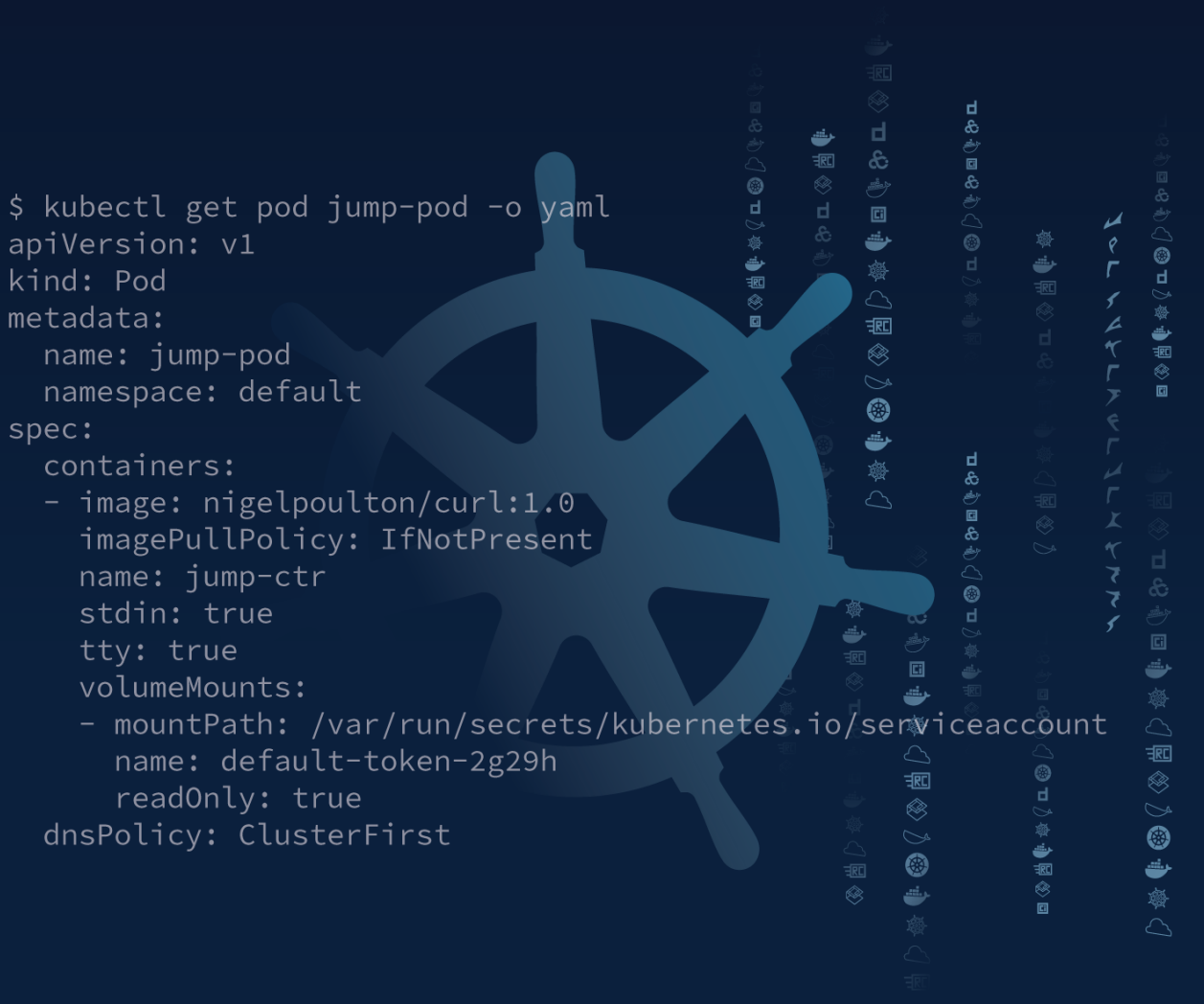

Kubernetes на практике

Издание 2022

Найджел Пультон
& Пушкар Джоглекар

Kubernetes на практике

Nigel Poulton и Alexey Pyltsyn

Эта книга предназначена для продажи на http://leanpub.com/thekubernetesbook-russian

Эта версия была опубликована 2021-12-15

Это книга с Leanpub book. Leanpub позволяет авторам и издателям участвовать в так называемом Lean Publishing - процессе, при котором электронная книга становится доступна читателям ещё до её завершения. Это помогает собрать отзывы и пожелания для скорейшего улучшения книги. Мы призываем авторов публиковать свои работы как можно раньше и чаще, постепенно улучшая качество и объём материала. Тем более, что с нашими удобными инструментами этот процесс превращается в удовольствие.

Оглавление

0: Предисловие

Kubernetes не стоит на месте и до сих пор активно разрабатывается. Поэтому для себя я решил обновлять книгу каждый год. Под обновлением я имею в виду повторную вычитку каждой главы, где также каждый пример будет протестирован и актуализирован в соответствии с последней версии Kubernetes. Не сомневайтесь, я полон решимости добиться того, чтобы эта книга по Kubernetes считалась лучшей в мире.

Если ежегодное обновление кажется вам чем-то излишним… что ж, добро пожаловать в новый мир.

Мы живем в быстроизменяющимся мире, где от книги по Kubernetes, выпущенной два года назад, будет мало пользы. Не поймите меня неправильно, как автор, я бы и сам прежде всего хотел, чтобы написанная мною книга была актуальной в течение многих лет. Но сейчас информация быстро устаревает. Так что, привыкайте к новой реальности.

Репозиторий книги на GitHub

Используемые в книге примеры и YAML-код можно найти в репозитории на GitHub.

```
1 https://github.com/nigelpoulton/TheK8sBook
```

Рекомендуется, хотя и не обязательно, склонировать репозиторий локально (на жаргоне это означает скопировать его на свой компьютер). Для этого потребуется установить `git`, а затем выполнить следующую команду.

```
1 $ git clone https://github.com/nigelpoulton/TheK8sBook.git
```

В директории, где выполнялась команда, должна появиться новая директория `TheK8sBook` со всеми файлами, необходимыми для воспроизведения примеров из книги.

Обратная связь

Если вам понравилась моя книга, я буду рад, если вы оставите отзыв и оцените книгу на Amazon. Конечно, не настаиваю на этом — все мы занятые люди.

Вы можете связаться со мной по следующим ссылкам:

- twitter.com/nigelpoulton
- nigelpoulton.com
- linkedin.com/in/nigelpoulton
- youtube.com/nigelpoulton

Если вы хотите предложит улучшение или сообщить об ошибке, напишите мне на электронную почту по адресу `tkb@nigelpoulton.com`. Я постараюсь ответить.

Приятного чтения!

2: Принципы работы Kubernetes

В этой главе вы узнаете про основные составляющие, необходимые для создания кластера Kubernetes и развертывания приложения. Это будет краткий обзор главных понятий. Не переживайте, если вы не сразу все поймете, в дальнейших главах книги будут даны подробные объяснения, а практические примеры помогут закрепить теорию.

В этой главе мы разберем следующее:

- Из чего состоит Kubernetes
- Главные и рабочие узлы
- Упаковка приложений для Kubernetes
- Декларативная модель и желаемое состояние
- Поды
- Деплойменты
- Сервисы

Kubernetes

По большому счёту Kubernetes это:

- Кластер для запуска приложений
- Оркестратор облачно-ориентированных микросервисных приложений

Kubernetes как кластер

Kubernetes очень похож на любой другой кластер – совокупность машин, на которых размещаются и работают приложения. Обычно они называются “узлами”, к которым относятся не только физические серверы, но и виртуальные машины, облачные экземпляры, Raspberry Pi и т.д.

Кластер Kubernetes состоит из *управляющего уровня (control plane)* и *узлов (nodes). Управляющий уровень* открывает доступ к API, имеет планировщик для распределения рабочей нагрузки и записывает состояние кластера и приложений в постоянное хранилище. На *узлах* собственно и выполняются все приложения.

Управляющий уровень можно представить себе как мозг кластера, а *узлы* — мускулы. В этой аналогии управляющий уровень является мозгом, поскольку он выполняет такие сложные задачи, как планирование, автоматическое масштабирование и обновление без простоя. Узлы — это мускулы, поскольку берут на себя тяжелую работу по непосредственному выполнению приложений.

Kubernetes как оркестратор

Оркестратор — это такое модное слово для обозначения системы, которая развертывает и управляет приложениями.

Чтобы лучше понять это, проведем небольшую аналогию.

Допустим, возьмём футбольную команду, которая состоит из разных людей. У каждого члена команда есть своя роль – кто-то защищается, кто-то атакует, кто-то отлично пасует, кто-то перехватывает, кто-то бьет… И тут приходит тренер, который определяет позиции каждого игрока, чтобы команда работала на максимуме и достигла успеха. На рисунке 2.1 показано изначальное положение игроков, а на рисунке 2.2 — позиции игроков после того, как тренер начал свою работу.

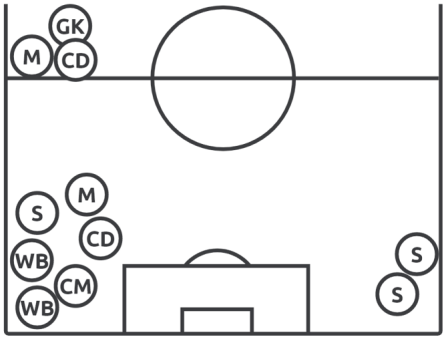

Рисунок 2.1

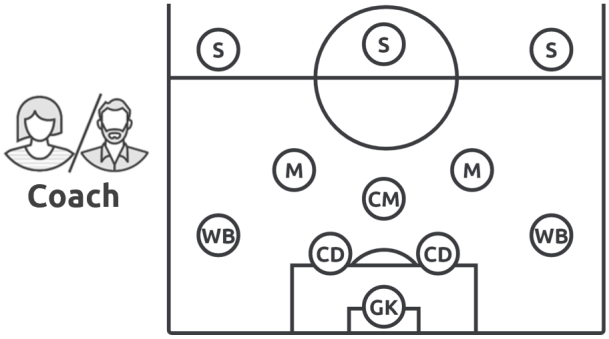

Рисунок 2.2

Тренер также следит за тем, чтобы команда сохраняла правильный настрой, придерживалась плана игры и справлялась с любыми травмами и разными проблема на протяжении всего матча.

И знаете что, всё это не отличается от микросервисных приложений, запущенных в Kubernetes.

Посудите сами…

У вас есть множество разных узкоспециализированных микросервисов. У каждого из них своя задача – вывод веб-страниц, аутентификация, поиск, сохранение данных. Приходит

Kubernetes (своего рода тренер в той футбольной аналогии) и собирает их вместе в полноценное приложение, контролируя его бесперебойную работу.

В спорте это называется *тренерством*, в мире приложений — *оркестровкой*. Поэтому говорят, Kubernetes *оркеструет* облачно-ориентированные микросервисные приложения.

Как работает Kubernetes

Вы создали приложение, упаковали его в контейнер, а затем передали кластеру (Kubernetes). Кластер состоит из одного или нескольких *узлов управляющего уровня* и множества *рабочих узлов (worker nodes)*.

Как отмечалось ранее, на узлах управляющего уровня размещается своего рода мозг, командный пункт кластера. На рабочих узлах работают сами пользовательские приложения.

Итак, процесс запуска приложения в кластере Kubernetes выглядит не таким сложным, для этого вам потребуется:

1. Спроектировать и разработать приложение в виде небольших независимых микросервисов, написанных на ваших любимых языках.
2. Упаковать каждый микросервис в отдельный контейнер.
3. Завернуть каждый контейнер в под Kubernetes.
4. Развернуть поды в кластере с помощью высокоуровневых контроллеров, таких как *Deployments, DaemonSets, StatefulSets, CronJobs и т.д.*

Пока что это только начало книги, поэтому вам всё это мало о чём говорит. Однако стоит упомянуть, что в Kubernetes есть несколько *контроллеров*, которые привносят в поды важные функции – самовосстановление, масштабирование, плавающие обновления и многое другое. Некоторые контроллеры предназначены для приложений без состояния, а другие – для приложений с состоянием. В процессе изучения в книги вы познакомитесь со всеми ними.

Обычно в Kubernetes управление приложениями происходит *декларативным* способом. Смысл в том, что в YAML-файлах вы определяете всё необходимое вам, после чего отправляете их в Kubernetes, а затем сидите спокойно, наблюдая за тем, как Kubernetes выполняет то, о чём вы его попросили.

Но на этом он не остановится. Поскольку декларативным путём Kubernetes указывается, каким должно быть приложение, он может отслеживать приложение, чтобы его состояние соответствовало вашим ранее определённым требованиям к нему. Если что-то будет не так, как вы указали, Kubernetes попытается исправить это.

В общих чертах работа Kubernetes выглядит так. Давайте теперь немного разберем ее поподробнее.

Управляющий уровень и рабочие узлы

Как вы уже знаете, кластер Kubernetes состоит из *узлов управляющего уровня* и *рабочих узлов*. Это Linux-хосты, которые могут быть виртуальными машинами (ВМ), "чистыми" (bare metal) серверами в дата-центре или экземплярами в приватном или публичном облаке. Kubernetes можно запустить также на устройствах ARM и IoT.

Управляющий уровень

Узел управляющего уровня Kubernetes — это сервер, на котором запущено множество системных сервисов, образующих вместе сам управляющий уровень кластера. Такие узлы называются *главными* или *ведущими*, иногда для краткости их именуют *мастерами* (masters).

В простейшей установленном экземпляре Kubernetes используется один главный узел. Однако это больше характерно для песочниц и тестовых окружений. В продакшене жизненно необходимо иметь по несколько главных узлов, тогда их можно будет считать высокодоступными и отказоустойчивыми. Вообще говоря, для высокой доступности рекомендуется использовать три или пять узлов.

Также крайне не рекомендуется запускать рабочие приложения на главных узлах. Только тогда их основная задача будет сконцентрирована на управлении кластером.

Давайте вкратце рассмотрим различные компоненты, из которых состоит управляющий уровень.

API-сервер

API-сервер – это своего рода главный вокзал Kubernetes. **Взаимодействие всех компонентов происходит через API-сервер**. Мы разберёмся с деталями позже, пока что нужно понимать, что как и внутренние системные компоненты, так и внешние пользовательские компоненты, **все** они работают через API-сервер – в общем, *все дороги ведут к API-серверу*.

Он открывает доступ к API на основе RESTful, в который методом `POST` по протоколу HTTPS отправляются конфигурационные YAML-файлы. В этих YAML-файлах, которые ещё называются *манифестами*, описывается желаемое состояние приложения. Например, к желаемому состоянию относятся образ контейнера, порты и количество реплик пода.

Все запросы к API-серверу проходят аутентификацию и авторизацию. После этого конфигурация, содержащиеся в YAML-файле, проверяется на корректность, затем сохраняется в кластерном хранилище, и далее планируется размещение рабочих нагрузок в кластере.

Хранилище кластера

Хранилище кластера является единственной частью управляющего уровня, у которой *есть состояние*. Как ясно по имени, в хранилище содержится вся конфигурация и состояние

кластера. Таким образом, это жизненно важный компонент каждого кластера Kubernetes – без кластерного хранилища нельзя представить кластер.

На данный момент кластерное хранилище использует популярную распределенную базу данных `etcd`. Поскольку это *единственный источник данных* кластера, то для обеспечения высокой доступности нужно иметь от трёх до пяти реплик etcd, вдобавок следует предусмотреть способы восстановления данных, если что-то пойдет не так. По умолчанию Kubernetes устанавливает реплику хранилища кластера на каждом узле управляющего уровня, тем самым делая хранилище высокодоступным.

Что касается *доступности*, то в случае etcd прежде всего важна целостность данных. Это значит, что хранилище не допускает ситуации *split brains*, поэтому остановит обновления кластера, чтобы обеспечить согласованность данных. Однако, если всё же это произойдет, приложения хоть и будут по-прежнему работать, но вы не сможете обновить конфигурацию кластера.

Как и при распределенных базах данных, согласованность записи играет ключевую роль. Например, необходимо обрабатывать несколько операций записи одного и того же значение из разных мест. etcd использует популярный алгоритм консенсуса RAFT для решения этой задачи.

Менеджер контроллеров и контроллеры

Менеджер контроллеров реализует все фоновые контроллеры, которые следят за компонентами кластера и выполняют определённые действия при возникновении событий.

С архитектурной точки зрения, менеджер контроллеров — это *контроллер контроллеров*, поскольку он создаёт все независимые контроллеры и следить за их работой.

Например, Deployment, StatefulSet, ReplicaSet — это всё примеры контроллеров. Каждый из них отвечает за определённую функциональность кластера, работая в фоновом режиме, постоянно отслеживая изменения в API-сервере.

В целом, все контроллеры нацелены на то, чтобы *наблюдаемое состояние* кластера соответствовало *желаемому состоянию*.

Логика работы каждого из контроллеров описана ниже, она же лежит в основе всего Kubernetes:

1. Получение желаемого состояния
2. Наблюдение за текущим состоянием
3. Определение различий между желаемым и текущим состоянием
4. Устранение найденных различий

Каждый контроллер предназначен для выполнения своей четко определенной задачи и поэтому ответственен только за свою маленькую часть в кластере Kubernetes. Таким образом, один контроллер ничего не знает о другом контроллере – каждый из них занимается своим делом. Это основа, заложенная в Kubernetes и соответствует философии Unix, когда сложная система состоит небольших узкоспециализированных частей.

Планировщик

Основная задача планировщика следить за новыми рабочими нагрузками на API-сервере и распределять их по узлам. Под его капотом скрывается сложная логика, которая отсеивает неработоспособные узлы, а затем ранжирует функционирующие узлы. Само ранжирование тоже непростое, так что для работы выбирается узел с самым высоким количеством баллов.

При определении узлов, на которых можно что-то запустить, планировщик выполняет базовые проверки. Например, не был ли испорчен узел, есть ли у него правила аффинити или анти-аффинити, свободен ли на нём требуемый сетевой порт, достаточно ли у него доступных ресурсов и т.д. В общем, игнорируется любой узел, который не подходит для выполнения текущей задачи, а оставшиеся узлы ранжируются по таким параметрам, как наличие нужного образа, количество свободных ресурсов и т.п. Каждый из них оценивается в баллах, и узел с наибольшим количеством баллов выбирается для выполнения задачи.

Если планировщик не находит нужный узел, задача переходит в режим ожидания.

Важно понимать, что планировщик не отвечает за *выполнение* задач, он только ищет узлы, на которых они должны быть выполнены. *Задача* обычно представляет собой под/контейнер. В следующих главах вы узнаете подробнее о подах и контейнерах.

Облачный менеджер контроллеров

Если вы создаёте кластер в публичном облаке, таком как AWS, Azure, GCP или Linode, в управляющем уровне будет работать *облачный менеджер контроллеров*. Его задача состоит в том, чтобы облегчить интеграцию с облачными сервисами – экземплярами, балансировщиками нагрузки и хранилища. Например, если ваше приложение запрашивает балансировщик нагрузки с выходом в интернет, облачный менеджер контроллеров выделит балансировщик нагрузки из вашего облака и подключит его к вашему приложению.

Резюме по управляющему уровню

Узлы управляющего уровня Kubernetes — это серверы, на которых выполняются сервисы управляющего уровня кластера. Эти сервисы выполняют роль мозга кластера, где принимаются все решения по управлению и планированию. К сервисам относятся API-сервер, кластерное хранилище, планировщик и специализированные контроллеры.

API-сервер — это точка доступа к управляющему уровню, через него проходит всё взаимодействие и команды. По умолчанию он работает на порту 443.

На рисунке 2.3 показана схема узла управляющего уровня Kubernetes.

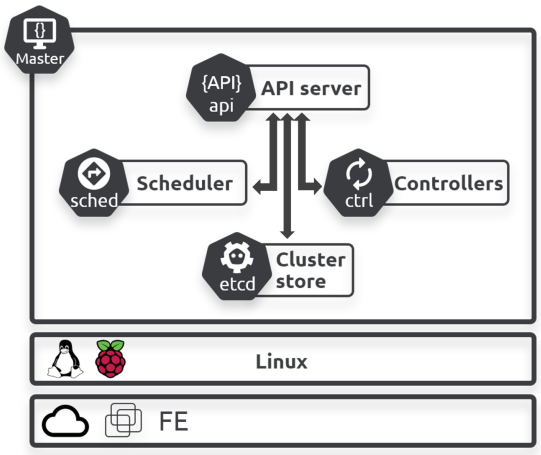

Рисунок 2.3: узел управляющего уровня

Рабочие узлы

Узлы (nodes, или ноды) — это серверы, на которых выполняются сами пользовательские приложения в кластере Kubernetes.

По большому счёту их рабочий процесс состоит из трёх шагов:

1. Отслеживают появление новых задач на API-сервере
2. Выполняют эти задачи
3. Сообщают о выполнении задачи управляющему уровню (через API-сервер).

Как видно на рисунке 2.4, они немного попроще, чем *узлы управляющего уровня.*

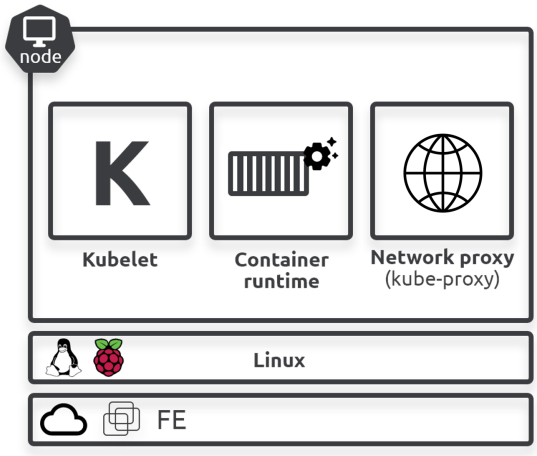

Рисунок 2.4: узел Kubernetes

Давайте рассмотрим три основных компонента узла.

Kubelet

kubelet — основной агент Kubernetes, который запускается на каждом узле кластера. Зачастую даже под *узлом* и *kubelet* имеют в виду одно и то же.

При добавлении узла к кластеру, устанавливается kubelet, который затем отвечает за регистрацию узла в кластере. Этот процесс регистрирует процессор, память и хранилище узла в более широком пуле кластера.

Одна из основных задач kubelet — отслеживать появления новых рабочих заданий на API-сервере. Когда он обнаружит новое задание, то начинает выполнять его, вместе с этим уведомляя о ходе выполнения управляющему уровню.

Если kubelet не может выполнить задание, он сообщает об этом управляющему уровню, который уже решает, какие дальнейшие действия нужно предпринять. Например, если kubelet не может выполнить задачу, он **не** станет искать другой узел, на котором можно её выполнить. Вместо этого он доложит об этом управляющему уровню, который будет должен определить, что делать дальше.

Среда выполнения контейнеров

kubelet не может работать без *среды выполнения контейнеров*, чтобы выполнять задачи, связанные с контейнерами, например, загрузка образов, запуск и остановка контейнеров.

Поначалу в Kubernetes была встроенная поддержка Docker. Но со временем он перешёл на модель с использованием плагинов под названием Container Runtime Interface (CRI). CRI скрывает внутренние механизмы Kubernetes и даёт понятный и документированный интерфейс, чтобы как можно легче можно было подключить сторонние среды выполнения контейнеров.

Сейчас Kubernetes отказывается от поддержки Docker в качестве среды выполнения контейнеров. Это связано с тем, что Docker стал довольно громоздким и не совместим с CRI (поэтому требуется дополнительный код для исправления этого недостатка). На его замену в Kubernetes приходит containerd как наиболее популярная среда выполнения контейнеров.

> **Примечание**: `containerd` — это супервизор контейнеров и среда выполнения, код которой был взят из Docker Engine. Он был передан в CNCF компанией Docker, Inc. и пользуется большой поддержкой сообщества. Существуют и другие контейнерные среды выполнения, совместимые с CRI.

Kube-proxy

И последним компонентом *узла* является kube-proxy. Он запускается на каждом узле и отвечает за локальную сеть кластера. Он назначает каждому узлу уникальный IP-адрес, а также создаёт локальные правила iptables или IPVS для маршрутизации и балансировки нагрузки трафика в сети подов. Подробнее обо всем этом мы расскажем далее в книге.

Kubernetes DNS

Помимо различных компонентов управляющего уровня и узлов, в каждом кластере Kubernetes есть внутренняя DNS-сервер, без которого невозможна работа механизма по обнаружению сервисов.

DNS-сервер кластера привязан к статическому IP-адресу, который указан в каждом поде кластера. Таким образом, каждый контейнер и под смогут обратиться к DNS-серверу. Регистрация сервисов также происходит автоматически. То есть со стороны приложений не нужно регистрироваться в механизме обнаружения сервисов Kubernetes.

DNS-сервер кластера создан на основе опенсорс-проекта CoreDNS (https://coredns.io/).

Теперь, когда у вас есть базовые понимание узлов управляющего уровня и рабочих узлов, давайте посмотрим, как упаковывать приложения, чтобы их можно было запустить в Kubernetes.

Упаковка приложений для Kubernetes

Чтобы запустить приложение в кластере Kubernetes нужно проделать три шага, а именно:

1. Упаковать его в контейнер
2. Далее обернуть в под
3. И декларативно развернуть через файл манифеста

Это происходит следующим образом…

Вы пишете микросервисное приложения на вашем языке программирования. Затем вы собираете его в образ контейнера, который затем сохраняете в реестре. Тем самым вы *контейнеризировали* приложение.

Затем вы определяете Pod-объекте в Kubernetes, чтобы запустить своё контейнеризированное приложение. Под — это своего рода обертка, при помощи которой контейнер может работать в кластере Kubernetes. Как только у вас будет файл с манифестом пода, можно приступать к развертыванию приложения в Kubernetes.

Хотя в кластере Kubernetes можно запускать статические поды, лучше всего развертывать все поды с помощью высокоуровневых контроллеров. Среди них наиболее часто используемым является *Deployment*. Он может масштабировать, самовосстанавливать и обновлять приложения. Аналогично поду, манифест *Deployment-объекта* определяется в YAML-файле, в котором можно указать, сколько реплик нужно развертывать и как выполнять обновления.

На рисунке 2.5 показан код приложения, упакованный в *контейнер*, который работает внутри *пода*, созданный *Deployment-контроллером*.

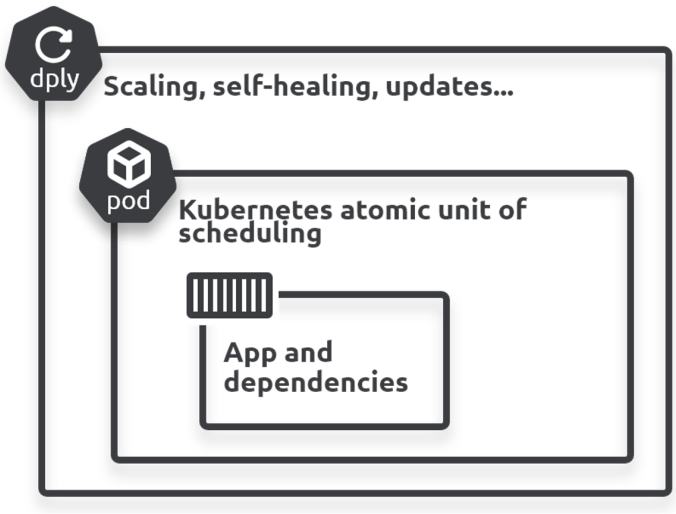

Рисунок 2.5

После того, как объект *Deployment* определён в YAML-ƒайле, можно воспользоваться вы можете инструментом командной строки Kubernetes, чтобы отправить его на API-сервер – это будет ваше *желаемое состояние* приложения, которое Kubernetes будет поддерживать.

Кстати о желаемом состоянии…

Декларативная модель и желаемое состояние

Декларативная модель и концепция *желаемого состояния* являются краеугольным камнем Kubernetes. Поэтому очень важно понимать эти термины.

В Kubernetes декларативная модель работает следующим образом:

1. В ƒайле манифеста определяется желаемое состояние приложения.
2. Далее ƒайл с определением отправляется на API-сервер.
3. Kubernetes сохраняет его в кластерном хранилище в виде *желаемого состояния* приложения
4. Первоначально Kubernetes размещает приложение в кластере в соответствии с его желаемым состоянием.
5. Затем контроллер следит за тем, чтобы *наблюдаемое состояние* приложения не отличалось от *желаемого.*

Давайте рассмотрим каждый шаг немного подробнее.

Манифесты пишутся на простом языке разметки YAML и сообщают Kubernetes, каким должно быть приложение. Это называется *желаемым состоянием.* Например, к нему относятся используемый образ, количество реплик, какие порты нужно открыть, или как выполнять обновления.

После создания манифеста, его отправляют на API-сервер. Проще всего это сделать с помощью утилиты командной строки `kubectl`. Она передаст манифест управляющему уровню через HTTP-запрос методом POST, как правило, на порт 443.

Каждый запрос проходит проверки аутентификации и авторизации, после них Kubernetes определяет, какому контроллеру нужно переадресовать манифест (например, *контроллеру Deployments*), а затем сохраняет манифеста в хранилище кластера как часть *желаемого состояния*. После этого на узлах кластера начинают выполняться рабочие задания, на этом этапе kubelet координирует работу по загрузке образов, запуску контейнеров, подключению к сетям и запуску процессов приложения.

В завершение, контроллеры неустанно следят за состоянием кластера. Если *наблюдаемое* (текущее) состояние отличается от *желаемого*, Kubernetes пытается самостоятельно исправить все различия.

Здесь ещё важно понимать, что этот процесс идёт в разрез с традиционной *императивной моделью*. При императивном подходе вам бы пришлось иметь дело с длинными скриптами, содержащими команды, специфичные для определенной платформы, которые бы делали то, что мы описали.

Декларативная модель не только намного проще скриптов с огромным количеством императивных команд, она также предлагает такие возможности как самовосстановление, масштабирование, контроль версий и самодокументирование. И всё это можно *настроить в файле манифеста*. Если что-то будет отличаться от того, что было описано в манифесте, соответствующий контроллер попытается устранить это несоответствие.

> **Терминологическая справка**: *наблюдаемое состояние*, *фактическое состояние* и *текущее состояние* — всё это слова-синонимы.

Для лучшего понимания рассмотрим пример.

Пример с декларативной моделью

Предположим, что у вас есть приложение с желаемым состоянием, включающее десять реплик пода с веб-фронтендом. Если узел с двумя репликами выйдет из строя, *наблюдаемое состояние* уменьшится до восьми реплик, однако *желаемое состояние* по-прежнему будет десять. Поэтому данное отличие заметит контроллер, и Kubernetes создаст две новые реплики, чтобы их общее количество снова стало десять.

То же самое произойдет, если вы намеренно увеличите или уменьшите желаемое количество реплик. Можно даже изменить образ, который вы хотите использовать (это называется выкладкой). Например, если приложение в настоящее время использует образ с версией `v2.00`, а вы обновите желаемое состояние, указав версию `v2.01`, соответствующий контроллер заметит разницу и обновит кластер, чтобы все десять реплик работали на новой версии.

То есть вместо того, чтобы писать достаточно сложный скрипт, который бы обновлял каждую реплику до новой версии, вам достаточно сообщить Kubernetes, что вы хотите использовать новую версию, и Kubernetes выполнит эту тяжелую работу за вас.

Несмотря на то, что это представляется таким простым, это очень действенный процесс, лежащий в самой основе работы Kubernetes.

Поды

При использовании VMware рабочей единицей является виртуальная машина (ВМ), в Docker ей будет контейнер, а в Kubernetes — *Pod-объект (под)*.

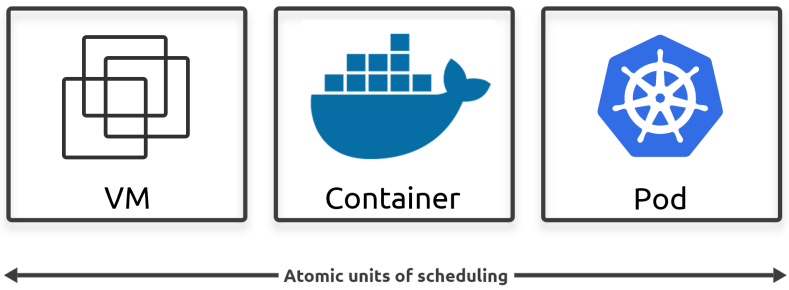

Рисунок 2.6

Kubernetes может запускать контейнеризированные приложения, однако каждый контейнер запускается внутри специального Kubernetes-объекта под названием Pod.

Поды и контейнеры

Прежде всего стоит отметить, что термин *Pod* происходит от от английского *pod of whales*, что значит *стадо китов*. Поскольку на логотипе Docker изображен кит, Kubernetes решил, что будет уместно назвать им объект, вот так и появился *Pod*.

Наиболее простой сценарий их применения — запуск одного контейнера в каждом поде. Именно зачастую “под” и “контейнер” используются как слова-синонимы. Однако, конечно, бывают ситуации, когда в одном поде запускается несколько контейнеров. В качестве примеров многоконтейнерных подов можно привести:

- Сервисные сетки (service meshes).
- Веб-контейнеры, используемые в связке с *вспомогательным контейнером*, обновляющий информацию.
- Контейнеры с тесно связанным с ними сборщиками логов.

В общем, под в Kubernetes — это конструкция для запуска одного или нескольких контейнеров. На рисунке 2.7 показан многоконтейнерный под.

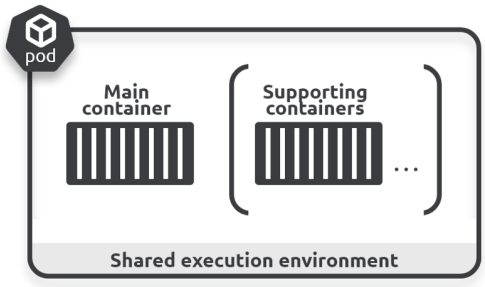

Рисунок 2.7

Анатомия пода

Под представляет собой изолированное окружение для запуска контейнеров. Сами по себе поды выполняют приложения – приложения всегда выполняются в контейнерах, а под служит площадкой для запуска контейнеров. Поды можно представить себе как отгороженную область в ОС хоста, у которой есть своя сеть и пространства имен ядра.

Если вы запускаете несколько контейнеров в поде, все они совместно используют одно и то же окружение пода – это и сетевой стек, тома, пространство имен IPC, общая память и многое другое. Это также значит, что все контейнеры в поде будут иметь один и тот же IP-адрес (это IP-адрес самого пода). Наглядно это продемонстрировано на рисунке 2.8:

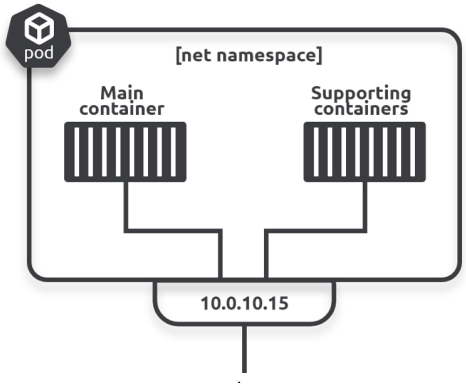

Рисунок 2.8

Если двум контейнерам в одном и том же поде нужно взаимодействовать друг с другом, они могут использовать `localhost` пода, как показано на рисунке 2.9:

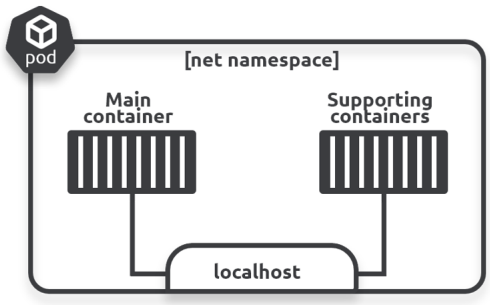

Рисунок 2.9

Многоконтейнерные поды будут идеальным вариантом, если у вас есть тесно связанные контейнеры, которым может потребоваться совместно использовать память и хранилище. Однако если вам не *нужно* тесно связывать контейнеры, то каждый из них лучше всего поместить в собственный под. Тогда всё будет намного проще и понятнее, поскольку каждый под предназначен для выполнения конкретно одной задачи. Однако в таком случае не стоит забывать, что образуется много потенциально незашифрованного сетевого трафика. Чтобы обезопасить трафик между подами и сервисами приложений, вам следует воспользоваться сервисными сетками.

Поды можно масштабировать

Поды относятся к минимальным единицам работы в Kubernetes. Так что если вам нужно масштабировать приложение, вы либо добавляете, либо удаляете поды. Масштабирование приложение *не* происходит путём добавления новых контейнеров в уже существующие поды. Как отмечалось, поды могут иметь несколько контейнеров, но это нужно только в тех ситуаций, когда два разных, но взаимодополняющих контейнера должны совместно использовать ресурсы. На рисунке 2.10 показано, как выглядит масштабирование `nginx` приложения через поды:

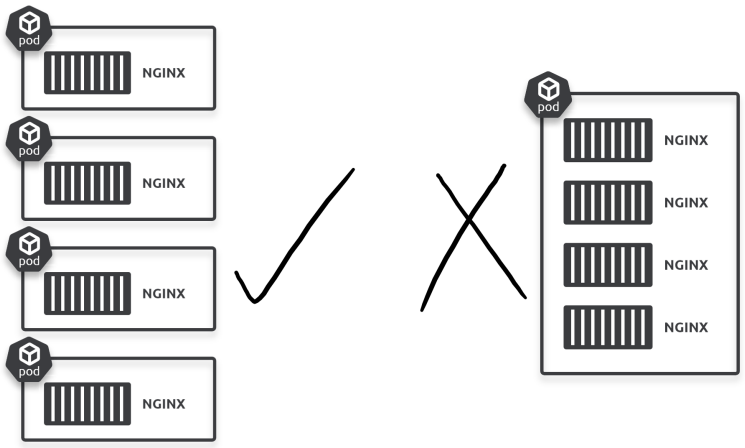

Рисунок 2.10 - Scaling with Pods

Поды как атомарные операции

Развертывание пода представляет собой атомарную операцию. Это значит, что под готов к работе только тогда, когда были запущены все его контейнеры. В таком варианте, если все контейнеры будут успешно созданы, то под можно использовать, если же один из контейнеров не сможет запуститься, то под не будет функционировать.

Один конкретный под может быть размещён только на одном узле – нельзя разместить один и тот же под на нескольких узлах. То же самое относится к многоконтейнерным подам – все контейнеры, запущенные в рамках одного пода, работают на одном и том же узле.

Жизненный цикл пода

Поды создаются, чтобы что-то выполнить, и как только, они это сделали, они уничтожаются. Если под неожиданно вышел из строя, то его не пытаются восстановить. Вместо этого Kubernetes запускает на его месте новый под. Однако хотя новый под работает подобно старому, технически это абсолютно новый под, которому будет назначен новый идентификатор и IP-адрес.

Поэтому стоит учитывать эти моменты при проектировании приложений. Например, не делайте так, чтобы приложения зависели или привязывались к конкретному экземпляру пода. Разрабатывайте приложения с учетом того, что на вместо сломавшегося пода может быть совершенно новый (с новым ID и IP-адресом).

Иммутабельность пода

Кроме всего этого, поды по своей сути иммутабельны – после того, как они были запущены, их нельзя изменить.

Когда под начал работать, вы никогда не сможете поменять его конфигурацию. Если вам нужно изменить или обновить в нём – вы создаете новый под с изменённой конфигурацией. Когда речь шла про *обновление подов*, то это означало удаление старых подов, на месте которых создаются новые.

Деплойменты

Почти всегда вы будете развертывать поды косвенно через высокоуровневые контроллеры. К ним относятся *Deployment*, *DaemonSet* и *StatefulSet*.

Например, Deployment — это объект Kubernetes, служащий обёрткой над подом, и реализующий такие возможности, как самовосстановление, масштабирование, развертывание без простоев и версионированный откат.

Все контроллеры – Deployments, DaemonSets и StatefulSets – постоянно отслеживают кластер, чтобы наблюдаемое состояние соответствовало желаемому.

Сервисы и стабильная сеть

Вы только что узнали, что поды не вечны. Однако, если они находятся под управлением контроллеров, то вышедшие из строя поды, будут заменены на новые. Но это будут поды с совершенно другими IP-адресами. То же самое касается при обновлении и масштабировании. При обновлении старые поды заменяются новыми, у которых будут другие IP-адреса. При масштабировании вверх добавляются новые поды с новыми IP-адресами, а при масштабировании вниз существующие поды удаляются. Подобные события вызывают большой *отток IP-адресов*.

Отсюда можно сделать вывод, что **поды ненадежны**, и это может вызвать трудности…

Предположим, у вас есть микросервисное приложение с кучей подов, выполняющих рендеринг видео. Как можно гарантировать работу других частей приложения, использующих сервис рендеринга, если соответствующие поды будут недоступны?

Именно здесь на помощь приходят *сервисы (services)*, цель которых обеспечить надежное сетевое взаимодействие для набора подов.

На рисунке 2.11 показано, как микросервис загрузчика взаимодействует с микросервисом рендеринга через Service-объект Kubernetes. Сервис даёт подам постоянные имена и IP-адреса. Он также балансирует нагрузку запросов к двум подам рендеринга, находящимися за ним.

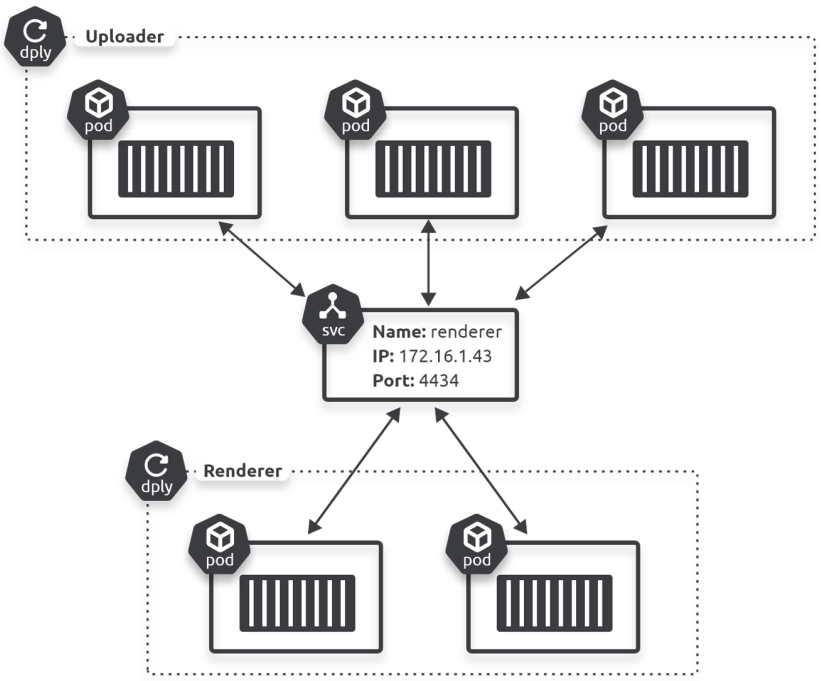

Рисунок 2.11

Сервисы – это полноценные объекты в API Kubernetes – такие же, как поды и деплойменты. У них есть постоянные DNS-имена, IP-адреса и порты. Их работа заключается в балансировке нагрузки между разными подами. Сервис отслеживает доступные поды и обеспечивает надёжную работу сети.

То же самое происходит, если вы увеличиваете или уменьшаете количество подов. Новые поды незаметно для всех добавляются в сервис и начинают получать трафик. Остановленные поды также гладко удаляются из сервиса и на них прекращается идти трафик.

В этом и заключается работа сервиса – пункт доступа к стабильной сети, обеспечивающий балансировку нагрузки по TCP и UDP для постоянно меняющегося набора подов.

Поскольку сервисы работают на уровне TCP и UDP, они *ничего не знают о конечном приложении*. То есть маршрутизацию хостов и путей на уровне приложения сделать не получится. На этот случай есть *Ingress*, который работает по HTTP и следовательно через него можно реализовать маршрутизацию по хосту и пути.

Это были всего лишь основы. Итак, сервисы дают стабильные IP-адреса и DNS-имена в нестабильный мир подов.

Резюме главы

В этой главе мы посмотрели на некоторые основные компоненты кластера Kubernetes.

Узлы управляющего уровня — это серверы, на которых работают компоненты управляющего уровня. Эти серверы быть физическими, виртуальными машинами, облачными экземплярами и даже Raspberry Pi.

Управляющий уровень состоит из нескольких сервисов, включая API-сервер, который открывает доступ к публичному REST-интерфейсу кластера. В продакшен-окружениях для поддержания высокой доступности управляющего уровня, его узлы следует развертывать на нескольких главных узлах.

Узлы — это серверы, на которых работают рабочие приложения. Они также могут быть физическими, виртуальными, облачными экземплярами и Raspberry Pi.

На каждом узле работает специальный сервис kubelet, который регистрирует узел в кластере и взаимодействует с API-сервером. Он отслеживает API-сервер на наличие новых рабочих заданий. Также имеется среда выполнения контейнеров и служба kube-proxy. Среда выполнения контейнеров отвечает за все операции, связанные с контейнерами. Служба kube-proxy отвечает за сетевое взаимодействие на узле.

Мы также поговорили о некоторых основных API-объектах в Kubernetes: Pod (поды), Deployment (деплоймент) и Service (сервис). Под — это основной кирпичик Kubernetes, внутри которого работают контейнеры приложения. С помощью деплойментов возможны самовосстановление, масштабирование и обновление приложения. Сервисы задают стабильную работу сети и реализуют базовую балансировку нагрузки.

Сейчас мы только прошлись по верхам, так что в последующих главах подробнее разберемся во всём, что узнали из этой главы.