Лабораторная работа 11

Kubernetes

Теоретическая часть

Вступление

Мир узнал о Kubernetes в 2014 году, когда корпорация Google опубликовала исходные коды проекта под названием “Project 7”, которое отсылало к персонажу известного и популярного в то время сериала “Star Trek”. Со временем у проекта появилось то название, позаимствованное из греческого языка - Kubernetes.

Изначально проект задумывался исключительно для нужд самой компании и разрабатывался под влиянием другого проекта Google - Google Borg, системы управления кластерами. Выпустив первый релиз Kubernetes, корпорация решила сделать его своим вкладом в развитие ИТ. Поэтому в 2015 году спустя год после публикации исходных кодов проекта, Google и Linux Foundation организовали Cloud Native Computing Foundation (CNCF), которому были переданы права на Kubernetes.

До появления Kubernetes разработка и управление приложениями были далеко не такими, какими мы их знаем сегодня. Изначально все приложения строились как монолиты — большие, монолитные структуры, где все компоненты тесно связаны друг с другом. Эти приложения разворачивались на выделенных физических серверах, и каждый сервер обслуживал только одно приложение. На первый взгляд, это казалось удобным, но на практике приводило к множеству проблем, с которыми разработчикам и администраторам приходилось постоянно сталкиваться.

Основной трудностью было управление ресурсами. Серверы либо оставались не загруженными, либо, наоборот, перегружались. Например, если серверу было выделено больше ресурсов, чем нужно, большая часть этих ресурсов просто простаивала. Но если ресурсов не хватало, то приложение начинало «захлебываться» от нагрузки. Гибкости в управлении не было, а добавление новых серверов или перераспределение нагрузки требовало больших усилий. Всё это приводило к неэффективности, лишним затратам и, что самое главное, ограничивало возможности масштабирования.

Кроме того, монолитная структура приложений создавала серьёзные сложности **при их обновлении**. Чтобы внести изменения в один из компонентов, нужно было пересобирать и тестировать всё приложение целиком. Такой подход увеличивал время разработки, замедлял выпуск новых версий и добавлял рисков: любое изменение могло затронуть другие части системы, вызывая неожиданные сбои.

Ситуация стала меняться с появлением **микросервисной архитектуры**. В отличие от монолитов, приложения начали делить на небольшие, независимые компоненты — микросервисы. Каждый микросервис решал свою задачу, мог разрабатываться и обновляться отдельно. Это дало огромные преимущества:

ускорились релизы, стало проще масштабировать отдельные части приложения, а сбои в одном микросервисе больше не затрагивали всю систему. Однако возникли новые вызовы. Управлять множеством микросервисов вручную оказалось ещё сложнее. Представьте: десятки или даже сотни компонентов, каждый из которых нужно развернуть, настроить, обновить и обеспечить их взаимодействие друг с другом. Это требовало автоматизации.

Именно здесь на сцену вышел Kubernetes. Он стал ответом на все эти проблемы. Kubernetes появился как инструмент для управления контейнеризированными приложениями. Контейнеризация позволила упаковывать приложения вместе со всеми их зависимостями в изолированные среды, а Kubernetes взял на себя задачу оркестрации — автоматического управления этими контейнерами.

С Kubernetes всё стало проще. Теперь разработчики могли развертывать свои приложения не на отдельных серверах, а в виде контейнеров, которые Kubernetes распределял по доступным ресурсам. Если нагрузка на приложение возрастала, Kubernetes автоматически масштабировал его, добавляя новые экземпляры контейнеров. Если что-то выходило из строя, Kubernetes сам перезапускал упавшие контейнеры, обеспечивая стабильность системы. Обновления тоже перестали быть проблемой: с помощью Kubernetes можно было обновлять отдельные компоненты приложения, не затрагивая другие.

Кроме того, Kubernetes оказался невероятно гибким. Он одинаково хорошо работал как на локальных серверах, так и в облаке, а также поддерживал гибридные решения, где часть инфраструктуры находилась в дата-центре компании, а часть — в облаке. Это сделало его идеальным выбором для современных распределённых систем.

Как работает Kubernetes

Kubernetes — это портативная расширяемая платформа с открытым исходным кодом для управления контейнеризованными рабочими нагрузками и сервисами, которая облегчает как декларативную настройку, так и автоматизацию.

- Node или узлы – это базовая вычислительная единица, которой обычно является одна машина.
- Master node или управляющий узел – узел, на котором расположены элементы, отвечающие за управление кластером.
- Worker node или рабочий узел – узел, на котором запускаются контейнеризованные приложения. Данный узел позволяет получить пользователям доступ к самому приложению, работающему на кластере.
- Pods или поды – это группа из одного или нескольких контейнеров с общим хранилищем и сетевыми ресурсами, а также спецификацией для запуска контейнеров

Кластер Kubernetes состоит из уровня управления и набора рабочих машин, называемых узлами, которые запускают контейнеризованные приложения. Каждому кластеру требуется как минимум один рабочий узел для запуска подов.

Архитектура управляющего узла

Обычно в кластере используется группа управляющих узлов и несколько рабочих узлов. Задача управляющих узлов – следить за состоянием кластера, а также предоставлять возможность взаимодействия с ним инженера. Для выполнения своих задач на управляющем узле разворачиваются следующие компоненты слоя управления:

- **API Server** – это центральный компонент Kubernetes, который отвечает за обработку всех внутренних и внешних запросов, взаимодействующих с кластером. Например, API Server является единой точкой входа для команд, отправленных инженером, производя их валидацию, аутентификацию и авторизацию запросившего.
- **Scheduler или планировщик** – отвечает за распределение подов по рабочим узлам в кластере. Основная задача планировщика – размещение подов, учитывая доступные ресурсы на узлах, требования каждого пода и различные другие факторы.
- **Controller Manager** – это компонент, который управляет различными контроллерами (controllers), отвечающими за поддержание желаемого состояния объектов кластера.
- **etcd** – распределенное и высоконадежное хранилище данных в формате "ключ-значение", которое используется как основное хранилище всех данных кластера в Kubernetes.

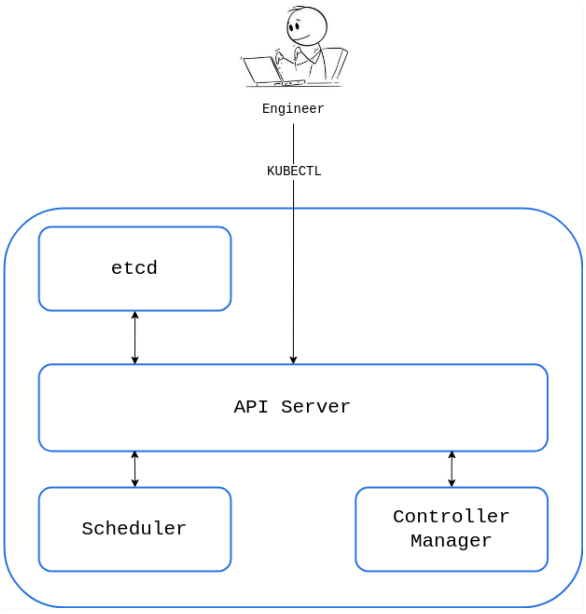

Рисунок 2. Устройство управляющего узла.

Архитектура рабочего узла

Все объекты, толерантные к метке controlplane, которые развертываются в кластере Kubernetes, размещаются на рабочих узлах. Рабочие узлы являются основными компонентами кластера и состоят из нескольких ключевых элементов, которые обеспечивают функционирование приложений.

- **Kubelet** – это агент, который запускается на каждом рабочем узле в Kubernetes-кластере. Он отвечает за управление подами. Основная задача – обеспечивать, чтобы состояние подов на узле соответствовало описанному в спецификациях рабочей нагрузки. Kubelet отслеживает состояние подов и контейнеров, а также выполняет команды, полученные от мастер-узла.
- **Container Runtime** – это элемент, отвечающий за запуск и управление жизненным циклом контейнеров в Kubernetes. Container Engine может быть одним из нескольких вариантов, таких как Docker, containerd. Container Engine обеспечивает создание, запуск и остановку контейнеров, а также управление их ресурсами.
- **Kube Proxy** - компонент Kubernetes, который обрабатывает сетевой трафик маршрутизации для служб в кластере. Kube Proxy позволяет микросервисам общаться друг с другом, даже если они находятся на разных узлах. Kube Proxy обеспечивает прозрачную маршрутизацию трафика между службами в кластере.Этот компонент может отсутствовать, если его функции берет на себя плагин CNI
- **Pod** - минимальная единица измерения в Kubernetes, которая содержит в себе один или более контейнеров. Под обеспечивает общее пространство имен для контейнеров, что позволяет им общаться друг с другом. Пода также обеспечивает возможность масштабирования и управления контейнерами как единым целым. Например, на рисунке 5 в поде представлено два контейнера, приложение и сборщик логов с приложения (сайдкар).

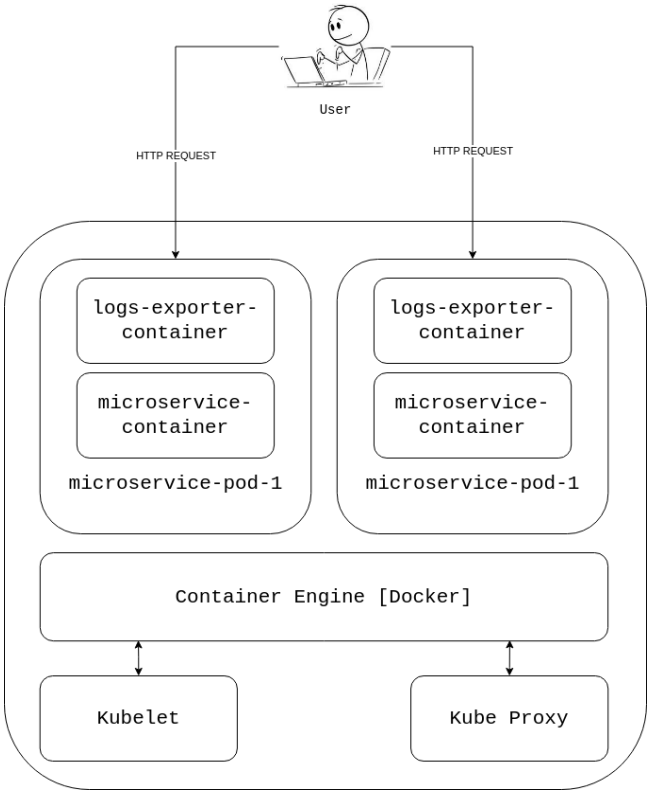

Рисунок 3. Архитектура рабочего узла

Установка локального кластера Kubernetes

Развертывание кластера на локальной машине является очень дорогим мероприятием как с точки зрения трудозатрат, так и с точки зрения требования к ресурсам. Таким образом, чтобы не нагружать локальный компьютер сложными вычислениями, можно установить специально созданный для этого инструмент minikube, а также клиент для управления кластером kubectl.

Установка Minikube с помощью прямой ссылки:

```
curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 \
  && chmod +x minikube
sudo mkdir -p /usr/local/bin/
sudo install minikube /usr/local/bin/
```

Для того, чтобы запустить minikube, нужно прописать следующую команду:

```
minikube start
```

Также стоит установить клиентский модуль для kubernetes, который ставится с помощью следующей команды:

```
curl -LO https://dl.k8s.io/release/`curl -LS https://dl.k8s.io/release/stable.txt`/bin/linux/amd64/kubectl
curl -LO https://dl.k8s.io/release/v1.32.0/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
```

```
kubectl version --client
```

Далее, чтобы проверить работоспособность кластера, можно отправить тестовый запрос, который покажет доступные узлы.

```
    kubectl get nodes
NAME       STATUS   ROLES          AGE   VERSION
minikube   Ready    control-plane   15d   v1.30.0
```

В ответ minikube отправит краткую информацию о кластере, где главным показателем его работоспособности будет являться статус Ready.

Создание stateless-манифестов Kubernetes

В Kubernetes объекты создаются в декларативном стиле, поэтому инженер описывает желаемое состояние объекта, а не конкретные шаги, необходимые для его создания.

Для этого инженер пишет код манифеста в формате YAML, который является текстовым форматом для описания данных. Манифест содержит описание объекта, включая его свойства, конфигурацию и зависимости.

Самым низкоуровневым манифестом в Kubernetes является под, он лежит в основе почти всех более высокоуровневых манифестов. Для того, чтобы создать под, нужно прописать в файле pod.yaml следующую конфигурацию.

```
    kubectl create namespace test
    kubectl config set-context --current --namespace=test
```

```
    ---
    apiVersion: v1
    kind: Pod
    metadata:
     name: nginx
    spec:
     containers:
     - name: nginx
       image: nginx:1.14.2
       ports:
       - containerPort: 80
```

Далее можно создать объект и просмотреть его состояние с помощью команд:

```
    kubectl create -f pod.yaml
pod/nginx created

    kubectl get pods
NAME   READY   STATUS    RESTARTS  AGE
nginx  1/1     Running   0         50s
```

Индикатором готового работать пода является стутус, равный Running и готовность 1/1, а готовность 0/1. Единицей справа от слеша является количество готовых к работе контейнеров.

Однако пода никогда не используется как полноценная единица развертывания, так как если прописать команду удаления поды.

```
kubectl delete pod nginx
```

То она удалится и больше не появится. Таким образом пода никак не позволяет развернутому приложению быть отказоустойчивым, поэтому можно воспользоваться более высокоуровневым манифестом под названием ReplicaSet.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
 name: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
     - name: nginx
       image: nginx:1.27
```

Теперь если мы создадим этот объект и попробуем удалить любой из под nginx, то ReplicaSet создаст новый взамен удалённому, так как у него запись в конфигурации – должно быть запущено ровно 3 поды с меткой nginx.

```
kubectl get pods
NAME       READY  STATUS   RESTARTS     AGE
nginx-vkrbh 1/1    Running  0          32s
nginx-vzscz 1/1    Running  0          32s
nginx-xpz6n 1/1    Running  0          32s

kubectl delete pod nginx-vkrbh
pod "nginx-vkrbh" deleted

kubectl get pods
NAME       READY   STATUS    RESTARTS     AGE
nginx-vzscz 1/1     Running   0          3m14s
nginx-wsvzs 1/1     Running   0          36s
nginx-xpz6n 1/1     Running   0          3m14s
```

Также хочется отметить, что поды, созданные с помощью ReplicaSet имеют в своем составе дополнительный набор символов – это уникальный идентификатор пода, который добавляется автоматически для того, чтобы не было коллизий названий под.

Однако и у ReplicaSet есть один существенный недостаток – невозможность автоматически обновлять версию контейнера на подах. Например, поменяем версию nginx с 1.14.2 до 1.15.0 и применим манифест

```
    kubectl apply -f replicaset.yaml
replicaset.apps/nginx configure

    kubectl get podes
NAME       READY   STATUS    RESTARTS     AGE
nginx-vzscz 1/1     Running   0          8m21s
nginx-wsvzs 1/1     Running   0          5m43s
nginx-xpz6n 1/1     Running   0          8m21s

    kubectl describe pod nginx-vzscz
...
Containers:
  nginx:
    Container ID: ...
    Image: nginx:1.14.2
    ...
```

Мы увидим, что версия nginx так и не поднялась, однако если удалить текущую поду, то она поднимется с уже обновлённой версией образа.

```
    kubectl delete pod nginx-vzscz
    pod "nginx-vzscz" deleted

    kubectl get po
    NAME       READY   STATUS    RESTARTS     AGE
    nginx-jhgrc 1/1     Running   0          37s
    nginx-wsvzs 1/1     Running   0          10m
    nginx-xpz6n 1/1     Running   0          13m

    kubectl describe pod
    ...
    Containers:
     nginx:
       Container ID: ...
       Image: nginx:1.15.0
    ...
```

Вместо автоматического обновления, инженеру придется вручную удалять текущие поды, чтобы обновить версию образа. Однако на такой случай есть еще более высокоуровневый манифест Deployment, который не имеет данного недостатка и является самым популярным манифестом для развертывания stateless-приложений.

```
    ---
    apiVersion: apps/v1
    kind: Deployment
    metadata:
```

```
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
     - name: nginx
       image: nginx:1.14.2
       ports:
       - containerPort: 80
```

Теперь, если повторить ситуацию, описанную выше с ReplicaSet, то Deployment автоматически удалит текущие поды, и поднимет новые с обновленной версией nginx.

```
kubectl create -f deployment.yaml
deployment.apps/nginx-deployment created

kubectl get podes
NAME      READY   STATUS    RESTARTS     AGE
nginx-... 1/1     Running   0            3m23s
nginx-... 1/1     Running   0            3m23s
nginx-... 1/1     Running   0            3m23s

vim deployment.yaml //изменяем версию образа nginx

kubectl apply -f deployment.yaml
deployment.apps/nginx-deployment configure

kubectl get podes
NAME      READY   STATUS    RESTARTS     AGE
nginx-... 1/1     Running   0            21s
nginx-... 1/1     Running   0            23s
nginx-... 1/1     Running   0            26s

kubectl describe pod
...
Containers:
 nginx:
   Container ID: ...
   Image:nginx:1.15.0
...
```

Также хочется отметить, что теперь имена подов поменялись. Для того, чтобы узнать принцип построения названия под для Deployment, можно заглянуть в официальную документацию Kubernetes.

Однако после развертывания под, например, с помощью Deployment получить доступ извне к nginx не получится, для этого есть ещё один манифест, который называется Service. Его задача предоставить возможность отправки трафика извне на поды, указанные в селекторе.

```
---
apiVersion: v1
kind: Service
metadata:
 name: nginx-service
spec:
 selector:
   app: nginx
 ports:
   - protocol: TCP
     port: 80
     targetPort: 8000
```

Теперь нужно применить данный манифест и настроить проброс портов из minikube.

```
kubectl apply -f service.yaml
kubectl port-forward svc/nginx-service 8000:80
curl -XGET http://localhost:8000/
```

После отправки HTTP запросов поды nginx должны отвечать ответом 200.

Создание statefull-манифестов Kubernetes

Манифест Deployment имеет одну особенность, он не способен сохранять информацию о предыдущих состояниях или сеансах. Каждый запрос к подам такого манифеста рассматривается, как отдельное изолированное взаимодействие. Однако, когда речь идёт о таких программных продуктах, как базы данных, брокеры сообщений stateless-манифест не будет способен восстановить всю информацию, которая была сохранена до перезапуска. Stateful приложения требуют в первую очередь возможность сохранения информации о предыдущих состояниях или взаимодействиях с клиентами.

Для того, чтобы разворачивать stateful-приложения в Kubernetes потребуются абсолютно другие манифесты, в основе которых также лежат поды, однако со своими особенностями.

Перед тем, как начать разворачивать новые манифесты, нужно ознакомиться с особенностями их построения. Самым высокоуровневым stateful-манифестом в ванильном Kubernetes является StatefulSet, схема устройства которого представлена на рисунке ниже. В основе stateful-приложения лежат следующие манифесты.

- PersistentVolume – определяет часть хранилища в кластере.

- PersistentVolumeClaim – определяет запросы на хранение данных. По аналогии под потребляет ресурсы узла, а PersistentVolumeClaim – ресурсы PersistentVolume.
- StatefulSet – аналог Deployment. Отличается тем, что требует не только развертывания под, но также и места, куда будут складываться данные.

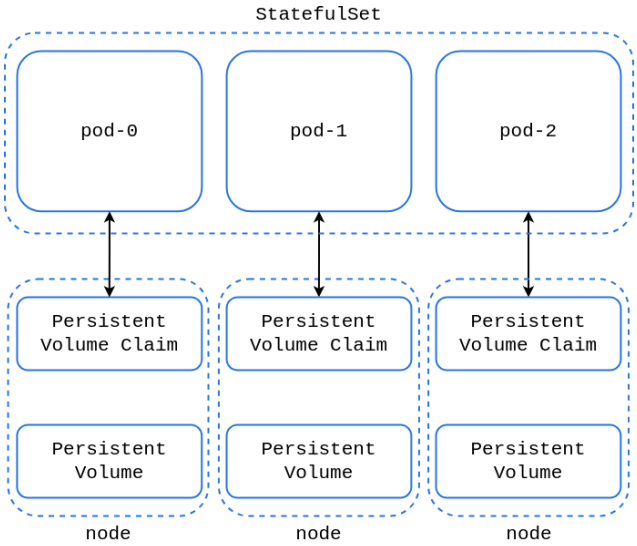

Рисунок 4. Структура Stateful-манифеста

Практическая работа

Задание 1. Исследование.

Для выполнения нижеописанной части лабораторной работы Вам потребуется:

- 3 чистые ВМ с Astra Linux 1.8 (мастер-узлу выделите 2 ядра цп/3гб ОЗУ, остальным 2/2)
- Доступ к ним по SSH для Ansible
- Ansible на хостовой ВМ для запуска сценария, подготавливающего узлы кластера.
- Скорректировать имена и адреса хостов в файле инвентаря. Изменять что-либо в самом сценарии не нужно. Запустите выполнение приложенного сценария playbook.yml и удостоверьтесь, что он выполнился на всех хостах без ошибок.

Подключитесь по SSH к мастер-узлу и выполните следующую команду для инициализации нового кластера Kubernetes (с заданными параметрами инициализации можно ознакомиться изучив файл */etc/kubernetes/kubeadm-config.yaml*, его содержимое также приведено в файле сценария Ansible, который Вы использовали для подготовки узлов кластера)

```
sudo kubeadm init --config /etc/kubernetes/kubeadm-config.yaml
```

Просмотрите вывод kubeadm (это может быть полезно, так как он содержит информацию о том, какие этапы развертывания были проделаны, полученный итоговый результат, а также сведения по дальнейшим действиям, которые надо предпринять для получения полностью работоспособного кластера, что впрочем и будет Вами проделано по мере выполнения данной лабораторной работы) и найдите команду для добавления рабочих узлов (не control-plane), она начинается с kubeadm join ..., скопируйте ее для последующего выполнения на других узлах (*Важная информация: в этой команде для авторизации используется временный токен, который действителен 24 часа. При необходимости создать новый токен можно командой kubeadm token create --print-join-command*)

Скопируйте файл конфигурации кластера, чтобы kubectl мог автоматически считывать его (**Запомните, копируемый файл admin.conf содержит всю необходимую информацию для получения полного доступа к кластеру, его надо беречь**)

```
cd k8s/
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Теперь мы готовы осуществить свое первое взаимодействие с кластером, давайте для начала убедимся что кластер работает и мы имеем доступ к нему, выполнив

```
kubectl cluster-info
```

Если все в порядке, в выводе Вы получите информацию о статусе и том, по какому адресу (точнее будет сказать, URL) доступны плоскость управления (control plane) кластера (читай API-сервер) и сервис внутрикластерного DNS. Следующим шагом давайте проверим состояние узлов нашего кластера (состоящего пока что из единственного мастер-узла), сделать это можно командой

```
kubectl get nodes
```

Узел находится в статусе *NotReady*, однако больше ничего конкретного относительно данной ситуации мы узнать из полученных данных не можем, даже если попросим более подробный вывод в том же табличном формате: kubectl get nodes -o wide. Отложим пока ситуацию со статусом узла на некоторое время в сторону и попробуем выяснить, какие поды работают (и работают ли) сейчас в кластере. Выполним команду

```
kubectl get pods
```

и получим ответ, что ресурсов (подов в данном случае) в пространстве имен по умолчанию (default namespace) не обнаружено. Так как команде *get* для получения существующих объектов namespace-специфичного ресурса не было передано желаемое пространство имен, то и результат мы получили только для пространства имен по умолчанию, которое сейчас пустует. Чтобы получить информацию об объектах, существующих в других пространствах имен, их надо указать в запросе

kubectl. Дабы понимать, какие у нас пространства имеются "из коробки", получим их перечень

```
kubectl get namespaces
```

В выводе помимо уже знакомого default будет интересующее нас в текущий момент *kube-system*, поскольку в нем располагаются объекты, созданные самим кластером, а значит именно там мы должны найти поды с компонентами кластера.

Попробуем еще раз получить информацию о подах, только теперь уже с конкретикой, что мы хотим получить данные из пространства имен *kube-system*

```
kubectl get pods -n kube-system
```

Полученная информация должна нас успокоить в том плане, что компоненты кластера запущены и нормально функционируют, однако отметим, что поды CoreDNS почему-то зависли в состоянии ожидания (*Pending*), давайте узнаем точную причину происходящего. В этом может помочь команда *kubectl describe*, она используется, чтобы получить подробную информацию о конкретном объекте (объектах), в нашем случае о поде CoreDNS. (*Крайне полезная информация: Чтобы нормально читать вывод этой и некоторых других команд в табличном формате, используемом по умолчанию, может потребоваться достаточно сильно увеличить размер окна терминала и возможно уменьшить размер шрифта*) Полная команда будет выглядеть так (вместо *<pod_name>* подставьте имя любого пода CoreDNS):

```
kubectl describe -n kube-system pods/<pod_name>
```

И самый конец вывода, секция События (Events) прямым текстом говорит нам о том, что планировщик (Scheduler) не смог найти подходящий для развертывания пода узел по причине того, что на единственном доступном узле висит "черная метка", ограничение (taint), к которому у данного пода нет допуска (toleration). В этом можно убедиться, сравнив пункты из вышележащей секции Tolerations в выведенных данных, с вызвавшим проблему ограничением, оно не будет прописано в допусках.

Теперь посмотрим, а как же получить информацию обо всех однотипных объектах из всех пространств имен кластера сразу, на примере получения списка всех существующих на текущий момент в кластере подов:

```
kubectl get pods -A
```

Полезная информация: узнать какие еще ресурсы кластера существуют помимо pod и node, можно выполнив команду kubectl api-resources тут же можно узнать, краткие имена ресурсов, какие ресурсы namespace-специфичны (как pod), а какие едины для всего кластера (как node)

Итак, настало время разобраться, что не так с нашим узлом, для этого нам снова поможет команда *describe* для подробного удобочитаемого вывода доступной информации об этом объекте:

```
kubectl describe nodes
```

В ее выводе найдем секцию Conditions, а в ней пункт Ready. Данные в столбце Message достаточно прямолинейно говорят о существующей проблеме: не готова к работе сеть подов, так как не проинициализировано расширение (плагин) CNI. Еще выше можно найти информацию о текущих ограничениях на узле (секция Taints). Зафиксируйте информацию о том, какое еще ограничение, помимо найденного в событиях пода CoreDNS ранее, висит на узле (преподаватели могут спросить при сдаче работы) и подумайте, для чего оно существует *(подсказка: все увиденные нами ранее "системные" поды имеют специально прописанный в их спецификациях допуск (толерантность) к этому ограничению, пользовательские нагрузки по умолчанию допусков не имеют)*.

Теперь, когда мы выявили причину всех текущих бед с кластером, надо срочно подумать об установке и настройке CNI-плагина, который отвечает за сеть подов, а если быть точнее, за выделение каждому поду IP адреса, за обеспечение связности между подами на разных узлах (посредством маршрутизации либо инкапсуляции) и опционально за применение сетевых политик, разграничивающих доступ. В данной работе будет использоваться один из наиболее известных CNI-плагинов, Calico.

Подробно разобрать принцип работы CNI-плагинов, их особенности и тому подобное мы в рамках данной работы не сможем, поэтому ограничимся краткой информацией: CNI-плагин Calico в режиме "по умолчанию" использует для передачи данных инкапсуляцию (туннелирование) IPIP (если кратко, на пакет с заголовком, содержащим IP-адрес из подовой подсети, поверх навешивается еще один IP-заголовок, только уже с адресом узлового интерфейса, выходящего во внешнюю сеть) и динамическую маршрутизацию через протокол BGP с установлением всеми узлами соединений друг с другом (full mesh).

Инкапсуляция крайне важна для случая, когда узлы кластера разнесены по разным IP-сетям и трафик, передаваемый между ними должен маршрутизироваться промежуточными маршрутизаторами (которые в общем случае не в курсе про какие-то внутренние сети кластера из диапазона "серых" IP адресов), в нашем случае "все узлы в одной локальной сети" передача трафика подов между узлами работала бы и без инкапсуляции, поскольку коммутатор не смотрит на IP-заголовки, но менять режим передачи мы не будем. Необходимость в динамической маршрутизации вызывается тем фактом, что единая IP сеть (с маской /16), выделенная для подсети подов в кластере (она задавалась в конфигурации kubeadm как *podSubnet* и потом будет указана в манифесте с параметрами для установки CNI-плагина Calico), разделяется на небольшие подсети (/26), каждая из которых (а при необходимости и несколько сразу) назначается отдельному узлу кластера. И, как нетрудно догадаться, если нужно доставить пакет поду из другой подсети, находящемуся на другом узле, узел, с которого пакет отправляется, должен точно знать, на каком именно узле кластера находится подсеть подов для адреса назначения.

Приступим к установке, для начала установим специальный оператор Kubernetes от разработчиков Calico, который самостоятельно развернет все требуемые компоненты CNI-плагина Calico согласно переданным ему параметрам. Оператор в Kubernetes - это приложение, которое следит за установкой и

осуществляет управление (создание и изменение) кастомными (определяемыми с помощью манифестов типа CustomResourceDefinition) ресурсами, помогает отслеживать изменения и поддерживать эти ресурсы в желаемом состоянии. Tigera operator, который мы установим далее, осуществляет полное управление жизненным циклом Calico в кластере k8s, помимо управления кастомными, создает объекты ресурсов стандартного (Deployment, DaemonSet,...) типа, масштабирует поды при необходимости, через него мы можем Calico устанавливать, изменять его конфигурацию, обновлять версию и тд. Следующей командой применив манифест, создадим все необходимые кастомные ресурсы и объекты (вывод покажет, какие именно).

```
kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.29.1/manifests/tigera-operator.yaml
```

Если теперь мы снова получим перечень всех подов kubectl get pods -A, то обнаружим, что в новом пространстве имен появился новый под *tigera-operator...*, содержащий контейнер с приложением оператора внутри. На самом деле под разворачивается не сам по себе, а как экземпляр (реплика), созданный согласно шаблону пода из спецификации объекта типа Развертывание (Deployment), созданного как раз для запуска tigera-operator в кластере. Объект Deployment управляет объектами типа набор реплик (ReplicaSet), которые в свою очередь ответственны за поднятие и поддержание указанного в их спецификации числа одновременно работающих реплик подов. При изменении развертывания, будет создан новый ReplicaSet, а старый удален (так как заменить поды внутри набора мы не можем). Подтвердить это можно получив список существующих объектов ReplicaSet и Deployment, заодно подметив определенную правила в именовании созданных для развертывания наборов реплик и подов.

```
kubectl get deployments -A
kubectl get replicasets -A
kubectl get pods -A
```

Мы можем подождать, пока под tigera-operator полностью запустится, однако больше никаких изменений в кластере не произойдет, проблема с отсутствием сети подов останется на своем месте. А все дело в том, что пока мы не создадим в кластере объект, содержащий параметры установки Calico, оператор ничего делать не будет. Нужная конфигурация в файле манифеста для кастомного ресурса была создана Ansible на подготовительном этапе и находится в файле *~/k8s/calico.yaml*. Просмотрите содержимое данного файла, наиболее важная информация там - выделенная для подов сеть (podCIDR), которая совпадает с указанной *podSubnet* в конфигурации kubeadm для инициализации кластера. Применим манифест (*в отличие от прошлой команды kubectl create, kubectl apply позволяет как создать новые объекты, так и обновить уже существующие*), а затем проверим существующие на текущий момент в кластере развертывания (deployments) и их статус, сколько реплик (подов) из желаемого их числа готово к работе, сколько соответствует последней версии манифеста, сколько всего доступно

```
kubectl apply -f ./calico.yaml
kubectl get deployments -A
```

Проверим поды kubectl get pods -A, дабы убедиться в том, что помимо подов, принадлежащих увиденным ранее развертываниям, появились также и *calico-node...*, они принадлежат другому типу высокоуровневого объекта - *DaemonSet*, который удобен в первую очередь для запуска служебных приложений, поскольку гарантирует запуск одного экземпляра пода на каждом из узлов кластера. Просмотреть существующие объекты можно по аналогии с развертываниями, командой kubectl get daemonsets -A Подождите, пока все созданные оператором поды проинициализируются и придут в состояние готовности, после этого инициализация CNI завершится, статус узла из вывода kubectl get nodes станет *Ready*, запустятся поды с CoreDNS. Давайте проверим, заработала ли сеть подов, узнав, получили ли поды свои IP адреса, информация о которых доступна при использовании расширенного вывода команды get

```
kubectl get pods -A -o wide
```

В выводе команды можно найти еще и ответ на вопрос, почему некоторые поды успешно работали с самого начала, до момента внедрения и запуска CNI-плагина (зафиксируйте свое предположение по этому поводу для отчетности). Ну а раз все работает, значит мы готовы расширить наш кластер, чтобы он мог называться так обоснованно, поэтому приготовьте ранее сохраненную команду для присоединения нового рабочего узла к кластеру, мы начинаем масштабироваться.

В новой вкладке/новом терминале подключитесь к любому будущему рабочему узлу и выполните ранее сохраненную команду для присоединения рабочих узлов на нем

```
sudo kubeadm join ...
```

На мастер узле, выведя список узлов и список подов, можем сразу заметить, что в перечне узлов появился новый рабочий узел, а контроллер DaemonSet (входит в состав контроллеров, запускаемых компонентом кластера kube-controller-manager), автоматически разворачивает по экземпляру подов для каждого существующего в кластере объекта DaemonSet на нем. Перед добавлением второго рабочего узла рекомендуем убедиться, что первый перешел в статус *Ready* (ну или Вы при условии недостаточности ресурсов хостовой ВМ рискуете уронить мастер).

Добавьте в кластер второй рабочий узел, действия аналогичны первому, дождитесь готовности на втором узле всех запланированных подов.

Раз мы упомянули выше про возможный недостаток ресурсов, давайте попробуем получить данные о текущей загрузке узлов кластера, используя встроенную команду kubectl для этого:

```
kubectl top node
```

И... мы получаем ошибку, что Metrics API недоступен. "Из коробки" Kubernetes не содержит компонента для реализации этого функционала, однако его можно очень легко установить, давайте посмотрим, как это можно сделать.

До сих пор мы при необходимости что-то создать/установить в кластере использовали kubectl и файлы с YAML-манифестами, но это не единственный способ, и в некоторых случаях (например, для сложных микросервисных приложений) точно не самый простой. Для Kubernetes существует менеджер пакетов, который подобно пакетному менеджеру Apt для Debian-подобных дистрибутивов Linux, значительно упрощает развертывание, обновление и обслуживание приложений в кластере, используя чарты (приблизительно это шаблоны требуемых для развертывания приложения манифестов, по примеру тех шаблонов, что Вы использовали для файлов конфигурации сервисов в работе, посвященной Ansible), хранимые в общедоступных репозиториях.

Установим сам Helm

```
curl -L https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash
```

Для удобства сразу настроим автодополнение вводимых команд (по нажатию Tab), как было на этапе подготовки сделано для kubectl (а Вы еще не пробовали?)

```
source <(helm completion bash)
helm completion bash | sudo tee /etc/bash_completion.d/helm > /dev/null
```

Пробуем установить теперь с его помощью компонент, реализующий Metrics API - Metrics server, который позволит получать данные о потребляемых ресурсов подами и загрузке узлов используя команду kubectl top (а также необходим для функционирования горизонтального автомасштабирования нагрузок), но для начала добавим нужный репозиторий и скачаем файл, содержащий значения параметров установки, поскольку потребуется внести некоторые изменения.

```
helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server
helm repo update
helm show values metrics-server/metrics-server > ./metrics-server.values
```

По умолчанию Metrics server требует, чтобы в кластере функционировал выпуск X.509 сертификатов от имени доверенного удостоверяющего центра (Certification Authority), иначе он не будет подключаться к узлам с недоверенными сертификатами. Чтобы исправить эту ситуацию, мы можем немного подправить его конфигурацию, для этого последней выполненной командой мы сохранили все используемые при установке Metrics server параметры (пока что имеющие значения по умолчанию) и теперь можем по своему усмотрению подправить этот аналог файла с переменными для сценария Ansible. В файле *metrics-server.values* найдите приведенный ниже фрагмент текста и добавьте последний, отсутствующий по умолчанию параметр

```
defaultArgs:
  - --cert-dir=/tmp
  - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
  - --kubelet-use-node-status-port
  - --metric-resolution=15s
  - --kubelet-insecure-tls
```

Теперь все готово к установке, осталось только создать выделенное пространство имен и можно приступать

```
kubectl create ns metrics-server
helm install metrics-server metrics-server/metrics-server -n metrics-server --values ./metrics-server.values
```

После того, как все поды в пространстве имен *metrics-server* запустятся, можете попробовать получить данные о загрузке узлов

```
kubectl top node
```

Если возвращается ошибка, немного подождите, сбор метрик при первом запуске Metrics server происходит не быстро

На пути к полностью готовому к развертыванию пользовательских приложений кластеру нам остался шаг, заключающийся в обеспечении возможности предоставления постоянного хранилища для развернутых в кластере Stateful-приложений, которые, как известно, должны сохранять свои данные между перезапусками. По аналогии с Docker, хранилища, монтируемые внутрь контейнера в поде называются томами (volume). Том в контексте Kubernetes может быть обычным (Volume), который определяется внутри спецификации пода и его жизненный цикл целиком завязан на под (т.е. удалится под, удалится и объект, представляющий том), либо постоянным (PersistentVolume). Объекты типа PersistentVolume (PV) имеют свой, независимый от других объектов жизненный цикл, их можно подключать к нескольким подам (контейнерам) сразу при указании в спецификации сответствующего типа доступа. Для подключения томов мы можем использовать как встроенные в k8s драйвера для некоторых типов хранилищ, и самостоятельно вручную определять каждый том отдельным манифестом, без необходимости доустанавливать что-либо, так и использовать автоматическое выделение тома соответствующего класса хранилища (StorageClass) по заявке PersistentVolumeClaim (PVC), используя встроенный либо внешний поставщик (Provisioner) хранилища и при необходимости, внешний CSI-драйвер для предоставления кластеру Kubernetes доступа к хранилищу.

Далее мы продемонстрируем на практике оба упомянутых варианта, полностью "ручной" и более "автоматизированный". В качестве хранилища будем использовать NFS сервер, который развернем на хостовой ВМ. (*Краткая справка, NFS (Network File System) - это в некотором роде аналог SMB протокола, позволяющий осуществлять доступ на файловом уровне к директориям на удаленном сервере*) Давайте первоначально его и установим, для этого потребуется запустить соответствующий сценарий Ansible на хостовой ВМ и создать отдельную директорию для выделенного "вручную" тома.

```
ansible-playbook ./nfs_host.yml
mkdir /srv/nfs_share/pv1
```

Хранилище для томов готово, теперь вернемся на мастер узел и создадим первую заявку PersistentVolumeClaim

```
cat << EOF > test-pvc.yaml
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nfs-claim-manual
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
EOF
kubectl apply -f test-pvc.yaml
```

Проверим состояние заявки PVC и информацию об имющихся томах PV в кластере (здесь мы воспользуемся кратким именем ресурса)

```
kubectl get pvc -A
kubectl get pv
```

Заявка PVC находится в состоянии *Pending* (посмотрите в подробном выводе информации об этом PVC, почему так, зафиксируйте для отчетности) и похоже, подходящий том PV для нее нам придется создать самостоятельно, применив следующий манифест. Параметр *persistentVolumeReclaimPolicy* отвечает за судьбу тома PV, после того, как сцепленная с ним заявка PVC будет удалена. В приведенном случае том останется существовать, как был (Retain).

```
cat << EOF > test-pv.yaml
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-host-pv1
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  nfs:
    server: 192.168.122.1
    path: /srv/nfs_share/pv1
EOF
kubectl apply -f test-pv.yaml
```

Подождем секунд 5 и проверим, что том создан и успешно "сцепился" с заявкой PVC.

```
kubectl get pvc
```

```
kubectl get pv
```

Если все прошло как надо, то в выводе информации о PVC и PV статус обоих будет *Bound*, кроме того для тома PV в столбце CLAIM будет указано, с какой заявкой PVC он сцеплен.

Для реализации варианта автоматизированного выделения томов PV для заявок PVC, нам потребуется установить внешний поставщик (Provisioner) для NFS (так как Kubernetes не имеет встроенного поставщика для NFS), который в директории, экспортируемой с сервера NFS, будет создавать по вложенной директории на каждый создаваемый им том PV. Установим, снова используя Helm, при установке также автоматически создадим для него класс хранилища, назначаемый по умолчанию (он используется, если желаемый класс не указан в заявке PVC)

```
helm repo add nfs-subdir-external-provisioner https://kubernetes-sigs.github.io/nfs-subdir-external-provisioner/
kubectl create ns nfs-provisioner
helm -n nfs-provisioner install nfs-provisioner nfs-subdir-external-provisioner/nfs-subdir-external-provisioner \
    --set nfs.server=192.168.122.1 \
    --set nfs.path=/srv/nfs_share \
    --set storageClass.defaultClass=true \
    --set replicaCount=1 \
    --set storageClass.name=nfs \
    --set storageClass.provisionerName=nfs-provisioner
```

После развертывания поставщика, экспресс-проверку работоспособности можно провести так: Создаем еще одну заявку PersistentVolumeClaim:

```
cat << EOF | kubectl apply -f -
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nfs-claim-auto
spec:
  storageClassName: nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
EOF
```

Проверим состояние созданной заявки и убедимся, что она сразу перешла в статус *Bound*, а среди томов PV автоматически появился новенький

```
kubectl get pvc/nfs-claim-auto
kubectl get pv
```

Поскольку том создавал поставщик, а не мы сами, то и параметры, с которыми он был создан, для нас сейчас не очень ясны, к счастью имеется способ для уже существующего в кластере объекта получить его YAML-манифест (с поправкой на присутствие в выводе дополнительных полей, вроде статуса, метки времени изменения и тп), делается это указанием YAML как желаемого формата вывода

```
kubectl get pv/<pv_name> -o yaml
```

На основе предыдущего вывода, зафиксируйте для отчетности, какую *persistentVolumeReclaimPolicy* имеет созданный поставщиком том, а также какое поле в секции метаданных содержит указание на то, что том был создан именно поставщиком. И наконец настало время финальной проверки, создаем для каждого тома PV по поду, задача которого - в директории, куда смонтирован том, создать файл, чтобы мы смогли удостовериться, что данные из пода успешно сохраняются на NFS сервере.

```
for method in auto manual; do
cat <<- EOF | kubectl apply -f -
kind: Pod
apiVersion: v1
metadata:
  name: nfs-pv-$method
spec:
  containers:
  - name: nfs-test
    image: busybox:stable
    command:
      - "/bin/sh"
    args:
      - "-c"
      - "touch /mnt/SUCCESS-$method && echo \"Successfully created the file SUCCESS-$method ;)\" && exit 0 || echo \"couldn't create the file :(\" && exit 1"
    volumeMounts:
      - name: nfs-pvc
        mountPath: "/mnt"
  restartPolicy: "Never"
  volumes:
    - name: nfs-pvc
      persistentVolumeClaim:
        claimName: nfs-claim-$method
EOF
done
```

Контейнер в поде при успешном создании файла (впрочем, при ошибке создания тоже) выводит запись о результате в свой STDOUT, что является стандартным способом логирования для развертываемых в k8s контейнеризованных приложений (*можете потом самостоятельно по аналогии с приведенной ниже командой просмотреть логи любого другого пода*). Для просмотра логов пода, существует отдельная команда kubectl - *logs*, подобно аналогичной для docker.

Давайте возьмем для примера один из созданных для проверки хранилища подов и убедимся по логам, что контейнер в нем справился с задачей, оставив запись об успехе ее выполнения:

```
kubectl logs pods nfs-pv-...
```

Полезно будет еще отметить, что поскольку контейнеры в этих двух подах были рассчитаны на выполнение конкретного действия и последующее завершение работы, то при просмотре их состояния, поды nfs-pv... будут в статусе *Completed*, что также говорит об успешном выполнении запущенной в контейнерах команды (код возврата 0). Если хотите окончательно убедиться в работоспособности хранилища, на хостовой ВМ внутри директории */srv/nfs_share* можете найти внутри созданной ранее вручную pv1 и автоматически созданной поставщиком директорий файл вида SUCCESS-... .

Ненужные объекты, как например, отработавшие свою задачу поды *nfs-pv...*, или ненужные более PC, PVC можно удалить из кластера с помощью команды kubectl delete, для практики удалите любой из подов *nfs-pv...*

```
kubectl delete pods <pod_name>
```

- Имея теперь на руках готовый для развертывания приложений, в т.ч. требующих сохранения данных, кластер, развернем в нем первое приложение, которое Вам знакомо по курсу БД - PostgreSQL 16. Перед началом развертывания доведем некоторую важную информацию и принятые условности:
- PostgreSQL будет развернут на базе StatefulSet с 1 репликой, увеличивать число реплик в рамках данной работы не требуется.
- Запрещать желающим попробовать сделать работоспособное решение для нескольких реплик также не станем, но только с оговоркой, что делать его надо на основе самостоятельно созданных манифестов, без использования операторов и готовых Helm-чартов.
- Начнем развертывание PostgreSQL мы с того, что познакомимся еще с двумя видами ресурсов k8s и создадим по объекту для каждого из них, первый - ресурс типа *Secret*, предназначенный для хранения чувствительной информации в кластере, без необходимости указывать ее в открытом виде в конфигурациях других объектов, которым она нужна. В объект такого типа (*Secret*) запишем информацию о пользователе и пароле для PostgreSQL, что позволит затем в манифесте развертывания StatefulSet с PostgreSQL, не прописывать пароль и имя пользователя в открытом виде, а указать ссылку на созданный объект типа *Secret*.

```
kubectl create namespace postgres
kubectl create -n postgres secret generic postgresql-secrets \
  --from-literal=user=postgres \
```

```
    --from-literal=password=pass
kubectl label secret -n postgres postgresql-secrets app=postgres
```

- *При необходимости администратор кластера всегда сможет вытащить информацию из секретов, вот так можно получить обратно заданный на предыдущем шаге пароль kubectl get secret --namespace postgres postgresql-secrets -o jsonpath="{.data.password}" | base64 -d*
- Вторым полезным и удобным объектом является *ConfigMap*, который обычно хранит в себе конфигурацию развертываемых приложений. Его можно как создать с нуля манифестом, так и использовать в качестве основы существующий файл конфигурации приложения, указав это в специальном параметре *--from-file* при создании объекта через kubectl.

```
kubectl apply -f -<< EOF
    apiVersion: v1
    kind: ConfigMap
    metadata:
      name: postgresql-config
      namespace: postgres
      labels:
        app: postgres
    data:
      POSTGRES_DB: mydb_prod
      PGDATA: /var/lib/postgresql/data/pgdata
    EOF
```

postgresql.conf

```
listen_addresses = '*'
max_connections = 10
shared_buffers = 64MB
max_wal_size = 512MB
min_wal_size = 80MB
datestyle = 'iso, mdy'
timezone = 'Europe/Moscow'
```

```
kubectl create configmap -n postgres postgresql-config --from-file=./postgresql.conf
```

- Ниже приведен набор манифестов для нескольких объектов k8s, создание которых необходимо для установки PostgreSQL в кластере, этот кусок кода Вам надо сохранить в отдельный yaml-файл, например *postgres.yaml* и затем применить его через команду kubectl apply

postgres.yaml

```
---
apiVersion: v1
kind: Service
metadata:
```

```
  name: postgresql
  namespace: postgres
  labels:
    app: postgres
spec:
  selector:
    app: postgres
  ports:
    - port: 5432
  clusterIP: None
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgresql
  namespace: postgres
spec:
  serviceName: postgresql
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:16
        imagePullPolicy: "IfNotPresent"
        ports:
          - containerPort: 5432
        resources:
          requests:
            memory: "64Mi"
            cpu: "128m"
          limits:
            memory: "256Mi"
            cpu: "500m"
        env:
          - name: POSTGRES_USER
            valueFrom:
              secretKeyRef:
                name: postgresql-secrets
                key: user
          - name: POSTGRES_PASSWORD
            valueFrom:
              secretKeyRef:
                name: postgresql-secrets
                key: password
          - name: POSTGRES_DB
```

```
            value: mydb_prod
          - name: PGDATA
            value: /var/lib/postgresql/data/pgdata
        volumeMounts:
          - name: postgresql-db-storage-claim
            mountPath: /var/lib/postgresql/data
          - name: postgres-config-volume
            mountPath: /etc/postgresql
      volumes:
      - name: postgres-config-volume
        configMap:
          name: postgresql-config
          items:
          - key: postgresql.conf
            path: postgresql.conf
  volumeClaimTemplates:
    - metadata:
        name: postgresql-db-storage-claim
      spec:
        accessModes:
          - ReadWriteOnce
        storageClassName: nfs
        resources:
          requests:
            storage: 5Gi
```

- Убедитесь, что все объекты успешно создались и готовы к работе. Если так, то переходим к развертыванию pgAdmin, создаем секрет с паролем администратора

```
kubectl create -n postgres secret generic pgadmin-secrets \
  --from-literal=pgadmin-pass=postgres
```

- Создаем файл с манифестами для развертывания pgAdmin

pgadmin.yaml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pgadmin
  namespace: postgres
spec:
  selector:
   matchLabels:
    app: pgadmin
  replicas: 1
  template:
    metadata:
      labels:
        app: pgadmin
    spec:
```

```
      containers:
        - name: pgadmin4
          image: dpage/pgadmin4
          imagePullPolicy: "IfNotPresent"
          env:
            - name: PGADMIN_DEFAULT_EMAIL
              value: "admin@mpsu.stu"
            - name: PGADMIN_DEFAULT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: pgadmin-secrets
                  key: pgadmin-pass
            - name: PGADMIN_PORT
              value: "80"
          ports:
            - containerPort: 80
              name: pgadminport
---
apiVersion: v1
kind: Service
metadata:
  name: pgadmin
  namespace: postgres
  labels:
    app: pgadmin
spec:
  selector:
   app: pgadmin
  type: NodePort
  ports:
   - port: 80
     nodePort: 30200
```

Применяем kubectl apply -f pgadmin.yaml. Ждем готовности пода с pgAdmin и проверям доступ с хоста через веб-браузер, используя адрес мастер-узла и порт 30200

Задание 2. Практика.

1. Развернуть stateful-манифесты для PostgreSQL и проверить их на работоспособность, например, с помощью pgAdmin4 или psql.
2. Развернуть stateless-манифесты для веб-приложения (не забудьте собрать образ приложения с помощью Dockerfile), а также настроить подключение к базе данных PostgreSQL внутри Kubernetes кластера.
3. Отправить несколько HTTP-запросов на веб-приложение и убедиться в отсутствии ошибок (код ответа должен иметь код 200, а также возвращать тело ответа).

Список рекомендованных статей:

1. https://habr.com/ru/articles/777728/
2. https://habr.com/ru/articles/651653/
3. https://habr.com/ru/companies/gazprombank/articles/789404/
4. https://kubernetes.io/ru/docs/reference/kubectl/cheatsheet/
5. https://habr.com/ru/companies/slurm/articles/783708/
6. https://habr.com/ru/companies/orion_soft/articles/834806/
7. https://habr.com/ru/companies/T1Holding/articles/781368/
8. https://habr.com/ru/companies/timeweb/articles/703550/
9. https://habr.com/ru/companies/slurm/articles/833408/
10. https://habr.com/ru/articles/856752/
11. https://habr.com/ru/companies/oleg-bunin/articles/790112/