Лабораторная работа 13

Kubernetes

Оглавление

Теоретическая часть

> Кубер унижает человеческое достоинство разными способами и на разных этапах.
> Это часть опыта от пользования продуктом. Так задумано.
> *(цитата из статьи на Хабре)*

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, единые 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 эффективное управление ресурсами. Серверы либо оставались недогружены, либо, наоборот, перегружались. Например, если серверу было выделено больше ресурсов, чем нужно, невостребованная из них часть 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. В качестве языка программирования для разработки нового инструмента был выбран молодой на тот момент язык Go.

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. Существуют готовые платформы на его основе, расширяющие его функционал с помощью других программных компнентов и позволяющие, например, управлять с его помощью не только контейнерами, но и виртуальными машинами. Kubernetes одинаково хорошо работает как на локальных серверах, так и в облаке, а также поддерживает гибридные решения, где часть инфраструктуры находилась в дата-центре компании, а часть — в облаке. Это сделало его идеальным выбором для оркестрации современных распределённых систем.

Архитектура Kubernetes

Для того, чтобы запускать приложения используя Kubernetes необходимо понимать его архитектуру и основные особенности его работы.

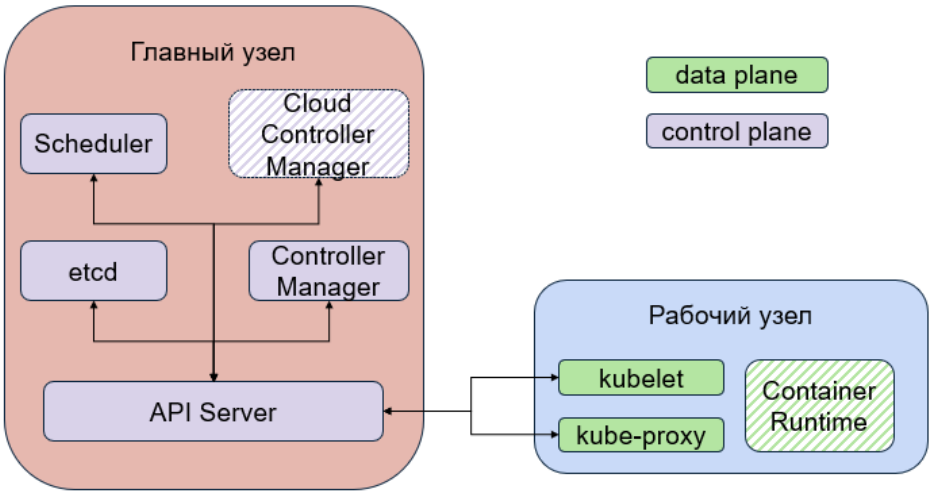
Kubernetes — это портативная расширяемая платформа с открытым исходным кодом для управления контейнеризованными рабочими нагрузками и сервисами, которая облегчает как декларативную настройку, так и автоматизацию. Слово “Kubernetes” происходит от древнегреческого κυβερνήτης, что означает капитан, рулевой, пилот; тот, кто управляет. Используется сокращенное название - “K8S”, где цифра 8 — это восемь букв между K и S.

Кластер Kubernetes состоит из взаимодействующих между собой и внешней средой узлов (*nodes*). Один из них является главным узлом (*master node*), управляющим кластером, остальные – рабочие (*worker node*), на которых запускаются контейнирезированные приложения. Такой подход построения характерен для многих видов оркестраторов. В системах, работающих под большими нагрузками главных узлов может быть несколько.

В качестве одного узла обычно совместимая операционная система с необходимыми для работы k8s пакетами в ней, которая может быть развернута в виде физического устройства или виртуальной машины. Теоретически, в рамках обучения и тестирования возможен запуск всего кластера Kubernetes на одном узле, однако в промышленных масштабах такой подход является бесполезным.

Основной единицей исполнения в Kubernetes является не непосредственно сам контейнер, а более высокоуровневый компонент – **Pod** – группа из одного или нескольких контейнеров с общими хранилищами и сетевыми ресурсами, а также спецификация для запуска контейнеров. При создании подов не указывается конкретный узел, на котором они будут расположены. IP адрес пода назначается сетевым плагином Kubernetes и может быть изменен при его пересоздании. Поды сами по себе эфемерны, т.е. недолговечны – создаются под конкретную задачу и после её завершения уничтожаются. Поды не восстанавливаются – только пересоздаются заново.

На рисунке 1 представлена архитектура кластера Kubernetes.



Начнем рассмотрение архитектуры с рабочего узла. Т.к. в системе запускаются контейнирезированные приложения, то логично, что он будет содержать компонент, отвечающий за их запуск и обслуживание – **container runtime**. Самый известный пример container runtime – Docker, однако в последних версиях k8s используются другие движки, например containerd или CRI-O.

Для взаимодействия между container runtime и главным узлом существует компонент **kubelet**. Он принимает задачу и в соответствии с неё передает указание для запуска контейнера. Далее kubelet следит, чтобы созданные контейнеры исправно работали. Кроме этого, задачей kubelet является регистрация узла в кластере.

Более сложным компонентом является **kube-proxy.** Он предназначен для сетевого взаимодействия контейнеров между собой и балансировки трафика между ними. Например, если мы обратимся к узлу по его IP адресу, а на нем по некоторым причинам будет отсутствовать необходимый нам под - kube-proxy перенаправит наш запрос на другой узел, где указанный под существует. Если узлов, на которые можно передать запрос несколько, то применяются алгоритмы балансировки нагрузки, позволяющие равномерно распределять запросы (например, при использовании ipvs в качестве режима работы kube-proxy по умолчанию используется алгоритм round robin).

Таким образом, на рабочих узлах располагаются следующие компоненты: container runtime, kubelet, kube-proxy и непосредственно сами приложения.

Архитектура главного узла является более сложной. Все его компоненты отвечают за управление всей системой, поэтому в соответствии с классификацией их относят к понятию **Control Plane** *(управляющий слой).*

Первым и центральным компонентом является **API Server**, который отвечает за обработку всех внутренних и внешних запросов, взаимодействующих с кластером. Доступ к нему можно получить с помощью REST API. Кроме этого, компонент управляет аутентификацией и авторизацией пользователей и служб – определяет отправителя запроса и выясняет может ли пользователь выполнять запрошенные действия над ресурсом. Если действия возможны, то переданный объект проходит валидацию и передается другим компонентам.

Полученные значения состояний кластера передаются через API Server и сохраняются в специальном хранилище – **etcd**. Etcd является распределенным хранилищем, в котором данные хранятся в формате «ключ-значение». Распределенность хранилища подразумевает наличие нескольких его копий (реплик). Для того, чтобы значения в них были согласованными друг с другом применяется алгоритм Raft. Он заключается в том, что среди копий хранилища выбирается одна, которая является лидером. Лидера выбирают путем «голосования» по большему числу голосов – поэтому количество копий etcd должно быть нечетным. Далее все запросы на запись будут идти только через лидера. Увеличение числа узлов с копией etcd повышает надежность и производительность чтения данных, однако производительность записи, очевидно, падает. Еще одной интересной особенностью etcd является наличие механизма подписки на изменение в хранилище. Так другие компоненты кластера могут быстро отреагировать на любое произошедшее событие.

Для того, чтобы работать с Kubernetes необходимо наличие контроллеров, которые будут приводить кластер в определенное состояние и поддерживать его. Такой механизм осуществляется с помощью менеджера контроллеров (**Controller Manager**)

Для того, чтобы выбрать узел, на котором будет запускаться под, необходим компонент, который учитывал бы все требования и ограничения, накладываемые на него. Таким компонентом является планировщик (**Scheduler).** Для управления кластером используется внешняя утилита **kubectl**.

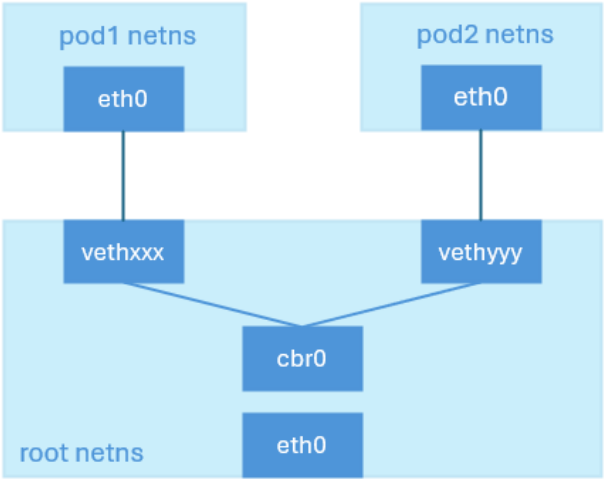
Перечисленные компоненты являются основными и критически важными для жизнеспособности кластера. Однако кроме них существуют дополнения, которые позволяют расширять функционал Kubernetes. К ним относится CoreDNS, сетевые плагины, дашборды и т.п. Более подробно о некоторых из них пойдет речь далее.

Сетевая подсистема Kubernetes

Очевидно, что после запуска приложения в Kubernetes необходимо организовать доступ к нему извне и взаимодействие между несколькими такими развернутыми приложениями. Для этого существует сетевая подсистема Kubernetes.

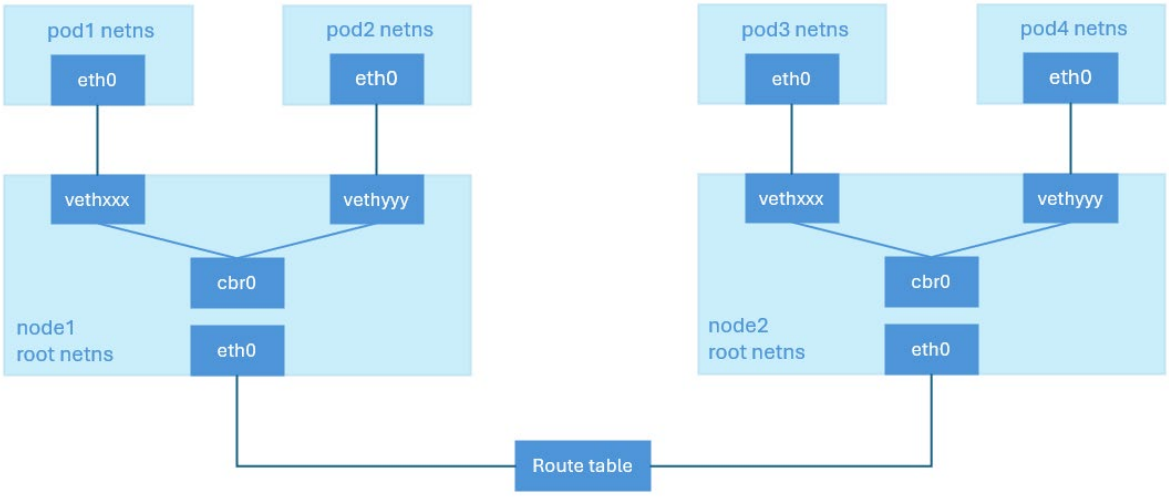
Напомним, что каждое приложение запущено в контейнере и «обернуто» в под, который в свою очередь размещен на узле. Основная задача – обеспечить взаимодействия приложений внутри пода, подов друг с другом, вне зависимости от того, на каком узле они расположены и приложений с «внешним миром».

Каждому поду назначается свой IP адрес. Этот адрес не постоянен – он может меняться при завершении работы пода или при его пересоздании на другом узле. Внутри одного пода может содержаться и более одного контейнера, они располагаются внутри одного сетевого пространства имен и обладают единым IP адресом, который соответствует адресу всего пода. Взаимодействие между ними может происходить за счет средств IPC, сокетов или loopback интерфейса. Для того, чтобы поды общались между собой, необходимо создать сетевой интерфейс, который бы связал сетевое пространство имен пода с пространством имен узла.



В примере, указанном на рисунке, на стороне пода такой интерфейс называется eth0, парный к нему – vethxxx. Далее сетевые интерфейсы необходимо связать между собой – для этого используется соединение типа «сетевой мост» cbr0. Напоминаем, что сетевой мост – это очень упрощенная версия коммутатора, таким образом, передача данных будет идти на канальном уровне, используя MAC адреса подов. Для определения адреса используется ARP запрос.

Чтобы обеспечить взаимодействие подов, расположенных на разных узлах, между собой применяется интерфейс eth0, через который трафик маршрутизируется и передается на нужный узел. Напомним, что за маршрутизацию отвечает компонент узла kube-proxy, который используются правила, указанные в iptables.



Однако у такого подхода есть свой недостаток. Как было сказано выше, IP адреса подов могут изменяться в процессе работы кластера, к тому же многие приложения могут быть реплицированы – им будут соответствовать несколько подов, каждый со своим адресом. Чтобы правильно маршрутизировать трафик в таком случае, создается объект Kubernetes – сервис (Service), в котором содержится ссылка на список IP адресов подов, в которых запущено приложение. Более подробно о них будет рассказано в одном из следующих разделов. Сервис имеет постоянный IP адрес и соответствующее ему доменное имя, по которому может быть произведено обращение. За работу DNS в Kubernetes отвечает **CoreDNS** или kube-dns.

Для работы сети, создания сетевых интерфейсов, их подключения и настройки применяется отдельный сетевой плагин, который должен быть разработан в соответствии со спецификацией **CNI** (*Container Network Interface*). Существуют различные варианты плагинов – Cilium, Flanel, Calico и другие. В рамках лабораторной работы будет рассмотрен плагин Calico.

Хранение данных

После осуществления связи между подами необходимо рассмотреть еще один важный вопрос – хранение данных. Очевидно, что при работе с Kubernetes мы сталкиваемся с двумя вопросами: «Где хранить?» и «Как сохранять?».

Еще раз напомним, что поды сами по себе не имеют постоянного места размещения – они могут быть уничтожены и заново созданы на том же или другом узле. Поэтому если хранение данные на конкретном одном узле сильно ограничивает работу оркестратора. Для решения проблемы применяются сетевые хранилища, такие как NFS, CEPH и др.

Вторая проблема – как сохранять данные после перезапуска или удаления подов? Если приложение не требует сохранения своего состояния после его завершения (такое приложение имеет **Stateless** архитектуру), то возможно сохранение данных внутри самого пода, во временное хранилище. Для этого

применяется абстрактное понятие том (**volume**). В первую очередь он предназначен для обмена данными между контейнерами внутри пода.

Большинство современных приложений имеет **Stateful** архитектуру – она предполагает сохранение состояния на все время работы и возможность его восстановления при повторном подключении. Для решения проблемы используется другой подход.

Все данные подов в Kubernetes, который хранятся в постоянных хранилищах (**Persistent Volumes**), существуют вне зависимости от жизненного цикла пода и не уничтожаются вместе с ним (если не указана иная политика). PVs представляют собой абстракцию над физическим хранилищем, позволяя администраторам предлагать хранилище, как объемные ресурсы в сети, независимо от подробностей реализации внутреннего или внешнего хранилища.

Место в хранилищах возможно получить по заявкам (Persistent Volume Claims). PVCs позволяют разработчикам запрашивать специфические классы хранилищ и доступа, таким образом абстрагируя работу с хранилищем от реализации.

Существуют различные типы томов, которые можно использовать для различных целей, например:

- **ConfigMap** - предоставляет способ хранения данных конфигурации в поде.
- **Secret** — это объект, содержащий небольшой объем конфиденциальных данных, таких как пароль, токен или ключ
- **emptyDir** – представляет собой пустую директорию, видимую всеми контейнерами пода. При уничтожении пода данные их нее удаляются.
- **hostPath, local –** тома, которые связываются с каталогом на узле. Основное отличие между ними заключается в том, что после уничтожения пода с томом hostPath он может быть восстановлен на любом другом узле кластера и связь с каталогом нарушится. Local тома закрепляют поды за узлом и после уничтожения будут восстановлены на той же самой ноде и вернет связь с директорией.
- **PersistentVolumeClaim -** заявка на получение места в постоянном хранилище.

В завершении рассмотрения архитектуры Kubernetes рассмотрим еще несколько понятий, которые встретятся вам во время выполнения лабораторной работы:

Метки

По мере увеличения числа подов в кластере требуется механизм для их классификации и группировки. Представьте, что у вас в системе есть несколько разных версий одного и того же микросервиса, который был реплицирован 20 раз. Чтобы систематизировать доступ применяется понятие метки. Метка – произвольная пара «ключ-значение», присоединяемая к ресурсу, которая затем используется при отборе ресурсов с помощью селектора меток. Один ресурс может иметь несколько меток.

Каждый под обозначается метками двух типов – *app* – название приложения, к которому он относится и *rel* – тип версии приложения.

Пространства имен

Kubernetes поддерживает несколько пулов ресурсов в одном физическом кластере. Они называются пространствами имён. Пространства имён предназначены для разделения ресурсов кластера между несколькими пользователями. Более подробно процесс их создания будет рассмотрен ниже.

Объекты Kubernetes. Манифесты.

Все объекты в Kubernetes описываются с помощью yaml манифестов. Скрипт обязательно должен содержать три поля – apiVersion (версия api), kind (тип создаваемого объекта) и metadata (содержит уникальное имя объекта). Все остальные поля заполняются дополнительно по необходимости.

Значение apiVersion характерно для каждого объекта и может принимать различные значения. Обычно, первая версия в своем названии содержит слово alpha – эта версия небезопасна, в ней могут содержаться ошибки, т.к. содержит новый функционал, который еще активно тестируется. Следующий этап – версия beta – уже протестированная и прошедшая первый этап работы. Устоявшееся версия называется stable и помечается как v1, без дополнительных указаний после числа.

Секция kind может содержать в себе различные значения, соответствующие типам ресурсов, для объектов, создаваемых внутри кластера. Уточним тут, что ресурс — это определение типа объекта (как класс в ООП), а объект — это конкретный экземпляр этого типа (объект класса). Перечислим наиболее популярные из них: Pod, ReplicaSet, Service, Deployment, Job, CronJob, StatefulSet, DaemonSet, StorageClass, ConfigMap, Secret. Основные из перечисленных типов ресурсов были описаны в теоретической части, теперь рассмотрим их процесс создания практически. Для этого развернем кластер Kubernetes используя minikube.

Практическая часть.

Обучающая часть

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/
```

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
```

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

```
    minikube start
```

Команда для просмотра статуса кластера

```
    minikube status
```

Просмотр IP адреса узла.

```
    minikube ip
```

Позволяет пользователю подключиться к пространству созданного узла.

```
    minikube ssh
```

Перед началом работы с кластером проведем некоторые подготовительные операции:

Т.к. большую часть команд, которые вы будете вводить будет начинаться с названия утилиты kubectl рекомендуется создать alias на него:

```
    alias k=kubectl
```

Для удобства можно подключить возможность автодополнения команд:

```
sudo apt-get install bash-completion
echo 'source <(kubectl completion bash)' >>~/.bashrc
echo 'source <(kubectl completion bash | sed s/kubectl/k/g)' >>~/.bashrc
kubectl completion bash >/etc/bash_completion.d/kubectl
```

Аналогично созданный alias возможно добавить в файл .bashrc:

```
echo 'alias k=kubectl' >>~/.bashrc
```

Начало работы с кластером

Для работы с кластером предназначена команда *kubectl*.

Общий синтаксис:

kubectl ДЕЙСТВИЕ [ОБЪЕКТ] [ОПЦИИ]

Наиболее популярные действия и объекты перечислены в таблице

kubectl	get	services pods deployment pv pvc …	имя объекта	Вывести значения указанных объектов
	describe			Вывести полное описание созданного объекта
	delete			Удалить объект
	explain	services pods deployment pv pvc …		Описание полей и структуры ресурса
	logs	имя объекта		Вывести логи объекта
	apply	-f имя манифеста		Применить манифест
	run	имя образа		Запуск контейнера с указанным образом
	exec	имя пода -- команда		Выполнить команду в поде
	scale	--replicas=N объект		Масштабирование ресурсов

Опции указываются, как и в стандартных командах Linux с использованием символов – для однобуквенных значений и -- для отдельных слов. Например, для запуска пода в отдельном пространстве имен возможно указать опцию --namespace=ИмяNS

Рассмотрим примеры:

Просмотр доступных узлов:

```
    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.

Просмотр информации о доступных в кластере нодах, подах (с их IP адресом) и пространствах имён, а также удаление пода соответственно.

```
    kubectl get pods -o wide
    kubectl get namespaces
    kubectl delete pod <name>
```

Прежде чем приступать к созданию объектов – создадим для них отдельное пространство имен

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

Создание 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.

Первым созданным нами объектом будет под. Для того, чтобы создать под, нужно прописать в файле pod.yaml следующую конфигурацию.

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

Рассмотрим указанные параметры. Значения apiVersion укажем v1 (описание пода соответствует версии v1 Kubernetes), тип объекта – под, имя объекта – nginx. В разделе spec укажем информацию о контейнере – его имя, образ и порт, на котором приложение будет слушать. Обратим внимание, что данные о порте – чисто информационные и никак не влияют на работоспособность системы. Поле в первую очередь предназначено для программиста, что было возможно быстро определить к каким портам возможно обращение.

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

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

kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP
nginx   1/1     Running   0          50s   10.244.0.24
```

Обратите внимание, что результат команды get pods указан не целиком. Индикатором готового работать пода является статус, равный Running и готовность 1/1. Единицей справа от слеша является количество готовых к работе контейнеров. В поле IP указан внутренний IP адрес пода. Очевидно, что используя его невозможно получить напрямую доступ к его содержимому – сетевое пространство подов на узле изолировано от общей сети.

Самый простой способ осуществить доступ к поду – пробросить порты с основной машины в под. Для этого необходимо выполнить команду:

```
kubectl port-forward nginx 8888:80
```

где 8888 – порт на локальной машине (можно выбрать любой другой из свободных) и 80 – порт внутри пода.

Теперь попробуем получить доступ к созданному поду. Для этого из соседнего терминала вызовем команду

```
curl localhost:8888
```

В результате на экране должна отобразиться информация с веб-страницы по умолчанию, расположенной на веб сервере, запущенном в поде.

Однако под 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. Контроллер репликации постоянно отслеживает количество запущенных подов и если оно меньше или больше указанного в манифесте – запускает недостающие поды или уничтожает лишние.

```
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
        ports:
        - containerPort: 80
```

Обратим внимание на новые параметры. Тип объекта – ReplicaSet, в разделе spec добавилось поле replicas с числом реплик=3.

Наиболее интересные поля – template, содержащий в себе labels и selector, аналогично содержащий matchLabels со значением метки. В параметре selector указываются поды с какой меткой необходимо отслеживать. Все поды создаются в соответствии с шаблоном (template), внутри которого содержится спецификация создаваемых подов и указывается базовый образ и метка, совпадающая с меткой в matchLabels.

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

Для того, чтобы удалить набор реплик необходимо выполнить команду:

```
    kubectl delete replicasets.apps nginx
```

Однако, у ReplicaSet есть один существенный недостаток – невозможность автоматически заменить под при изменении их спецификации. Например, изменим 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 текущий под, то он 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 высокоуровневая сущность Deployment, которая не имеет данного недостатка и является самой популярной для развертывания приложений. При создании ресурса такого типа автоматически создается несколько ресурсов типа ReplicaSet, которые управляются уже с помощью Deployment.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
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 текущий объект, и поднимет новый 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 удалить Deployment необходимо выполнить команду:

```
kubectl delete deployments.apps nginx
```

Как было рассказано в теоретической части, для того чтобы получить доступ к приложению возможно использовать еще один объект – сервис. Создадим его с помощью манифеста:

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

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

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

В результате должна быть выведена страница с nginx.

Для того, чтобы избежать проброса портов и предоставить доступ к службе, возможно присвоить типу службы значения NodePort, LoadBalanser, ClusterIP. В рамках лабораторной работы мы рассмотрим только первый способ.

При создании службы типа NodePort на каждом из узлов кластера открывается порт, который перенаправляет весь трафик на порт службы.

```
---
apiVersion: v1
kind: Service
metadata:
 name: nginx-service
spec:
 selector:
   app: nginx
 type: NodePort
```

```
  ports:
    - port: 80
      nodePort: 31234
```

Теперь выяснив IP адрес узла (minikube ip) можно обратиться к созданному сервису напрямую:

```
curl 192.168.49.2:31234
```

где 192.168.49.2 – адрес узла.

В завершении приведем пример манифестов для создания постоянных хранилищ и их подключения к поду. Т.к. minikube является одноузловым кластером, то директория для хранения будет создаваться непосредственно на узле. Подчеркнем, что для многоузлового кластера такой подход хоть и допустим, но накладывает большие ограничения, поэтому не применяется.

Для подключения к узлу minikube выполним команду:

```
minikube ssh
```

Внутри узла создайте директорию, например по пути /mnt/data

```
sudo mkdir /mnt/data
```

Поместите в созданную директорию файл с web страницей.

```
sudo sh -c "echo 'Hello MPSU!' > /mnt/data/index.html"
```

Отсоединитесь от узла с помощью команды *exit*.

Создайте постоянное хранилище с помощью следующего манифеста

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nginx-pv-volume
spec:
  storageClassName: “”
  capacity:
    storage: 10Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Тип хранилища – PersistentVolume – постоянное хранилище. Класс хранения – manual. Классы хранения (**Storage Classes**) предназначены для определения провайдера хранилища, примененных параметров, правил резервного копирования и т.п. В данном примере используется пустое имя для класса хранения, т.к. провайдер не используется.

Типы доступа (**accessModes**) у PV могут быть следующие:

- ReadWriteOnce – том может быть смонтирован на чтение и запись к одному поду.
- ReadOnlyMany – том может быть смонтирован на много подов в режиме только чтения.
- ReadWriteMany – том может быть смонтирован к множеству подов в режиме чтения и записи.

Вместимость хранилища – 10 Мб. В качестве пути указывается /mnt/data – директория внутри узла, к которой будет осуществлено монтирование.

Создайте заявку на использование созданного хранилища

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nginx-pv-claim
spec:
  storageClassName: nginx-sc-example
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Mi
```

В заявке указывается имя класса хранения и аналогично тип доступа. Далее запрашивается необходимый объем.

Создайте новый под с nginx и подключите к нему созданную заявку

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  volumes:
    - name: nginx-pv-storage
      persistentVolumeClaim:
        claimName: nginx-pv-claim
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: nginx-pv-storage
```

В параметре mountPath указывается директория внутри контейнера, которая будет соединена с хранилищем. В данном примере это директория, хранящая веб страницы для nginx.

После создания пода осуществите проброс портов (см. выше) и запросите web страницу. Убедитесь, что будет возвращен текст Hello MPSU!

Исследовательская часть

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 2/3)
- 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 полезно, так как он содержит информацию о том, какие этапы развертывания были пройдены, 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), 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 для kubectl, чтобы он мог подключаться к API-серверу кластера для выполнения вводимых команд. (**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 (его API-сервер) и сервис внутрикластерного DNS. Следующим шагом 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 сможем, даже если попросим более подробный вывод в том же табличном формате: *kubectl get nodes -o wide*. Отложим пока эту ситуацию со статусом узла на некоторое время в сторону и попробуем выяснить, какие поды запущены сейчас в кластере. Выполним для этого 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 отличном от default пространства имен, его надо напрямую указать в запросе kubectl. Чтобы понимать, какие у нас пространства вообще имеются "из коробки", получим их перечень

```
kubectl get namespaces
```

В выводе помимо уже знакомого default будет особо интересующее нас *kube-system*, поскольку в нем располагаются служебные объекты, в том числе созданные на этапе инициализации и запуска кластера, а значит именно там мы должны найти поды с компонентами кластера.

Попробуем еще раз получить информацию о имеющихся подах, только теперь уже с конкретикой, что мы хотим получить данные для 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 не запустились, а зависли в состоянии ожидания (*Pending*). Давайте узнаем причину происходящего. В этом нам поможет 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 в ширину 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 полученного 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 пора узнать, как 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, теперь когда мы ознакомились с несколькими очень полезными командами, настало время их применить для того, чтобы разобраться, что же не так с нашим единственным узлом, почему его статус не *Ready*. Для этого нам снова поможет команда *describe* которая выведет всю информацию об узле:

```
kubectl describe nodes
```

В полученном 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, "висящих" на узле (секция Taints). Зафиксируйте информацию о том, какое еще ограничение, помимо найденного в событиях пода CoreDNS ранее, висит на узле (преподаватели СПРОСЯТ 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 такого допуска не имеют)*.

Теперь, когда мы выявили причину текущих бед с кластером, надо подумать об установке и настройке CNI-плагина, который будет отвечать за сеть подов, а если быть точнее, за выделение каждому поду IP адреса, за обеспечение связности между подами на разных узлах (посредством механизмов маршрутизации и инкапсуляции) и опционально за применение сетевых политик, разграничивающих доступ. В данной работе будет использован 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 "серой" подовой подсети, поверх навешивается еще один IP-заголовок, только уже с адресом "физического" интерфейса узла, 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, за каким именно узлом находится подсеть подов адреса назначения.

Приступим к установке, она пройдет в два этапа. Сначала установим специальный оператор Kubernetes от разработчиков Calico - Tigera, который позже развернет нам все требуемые компоненты Calico согласно переданным ему параметрам установки. Оператор в Kubernetes - это приложение, которое следит за установкой и осуществляет в первую очередь управление (создание, запуск и изменение и удаление) кастомными (типа CustomResourceDefinition), а при необходимости еще и ресурсами стандартных типов, помогает отслеживать изменения и поддерживать эти ресурсы в желаемом состоянии. Tigera operator, который мы установим, осуществляет полное управление жизненным циклом Calico в кластере k8s, помимо управления своими кастомными, создает объекты ресурсов стандартных (Namespace, Deployment, DaemonSet,...) типов, масштабирует поды при необходимости, через него мы можем Calico устанавливать, изменять его конфигурацию, обновлять версию и тд.

Следующей командой, применив манифест установки Tigera operator, создадим все необходимые для него 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. Стоит сказать, что этот под разворачивается не сам по себе, как самостоятельный объект, а как экземпляр (реплика) внутри набора, созданный согласно шаблону пода из спецификации

объекта типа Развертывание (Deployment), созданного манифестом 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 такое положение дел можно получив список существующих объектов ReplicaSet и Deployment, заодно подметив определенную правила в именовании созданных для развертывания наборов реплик и подов, а также обратив внимание на параметр *Controlled By:* в выводе kubectl describe.

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

Мы можем подождать, пока под *tigera-operator*-а полностью запустится, однако больше никаких изменений в кластере не произойдет, проблема с отсутствием работающей сети подов останется на своем месте. А все дело в том, что пока мы не создадим объект, содержащий параметры установки Calico, оператор ничего делать не будет. Нужный манифест для кастомного ресурса конфигурации Calico был создан Ansible на подготовительном этапе и находится на мастере в файле *~/k8s/calico.yaml*. Просмотрите содержимое данного файла, наиболее важная информация там - выделенная для подов IP-подсеть (podCIDR), которая совпадает с указанной в параметре *podSubnet* в конфигурации kubeadm при 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 поды с именами *calico-node...*, которые принадлежат другому типу высокоуровневого ресурса - *DaemonSet.* Этот ресурс удобен в первую очередь для запуска служебных приложений, поскольку гарантирует запуск по одному экземпляру пода на каждом из узлов кластера. Просмотреть существующие объекты такого типа можно по аналогии с развертыванием (deployment), командой *kubectl get daemonsets -A* Подождите, пока все созданные оператором поды проинициализируются и придут в состояние готовности, после этого инициализация CNI завершится, статус узла кластера в выводе *kubectl get nodes* станет *Ready* и запустятся поды с CoreDNS. Давайте дополнительно убедимся, что сеть подов в самом деле заработала, 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 при желании найти еще и ответ на вопрос, почему некоторые поды успешно работали с самого начала, до момента запуска CNI-плагина (**зафиксируйте** свое предположение по этому поводу для отчетности). Ну а теперь, раз все работает, значит мы готовы расширить наш кластер, чтобы он мог называться кластером 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 поды на мастере).

Добавьте в кластер второй рабочий узел, действуя аналогично первому, дождитесь запуска на втором узле всех развернутых подов.

Раз мы упомянули выше про возможный недостаток вычислительных ресурсов на узлах, давайте попробуем получить данные о текущей загрузке узлов кластера, используя сцпециализированную 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 вкупе с YAML-манифестами, но это не единственный способ, и в некоторых случаях (например, для сложных микросервисных приложений) точно не самый простой. Для Kubernetes существует свой "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 ресурсах и загрузке узлов используя команду kubectl top (а также необходим для функционирования горизонтального автомасштабирования нагрузок), для начала добавим нужный репозиторий и скачаем из него файл, содержащий значения параметров установки, поскольку в них 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 отдельное пространство имен и можно приступать к установке

```
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, попробуйте 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 общего постоянного хранилища для развернутых в кластере Stateful-приложений, которые, как известно, должны сохранять свои данные между перезапусками, в том числе при "переезде" пода на другой узел. Напомним основные положения. По аналогии с Docker, хранилища, монтируемые внутрь контейнера в поде называются томами (volume). Том в контексте Kubernetes может быть обычным "временным" (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 типов доступа *ReadOnlyMany / ReadWriteMany*. Для подключения томов мы можем использовать как встроенные в k8s драйвера некоторых из 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) конкретного типа хранилища и при необходимости, внешний CSI-драйвер для предоставления кластеру Kubernetes доступа к этому хранилищу.

Далее мы продемонстрируем на практике оба упомянутых варианта, полностью "ручной" и "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 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 его, 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. В качестве логина pgAdmin укажите admin@mpsu.stu, пароля – postgres. При подключении к базе данных укажите postgresql как доменное имя сервера, пользователь postgres, пароль – pass.

Практико-ориентированная часть

1. Разверните кластер Kubernetes, состоящий из трёх узлов
2. Выполните деплой разработанного проекта в кластер
3. Убедитесь, что приложение работает корректно, мониторинг осуществляется.

Список рекомендованной литературы:

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/
12. Kubernetes в действии. Главы 1-11.