

Рис. 1. Иллюстрация концепции подов

2. Общие замечания

Для небольших проектов из нескольких контейнеров удобнее использовать оркестратор Nomad https://www.nomadproject.io/.

Кластер Kubernetes состоит из набора *машин*, так называемых *узлов*, которые запускают контейнеризированные приложения. Кластер должен иметь как минимум один рабочий узел.

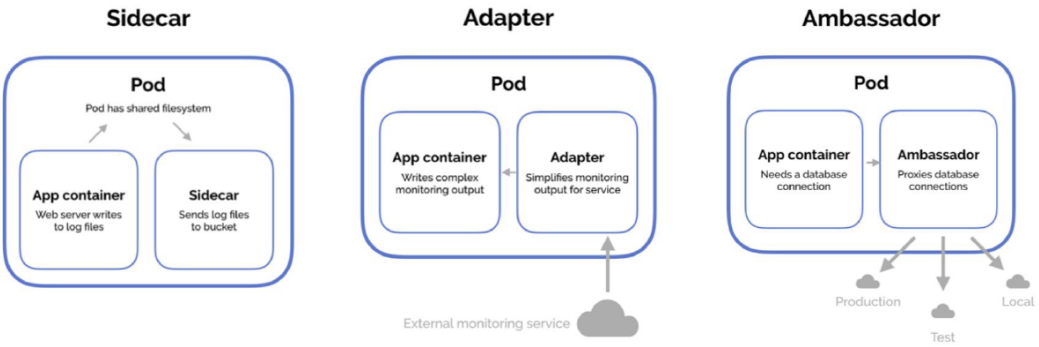

Рис. 2. Паттерны использования подов

На рабочих узлах размещены поды (pod's), являющиеся компонентами приложения. Внутренние сервисы Kubernetes управляют рабочими узлами и подами в кластере. Сервисы обычно запускаются на нескольких компьютерах, а кластер, как правило, развертывается на нескольких узлах, гарантируя отказоустойчивость и высокую надежность.

Например, в Mail Cloud Solutions топология кластеров включает в себя понятие мастер-узлов, на которых располагаются управляющие сервисы, и групп рабочих узлов, на которых запускаются приложения пользователя. Каждый кластер Kubernetes может содержать несколько групп рабочих узлов, каждая из которых создана на базе определенного шаблона виртуальной машины.

3. Начало работы в Kubernetes с помощью Minikube

Для работы с Kubernetes система должна поддреживать виртуализацию. На MacOS X это можно проверить так

```
sysctl -a | grep machdep.cpu.features | grep VMX
```

Если возвращается непустой результат, то можно продолжать. Теперь требуется установить гипервизор, например, VirtualBox https://www.virtualbox.org/wiki/Downloads.

Далее требуется установить `minikube`. На Mac OS X это можно сделать с помощью менеджера `brew`

```
brew install minikube
```

`minikube` – утилита командной строки для настройки и запуска *одноузлового кластера Kubernetes* в виртуальной машине на *локальном* компьютере. Этот вариант идеально подходит для первого знакомства с кластером под управлением Kubernetes и выполнения простых операций.

Проверка установки

```
minikube start --vm-driver=virtualbox
minikube status
```

Если кластер запущен, то в выводе команды `minikube status` должно быть что-то вроде

```
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Вместе с `minikube` устанавливается и утилита `kubectl` для работы с полноценным кластером под управлением Kubernetes.

Можно посмотреть список запущенных в кластере подов (групп контейнеров) и нод

```
kubectl get pods --all-namespaces
kubectl get nodes
```

Теперь можно запустить встроенный под `hello-minikube`. Для этого пода будет создан предварительно настроенный deployment

```
kubectl run hello-minikube --image=gcr.io/google_containers/echoserver:1.4 --port=8080 # pod/
    hello-minikube created
```

Можно снова посмотреть на актуальные списки подов

```
kubectl get pods
```

Удалить под и ноду

```
kubectl delete pod hello-minikube
kubectl delete node minikube
```

4. Процедура развертывания веб-приложения в Kubernetes с нуля

https://habr.com/ru/articles/752586/

Основные типы ресурсов:

- `Pod`: одноразовая, очень нестабильная сущность,
- `ReplicaSet`: следит за тем, чтобы было создано указанное количество подов,
- `Deployment`: упрощает процедуру обновления подов; деплоймент создает репликасет, который порождает заданное количество подов,
- `StatefulSet`: создает упорядоченный набор подов с фиксированными именами (следовательно, с адресами, по которым можно обращаться к ним внутри кластера).

В случае разворачивания простых и слабо нагруженных проектов в целом можно просто делать `docker compose up -d` на продакш-сервере и это будет работать. Но как только продакш-серверов становится больше одного, это, конечно, резко перестает быть удобно.

Поднятие кластера Kubernetes с нуля – нетривиальная задача и, как правило, удел команды DevOps. Но существует множество решений, позволяющих поднять кластер на несколько нод за небольшую плату (например, DigitalOcean).

Потребуется установить `kubectl` – это утилита командной строки для работы с кластером Kubernetes.

Все, что происходит в кластере Kubernetes, описывается через создание и изменение ресурсов разных типов. Собственно, большая часть операций `kubectl` сводится к CRUD-операциям над ресурсами.

Для удобства ресурсы рассортированы по пространствам имен (namespace). Пространство имен это тоже ресурс. Запросить список ресурсов можно так

```
$ kubectl get namespaces
# output
NAME              STATUS   AGE
default           Active   1d
kube-node-lease   Active   1d
kube-public       Active   1d
kube-system       Active   1d
```

С точки зрения пользователя кластера Kubernetes ресурсы – это просто файлы YAML/JSON. Командой `kubectl get` можно получить YAML для конкретного ресурса – например, для default

```
$ kubectl get namespace -o yaml default
# output
apiVersion: v1
kind: Namespace  # тип ресурса
metadata:
  creationTimestamp: "2022-07-25T18:00:57Z"
  labels:
    kubernetes.io/metadata.name: default
  name: default
  resourceVersion: "200"
  uid: 492e34b7-82e0-472b-9fb9-3ed7005a83eb
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
```

Поле `kind` содержит тип ресурса. `-o yaml` это формат вывода. Еще бывает полезен `-o json`

```
$ kubectl get namespace -o json default | jq .status.phase  # "Active"
# можно использовать короткие псевдонимы
$ kubectl get ns -o json default ...
```

Под – минимальная единица выполнения в Kubernetes. Под – это несколько (или даже один) Docker-контейнеров, гарантировано запускаемых на одном узле (ноде) виртуальной машины, так как поды нельзя делить между узлами кластера. В простейшем случае это один контейнер, в котором крутится сервис; другой типичный сценарий – один контейнер с сервисом и второй, например, с обработчиком логов вроде Filebeat.

Для примера создадим под со встроенным в Python HTTP-сервером

```
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
    - name: python-container
    image: python:bullseye
    command: ["python3", "-m", "http.server", "8080"]
EOF
```

Можно сохранить YAML в файл и затем вызвать

```
$ kubectl apply -f path/to/filename.yaml  # pod/test-pod created
```

Вывести список подов можно так

```
kubectl get pods
```

При создании под Kubernetes выбрал на какой ноде разместить под. Затем стянул (pull) на эту ноду указанный Docker-образ. Затем создал контейнер и запустил его.

По аналогии с `docker exec` мы можем запускать команды в контейнерах через `kubectl␣exec`, в том числе в интерактивном режиме

```
$ kubectl exec test-pod -- whoami  # root
$ kubectl exec -it test-pod -- /bin/bash
```

Проверим, что HTTP-сервер в нашем поде действительно работает, получив к нему локальный доступ

```
$ kubectl port-forward pod/test-pod 7080:8080
```

Теперь можно открыть `http://localhost:7080/` и увидеть стандартный ответ `http.server`.

Можно посмотреть логи контейнера

```
kubectl logs test-pod
```

Собираем Docker-образы

```
$ docker build --target backend -t habr-app-demo/backend:latest backend
$ docker build --target worker -t habr-app-demo/worker:latest backend
$ docker build -t habr-app-demo/frontend:latest frontend
```

Если бы у нас был настоящий продакшн-кластер Kubernetes, где-то радом с ним было бы корпоративное хранилище Docker-образов, в которое было бы достаточно их запушить. Например, если бы мы завели *хранилище* под названием `habr-app-demo-registry` в DigitalOcean, это выглядело бы так

```
# создаем псевдоним для образа habr-app-demo/backend:latest
$ docker tag \
    habr-app-demo/backend:latest \
    registry.digitalocean.com/habr-app-demo-registry/backend
$ docker push registry.digitalocean.com/habr-app-demo-registry/backend
```

Теперь можно имя `registry.digitalocean.com/habr-app-demo-registry/backend:latest` использовать как имя образа в поде.

Если мы хотим продолжить работать с `minikube`, не подключая платные облачные решения, нужно использовать `minikube docker-env`. `minikube` поднимает свой собственный Docker deamon и образы нужно собирать в нем, чтобы Kubernetes мог найти их локально

```
eval $(minikube docer-env)
docker build --target backend -t habr-app-demo/backend:latest backend
docker build --target worker -t habr-app-demo/worker:latest backend
docker build -t habr-app-demo/frontend:latest frontend
```

Поды задуманы как одноразовые, неустойчивые сущности: кластер Kubernetes может удалить любой под в любой момент. Для создания стабильного набора подов Kubernetes предоставляет другой низкоуровневый тип ресурса – `ReplicaSet`. Репликасет позволяет указать количество желаемых подов и шаблон, по которому можно клепать эти поды.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: backend-replicaset
spec:
  replicas: 2  # сколько подов нужно держать поднятыми

  # template -- шаблон для создания подов. Содержимое по синтаксису
  # аналогично описанию пода (см. пример выше), с небольшими различиями:
  # apiVersion и kind не нужно указывать -- и так понятно, что
  #   это под в той же версии API, что и репликасет;
```

```
  # также не нужно указывать name -- он будет генерироваться
  #   движком Kubernetes исходя из названия репликасета.
  template:
    metadata:
      # labels -- это произвольные key-value пары, хранящие
      # метаинформацию о поде. Выбор ключа app и значения
      # backend-app произволен. Но этот лейбл потребуется нам ниже!
      labels:
        app: backend-app

    spec:
      containers:
        - name: backend-container

          # Способ указать minikube использовать ранее собранные
          # нами локально образы. Без настройки imagePullPolicy
          # Kubernetes будет пытаться тянуть образ из интернета
          # (и, конечно, не сможет его найти и сфейлится).
          image: docker.io/habr-app-demo/backend:latest
          imagePullPolicy: Never

          ports:
          - containerPort: 40001

  # selector -- это способ для репликасета понять, какие поды
  # из числа уже существующих в кластере относятся к нему. Поскольку
  # мы прописали в шаблоне выше лейбл app: backend-app, мы точно
  # знаем, что все поды с таким лейблом порождены этим репликасетом.
  # Репликасет будет пользоваться этим селектором, чтобы понять,
  # сколько он уже насоздавал подов и сколько ещё нужно, чтобы
  # добиться количества реплик, указанного выше в поле replicas.
  selector:
    matchLabels:
      app: backend-app
```

Занимается репликасет исключительно тем, что создает и поддерживает нужное количество активных подов – рестартует поды при наличие ошибок, создает новые в случае удаления (например, в сценарии с падением ноды или если вы решите удалить один из подов руками и т.д.).

Мы могли бы создать репликасет по YAML выше и увидеть как создает два пода. Но у репликасет есть минус – неудобно обновлять поды. Нам бы хотелось, чтобы если версия бекэнда обновится, она выкатывалась постепенно, под за подом. С репликасетами же нам придется вручную создавать новый репликасет с актуальной версией и либо разом удалять старый (что под нагрузкой опасно), либо менять `replicas` в обоих репликасетах, сдувая старый и надувая новый. К счастью, есть более высокоуровневый компонент, умеющий заниматься ровно этим.

Для удобного развертывания сервисов в Kubernetes есть ресурс `Deployment`, для создания которого нужно указать все те же параметры, что и для репликасетов, плюс (опционально) настройки той самой плавной выкатки – например, какое максимальное количество подов может быть в переходном состоянии в каждый момент времени.

Поведение деплоймента в общих чертах выглядит так. При создании деплоймент создает репликасет, который порождает нужное количество подов и берется следить, чтобы их количество не менялось. Это распространенный шаблон Kubernetes – высокоуровневые абстракции создают низкоуровневые, чтобы реализовать поверх них более сложное поведение. Далее, при необхо-

димости перевыкатки деплоймент возьмет на себя создание второго репликасета и постепенное масштабирование их обоих до тех пор, пока в старом не останется подов.

```
# k8s/backend-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: backend-app
    spec:
      containers:
        - name: backend-container
        image: docker.io/habr-app-demo/backend:latest
        imagePullPolicy: Never
        ports:
          - containerPort: 40001
  selector:
    matchLabels:
      app: backend-app
```

Создадим деплоймент

```
$ kubectl apply -f k8s/backend-deployment.yaml  # deployment.apps/backend-deployment created
```

Список ресурсов

```
$ kubectl get all
# output
NAME                                    READY STATUS            RESTARTS    AGE
pod/backend-deployment-77cc555f4b-2wv24 0/1   CrashLoopBackOff 2 (19s ago) 45s
pod/backend-deployment-77cc555f4b-kd4k9 0/1   CrashLoopBackOff 2 (18s ago) 44s
pod/test-pod                            1/1   Running          0           1h

NAME                                READY UP-TO-DATE AVAILABLE AGE
deployment.apps/backend-deployment 0/2   2          0         50s

NAME                                           DESIRED CURRENT READY AGE
replicaset.apps/backend-deployment-77cc555f4b 2       2       0     47s
```

Случилось ровно то, что и ожидалось – создался деплоймент, он породил репликасет с названием, сгенерированным по названию деплоймента, а уже в рамках этого репликасета создались два пода с названиями сгенерированными по названию репликасета.

«`CrashLoopBackOff`» значит, что Kubernetes создал поды, но контейнеры в них падают с ошибкой, и делают это раз за разом. Если бы контейнер упал один раз, у него был бы статус `Error`, быстро сменяющийся рестартом.

Смотрим логи произвольного пода из деплоймента

```
$ kubectl logs deployment/backend-deployment
# output
Found 2 pods, using pod/backend-deployment-77cc555f4b-2wv24
...
pymongo.errors.ServerSelectionTimeoutError: mongo:27017: [Errno -2] Name does not resolve,
    Timeout: 1.0s, Topology Description: <TopologyDescription id: 64a319b1d732cacfbcdd43f1,
    topology_type: Unknown, servers: [<ServerDescription ('mongo', 27017) server_type: Unknown,
    rtt: None, error=AutoReconnect('mongo:27017: [Errno -2] Name does not resolve')>]>
```

```
...
```

Очевидно, не поднята база данных. Грамотно развернуть базу данных в промышленной среде – весьма нетривиальная задача, и нет общего мнения, что делать это в кластере Kubernetes – хорошая идея. Как правило, облачные провайдеры предоставляют свои SaaS-решения для разворачивания самых популярных БД и они будут куда надежнее, чем результаты самодеятельности.

Поды не очень стабильные объекты, создаваемые и удаляемые по прихоти Kubernetes-кластера. Под может быть размещен на любой подходящей ноде кластера (но этим можно управлять). При этом у пода генерируется непредсказуемый ID, и обратиться к нему извне становится сложно, как и, например, понять внутри самого пода, кто он: главная реплика базы данных, вторичная и пр.

Поэтому для stateful-сервисов Kubernetes предоставляет отдельную абстракцию – `StatefulSet`, позволяющую создавать относительно стабильный и *упорядоченный набор подов* с *фиксированными* именами (и соответственно адресами, по которым можно обращаться к ним внутри кластера) и доступом к персистентым томам.

Создадим простейший StatefulSet, поднимающий одну реплику MongoDB

```yaml
# k8s/mongodb-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongodb-statefulset
spec:
  serviceName: mongodb-service
  replicas: 1  # сколько подов требуется в стейтфулсете
  template:
    metadata:
      labels:
        # label нужен из тех же соображений, что и в деплойменте.
        app: mongodb-app
    spec:
      containers:
        - name: mongodb-container
        image: mongo:6.0.6

        # Выставляем наружу дефолтный порт монги.
        ports:
        - containerPort: 27017
        name: mongodb-cli

  # selector нужен из тех же соображений, что и в деплойменте.
  selector:
    matchLabels:
      app: mongodb-app
```

Чтобы файлы базы данных сохранялись при создании подов, нужно примонтировать персистентное хранилище. Персистентные тома в Kubernetes создаются посредством ресурса `PersistentVolumne`, в котором можно указать размер, тип хранилища (в разных облаках доступны разные типы), политику монтирования (можно ли монтировать том строго на одной ноде или на нескольких) и пр. Тома привязываются к подам посредством промежуточного ресурса `PersistentVolumeClaim` (PVC), в котором можно указать требования к тому.

Поскольку в стейфулсете в общем случае больше одного пода и чаще всего разным подам нужны отдельные тома, сейтфулсет предоставляет возможность задать шаблон PVC, по которому *на каждый созданный под будет создан свой том*

```
# k8s/mongodb-statefulset.yaml
  ...
    containers:
      - name: mongodb-container
        ...
        # Монтируем том с данными.
        volumeMounts:
          - mountPath: "/data/db"
          name: mongodb-pvc
  ...
volumeClaimTemplates:
  - metadata:
      name: mongodb-pvc
  spec:
    accessModes:
      - ReadWriteOnce # можно читать/писать только на одной ноде
    resources:
      requests:
        storage: 100Mi
```

Создаем statefulset

```
$ kubectl apply -f k8s/mogodb-statefulset.yaml  # statefulset.apps/mongodb-statefulset created
$ kubectl get pods
# output
NAME                                  READY  STATUS             RESTARTS         AGE
backend-deployment-77cc555f4b-2wv24   0/1    CrashLoopBackOff   42 (2m9s ago)    4h20m
backend-deployment-77cc555f4b-kd4k9   0/1    CrashLoopBackOff   42 (2m11s ago)   4h20m
# у пода mongodb неслучайный суффикс
mongodb-statefulset-0                 1/1    Running            0                5s
test-pod                              1/1    Running            0                4h20m
```

Обратите внимание, что вместо случайного суффикса название пода оканчивается на порядковый номер пода в стейтфулсете, то есть при пересоздании номер пода гарантированно останется таким же.

В списке персистентных томов также видим автоматически созданный том

```
$ kubectl get persistentvolumes
# output
NAME                                       CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM
                                  STORAGECLASS REASON AGE
pvc-88da024c-31ef-49f8-9b0b-3403bc3795d2 100Mi    RWO          Delete         Bound  default/
    mongodb-pvc-mongodb-statefulset-0 standard            13d
```

Теперь хорошы бы как-то проверить, что MongoDB работает; что можно к ней подключиться изнутри кластера, поделать запросы и пр. Но какой адрес у пода внутри кластера?

Для сетевого доступа к подам нам потребуется еще одна абстракция – `Service`. Описание сервиса состоит из селектора подов, к которым нужно обеспечить доступ, и тип доступа. Поскольку нам нужен внутренний доступ к одному конкретному поду в StatefulSet, нас устроит самый простой тип сервиса, так называемый headless service – без балансировки нагрузки и выделения статического IP для доступа извне

```
# k8s/mongodb-service.yaml
apiVersion: v1
```

```
kind: Service
metadata:
  name: mongodb-service
spec:
  # ClusterIP -- самый простой тип сервиса, который
  # позволяет подам связываться друг с другом в рамках
  # кластера, но абсолютно никак не виден снаружи:
  type: ClusterIP
  clusterIP: None
  selector:
    app: mongodb-app
```

```
$ kubectl apply -f k8s/mongodb-service.yaml
```

Теперь для доступа к поду с БД внутри кластера должен заработать URL

```
$ <pod name>.<service name>.<namespace>.svc.cluster.local
```

То есть в нашем случае

```
statefulset-0.mongodb-service.default.svc.cluster.local
```

Проверим, зайдя в тестовый под и попытавшись подключиться через pymongo

```
$ kubectl exec -it test-pod -- /bin/bash
/# python -m pip install pymongo
...
>>> import pymongo
>>> uri = "mongodb://mongodb-statefulset-0.mongodb-service.default.svc.cluster.local"
>>> c = pymongo.MongoClient(uri)
>>> c.some_database.some_collection.insert_one({"foo": "bar"})
>>> list(c.some_database.some_collection.find({}))  # Работает!
```

Проверив поды бекэнда, мы заметим, что они все еще в `CharshLoopBackOff`, поскольку ожидают, что хостнейм монги – `mongo`, как это было раньше, в решении с Docker Compose. Менять файл `k8s/backend-deployment.yaml` и пересоздавать деплоймент целиком не очень удобно; давайте внесем точечное изменение командой `kubectl patch`

```
# k8s/backend-deployment-patch.yaml
spec:
  template:
    spec:
      containers:
        - name: backend-container
        env:
          - name: APP_ENV
          value: k8s
```

```
$ kubectl patch deployment backend-deployment \
    --patch-file k8s/backend-deployment-patch.yaml
```

Удобной альтернативой `kubectl patch` может быть `kubectl edit`, а также в этом конкретном случае настройки переменной окружения

```
$ kubectl set env deployment/backend-deployment APP_ENV=k8s
```

Смотрим ресурсы

```
$ kubectl get pods
# output
```

```
NAME                                  READY   STATUS    RESTARTS   AGE
backend-deployment-7547fb8b7c-4k2x7   1/1     Running   0          11s
backend-deployment-7547fb8b7c-hhvhd   1/1     Running   0          8s
...
```

Для хранения паролей и прочей приватной информации Kubernetes предоставляет еще один тип ресурсов – `Secret`. Генерируем пароль и создаем секрет

```
echo -n "$(openssl rand -hex 14)" > password.txt
$ kubectl create secret generic mongodb-secret --from-file password.txt
```

В данном случае мы создали секрет, который можно будет подключать к контейнерам как раздел, в который будут подтягиваться исходные файлы.

Поправим конфигурацию MongoDB, чтобы у пользователя root был сгенерированный нами пароль. Конкретно в случае MongoDB это можно сделать, поправив переменную окружения `MONGO_INITDB_ROOT_PASSWORD_FILE`. Также нам нужно будет примонтировать раздел с файлом `password.txt`

```
# k8s/mongodb-statefulset-v2.yaml
...
  - env:
    ...
      - name: MONGO_INITDB_ROOT_PASSWORD_FILE
      value: "/run/secrets/mongodb/password.txt"
    ...

    # В разделе volumeMounts мы описываем, какие тома
    # по каким путям нужно примонтировать. Тома могут
    # иметь разное происхождение (например, сейчас у нас
    # один персистентный том и один томик с секретами).
    # Описание того, какого типа какой том, вынесено в
    # отдельное поле -- volumes (ниже).
    volumeMounts:
      - mountPath: "/data/db"
        name: mongodb-pvc
      - mountPath: "/run/secrets/mongodb"
        name: mongodb-secret-volume
        readOnly: true
    ...
    # В разделе volumes мы описываем тома, которые нужно
    # примонтировать в соответствии с инструкциями в поле
    # volumeMounts.
    volumes:
      - name: mongodb-secret-volume
        # Указываем, что данные для этого тома надо взять
        # из ресурса Secret с названием mongodb-secret.
        secret:
          secretName: mongodb-secret
          optional: false
      # А персистентный том не надо описывать -- Kubernetes
      # сделает это за нас для всех подов в стейтфулсете.
...
```

Поскольку пароль берется во внимание только при инициализации базы данных с нуля, нам проще всего полностью удалить стейтфулсет и все данные и создать заново с новыми настройками (НЕ ПОВТОРЯТЬ на ПРОДЕ!!!)

```
# Удаляем стейтфулсет (а с ним и под)
```

```
$ kubectl delete statefulset mongodb-statefulset
# statefulset.apps "mongodb-statefulset" deleted

# Удаляем PersistentVolumeClaim, чтобы Kubernetes разрешил удалить сам том
$ kubectl delete pvc mongodb-pvc-mongodb-statefulset-0
# persistentvolumeclaim "mongodb-pvc-mongodb-statefulset-0" deleted

$ kubectl get persistentvolumes
# No resources found

# Пересоздаём всё созданием нового стейтфулсета
$ kubectl apply -f k8s/mongodb-statefulset-v2.yaml
# statefulset.apps/mongodb-statefulset created
```

Если сейчас перезапустить поды бекэнда (например, командной `kubectl rollout restart deployment`), увидим, что они снова попали в `CrashLoopBackOff` – так как теперь нужен пароль. Нужно обновить `APP_ENV` и примонтировать секрет

```
# k8s/backend-deployment-patch-2.yaml
spec:
  template:
    spec:
      containers:
        - name: backend-container
          env:
            - name: APP_ENV
            value: k8s_secrets
          volumeMounts:
            - mountPath: "/run/secrets/mongodb"
              name: mongodb-secret-volume
              readOnly: true
      volumes:
        - name: mongodb-secret-volume
          secret:
            secretName: mongodb-secret
            optional: false
```

Применяем патч и видим, что вновь созданные поды работают без проблем

```
$ kubectl patch deployment backend-deployment \
    --patch-file k8s/backend-deployment-patch-2.yaml
# deployment.apps/backend-deployment patched
$ kubectl get pods
# output
NAME                                  READY  STATUS   RESTARTS  AGE
backend-deployment-775d8c8b8f-m686f   1/1    Running  0         9s
backend-deployment-775d8c8b8f-zghvf   1/1    Running  0         6s
...
```

Бекэнд и база заработали – давайте вспомним про остальные сервисы (фронтенд, воркер и очередь задач для него) и быстренько поднимем их

```
$ kubectl apply -f k8s/frontend-deployment.yaml
$ kubectl apply -f k8s/worker-deployment.yaml
$ kubectl apply -f k8s/redis.yaml
```

Теперь время настроить внешний доступ к нашим двум сервисам – фронтэнду, отдающему пререндеренные страницы и статические фалы, и бекэнду, предоставляющему JSON API. Kubernetes предоставляет много способов выставить наружу доступ к крутящимся в кластере сервисам, но простейшая схема, подходящая для сурового прода, выглядит так.

Доступ в кластер *извне* осуществляется созданием ресурса `Service` типа `LoadBalancer`

```
apiVersion: v1
kind: Service
metadata:
    name: ...
spec:
    type: LoadBalancer
    selector:
        ...
```

Если сделать это в настоящем Kubernetes-кластере (развернутом, например, в AWS или DigitalOcean), облачный провайдер выделит *статический* IP и создаст некий (имплементация завист от провайдера) *балансировщик нагрузки*, который будет обслуживать это IP и распределять трафик между подами, подходящими под указанный селектор.

Статические IP стоят денег, а функциональности непрозрачного проприетарного балансировщика может не хватить для многих задач, поэтому при наличии множества сервисов (в нашем случае аж двух) обычно поднимается один сервис типа LoadBalancer, который уже роутит трафик между сервисами. Для этого можно использовать готовые балансировщики нагрузки – например, `nginx` или `traefik`.

Роутинг трафика внутри кластера осуществляется созданием ресурсов `Service` типов `ClusterIP` или `NodePort`, ресурсов `Ingress` и установкой в кластер ингресс-контроллера.

`Service` – это ресурс, инкапсулирующий, собственно, сервис – совокупность подов, реализующих какой-то один сетевой интерфейс; например, HTTP или gRPC API. В правилах роутинга `Service` выступает в роли цели, куда роутить запросы.

`Ingress` – ресурс, инкапсулирующий правило роутинга. Если вы работаете с nginx, неплохой аналог ингресса – сайт в папке /etc/nginx/conf.d/sites-enabled. Так будет выглядеть ингресс для доступа к сервису `frontend-sevice`

```
# k8s/frontend-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
  annotations:
    # Аннотации позволяют настроить поведение
    # ингресс-контроллера и, конечно, зависят от
    # того, какой именно мы взяли -- сейчас это nginx:
    nginx.ingress.kubernetes.io/from-to-www-redirect: "true"
spec:
  rules:
    # Правила роутинга представляют собой ровно то,
    # что можно ожидать -- хост, протокол, пути, на какой
    # сервис (и какой порт) перенаправить трафик:
    - host: frontend.localhost
      http:
        paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: frontend-service
              port:
                number: 40002
```

Ингресс-контроллер – это компонент, следящий за существующими в кластере ингрессами и, собственно, реализующий роутинг. Процесс установки и настройки ингресс-контроллера различается в зависимости от его выбора, и чтобы не вдаваться в лишние детали, давайте остановимся на самом простом варианте, для установки которого в `minikube` есть готовая документация – `nginx`.

Устанавливаем контроллер

```
$ minikube addons enable ingress
```

Логика выбора подов для сетевого взаимодействия инкапсулирована в ресурсах типа `Service`, поэтому следующим шагом создаем сервисы

```
# k8s/frontend-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  # NodePort -- один из встроенных типов сервисов
  # в Kubernetes. Его выбор сейчас обусловлен особенностями
  # реализации minikube, в нормальном кластере можно
  # было бы использовать более простой type: ClusterIP.
  type: NodePort
  selector:
    app: frontend-app
  ports:
    - protocol: TCP
    port: 40002
```

```
$ kubectl apply -f k8s/frontend-service.yaml
$ kubectl apply -f k8s/backend-service.yaml
```

Теперь создаем ингрессы

```
$ kubectl apply -f "k8s/*-ingress.yaml"
```

Поскольку ингресс опирается на имя хоста для роутинга между двумя сервисами (здесь используются хосты `frontend.localhost` и `backend.localhost`), нам также придется прописать эти хосты в `/etc/hosts`, чтобы все заработало

```
$ echo "127.0.0.1 frontend.localhost\n127.0.0.1 backend.localhost" | sudo tee -a /etc/hosts
```

Теперь наши отладочные хосты указывают на 127.0.0.1. Осталось запустить `minikube tunnel`, чтобы прорезать доступ

```
$ minikube tunnel
```

Чтобы увидеть содержательный результат, осталось создать тестовый контент

```
$ kubectl exec deploy/backend-deployment -- python -m tools.add_test_content
```

Теперь можно зайти на `http://frontend.localhost/card/helloworld` и увидеть, что все работает.

Рекомендации по отладке. Начинать всегда стоит с ознакомления со списком ресурсов.

`kubectl get pods` покажет, какие поды запущены, в каком они статусе и, что немаловажно, сколько было рестартов каждого пода; частые рестарты могут указывать на систематические проблемы.

Можно смотреть логи произвольного пода как в деплойменте, так и в стейтфулсете

```
$ kubectl logs deployment/backend-deployment
$ kubectl logs statefulset/mongodb-statefulset
```

Также можно смотреть логи конкретных подов (откроется лог первого контейнера), логи конкретного контейнера и логи всех контейнеров

```
$ kubectl logs backend-deployment-775d8c8b8f-m686f
$ kubectl logs -c backend-container backend-deployment-775d8c8b8f-m686f
$ kubectl logs --all-containers backend-deployment-775d8c8b8f-m686f
```

Если под упал и уже пытается перезапуститься, логов падения вы не увидите. Для просмотра логов предыдущего запуска можно добавить флаг `--previous`

```
$ kubectl logs --previous backend-deployment-775d8c8b8f-m686f
```

Распростаненная проблема – падение пода по ошибке out of memory (OOM). Тут логах ничего не будет видно и стоит смотреть события через `kubectl describe`

```
$ kubectl describe pod oom-pod
```

Иногда бывает полезно посмотреть все события кластера в хронологическом порядке

```
$ kubectl get events --sort-by='.metadata.creationTimestamp' -A
```

В случае проблем с роутингом имеет смысл посмотреть логи ингресс-контроллера. У нас он живет в пространстве имен `ingress-nginx`

```
$ kubectl -n ingress-nginx get pods
# output
NAME                                        READY STATUS    RESTARTS    AGE
ingress-nginx-admission-create-wd4j2        0/1   Completed 0           3d
ingress-nginx-admission-patch-jfjrg         0/1   Completed 0           3d
ingress-nginx-controller-6cc5ccb977-s7659 1/1   Running   1 (2d ago) 3d
$ kubectl -n ingress-nginx logs deployment/ingress-nginx-controller
...
```

Для работы с ресурсами вне дефолтного пространства имен надо указывать флаг `-n␣<namespace>`, иначе `kubectl get` будет говорить, что такого ресурса не существует.

Если мы меняем что-то в деплойменте — например, версию Docker-образа при выкатке нового релиза, — Kubernetes должен создать новые поды по новому шаблону и удалить старые. По умолчанию это будет делаться постепенно — Kubernetes начнёт гасить 25% старых подов и создавать соответствующее количество новых, по мере успешного запуска новых продолжая гасить старые. Число в 25%, конечно, настраивается, причём с обеих сторон — какой процент подов от желаемого числа может быть в переходном состоянии (`maxUnavailable`) и какое количество лишних подов может существовать в моменте (`maxSurge`)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
spec:
  replicas: 10
  strategy:
    rollingUpdate:
      maxUnavailable: "10%"
      maxSurge: "50%"
...
```

Мы можем уменьшить `maxUnavailable`, чтобы обезопасить себя от отключения слишком большого числа подов разом (например, если по какой-то причине мы любим, чтобы все поды работали на пределе своих возможностей); можем уменьшить `maxSurge`, чтобы во время выкатки система не была перегружена подами (если у нас 100 подов, лишние 25 могут быть проблемой); или можем увеличить `maxUnavailable` и/или `maxSurge`, чтобы ускорить выкатку.

Помимо значений в процентах можно указывать абсолютное количество подов. Например, самая медленная и безопасная выкатка будет настроена как-то так:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
spec:
  replicas: 10
  strategy:
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
...
```

При такой настройке Kubernetes будет аккуратно добавлять по одному поду, дожидаться, пока новый под будет готов принимать нагрузку, и только после этого удалять один из оставшихся старых подов. Что, конечно, будет чрезвычайно медленно.

Для определения, запустился ли успешно контейнер, используется так называемая liveness-проба. Это некоторое действие, которое Kubernetes-движок может совершать с контейнером — например, запуск команды, HTTP- или gRPC-запрос, — и проверять, что получается ожидаемый результат. Если мы делаем обычный HTTP-сервис, нам идеально подходит HTTP-запрос. Конфигурируется он так:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
  - name: python-container
  image: python:bullseye
  command: ["python3", "-m", "http.server", "8080"]

  livenessProbe:
    httpGet:
      path: /ping
      port: 8080
      httpHeaders:
        - name: X-Request-Reason
        value: liveness-probe
    initialDelaySeconds: 5
    periodSeconds: 3
    failureThreshold: 2
```

Всё предельно просто: мы указываем порт, путь, дополнительные HTTP-заголовки (если они нужны) и три параметра: сколько ждать после запуска контейнера перед первой попыткой сделать пробу (`initialDelaySeconds`), как часто делать пробу, убеждаясь, что под ещё работает (`periodSeconds`), и сколько раз подряд проба должна завершиться неуспехом, чтобы под был убит (`failureThreshold`). С настройками выше Kubernetes подождёт пять секунд после созда-

ния контейнера, после чего начнёт каждые три секунды делать GET-запрос пути `/ping`. Если GET-запрос два раза подряд вернёт статус, отличный от 200 (или стаймаутится), под будет убит и пересоздан.

Liveness-проба совершается не только после запуска, но и всё время жизни пода — так Kubernetes защищает нас от возможных дедлоков и других проблем, решающихся рестартом контейнеров.

Для *автоматического горизонтального масштабирования* кластера, необходимо создать ресурс `HorizontalPodAutoscaler`

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: backend-hpa
spec:

  # Указываем, какую совокупность подов надо скейлить:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend-deployment

  # Нижний и верхний предел количества реплик,
  # чтобы случайно не сожрать весь кластер:
  minReplicas: 2
  maxReplicas: 20

  # Метрики, по значениям которых будем определять
  # потребность в масштабировании:
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 75
    - type: Resource
      resource:
        name: memory
        target:
          type: AverageValue
          averageValue: 400Mi
```

В этом примере автоскейлер сконфигурирован так, чтобы средняя загрузка CPU по всем работающим подам не превышала 75% (процент считается относительно requests), а среднее потребление памяти — 400 МБ. Автоскейлер будет с какой-то периодичностью проверять метрики подов и при необходимости скейлить деплоймент без ручного вмешательства.

Список литературы

1. *Джуба С.*, *Волков А.* Изучаем PostgreSQL 10. – М.: ДМК Пресс, 2019. – 400 с.

Листинги