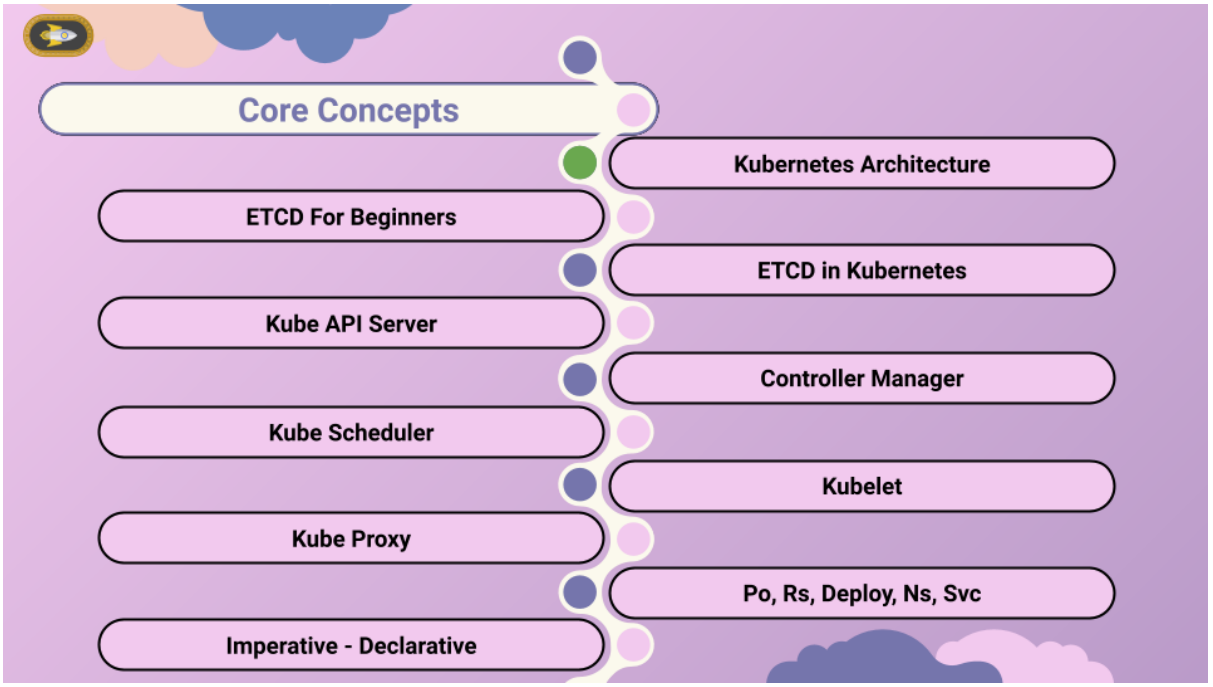


Привет и добро пожаловать в эту лекцию.
Мы начинаем с обзора базовой архитектуры кластера Kubernetes.

Вначале курса мы посмотрим на эту архитектуру что называется "с вертолета", а затем начнем погружение в каждый из компонентов этой системы.

Мы обсудим их роли, их обязанности, как их настроить. В процессе станет понятно зачем эти компоненты друг другу и почему они почти не ссорятся между собой.

Цель Kubernetes - развернуть наши контейнеризированные приложения на доступную физическую или виртуальную вычислительную среду, разместить их столько, сколько нам требуется в данный момент и одновременно обеспечить им возможность общаться между собой в той манере, в какой требуется нашему приложению.

Когда просто читаешь документацию и видишь названия компонентов - это просто сухие термины и они не очень-то заходят. Я решил использовать аналогию с космической станцией, которая распределяет потоки грузов на одном из галактических торговых путей.

Итак, давным-давно в далекой-далекой галактике..

Там существовало два вида кораблей:
 - один из них это трудяга контейнеровоз, его задача была доставить груз от станции загрузки до адресата
 - контрольная станция - это корабль отвечал за управление и мониторинг состояния перевозчиков

Его задача была удостовериться, что перевозчики загружены, заправлены, и имеют правильные маршруты. Также эта станция смотрела, чтобы капитаны кораблей-перевозчиков были на своих рабочих местах, а если они вдруг они пропадали, то могла заменить его новым кораблем.

Кластер Kubernetes состоит из набора `nodes`, которые могут быть физическими или виртуальными, on-premise или облачными хостами приложений с контейнерной нагрузкой.

Это похоже на нашу космическую станцию.
По этой аналогии `worker nodes` в кластере это как бы корабли-дальнобойщики, в которые загружают контейнеры.

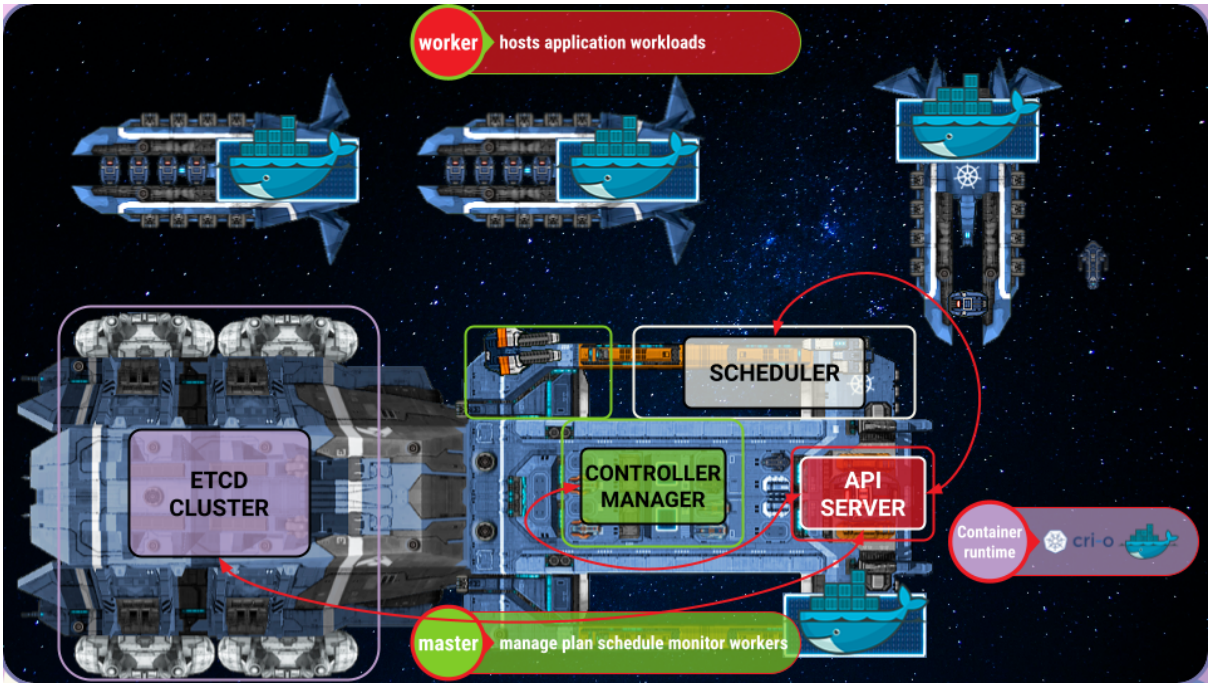
Но сами перевозчики не умеют грузить в себя контейнеры. У них нет журнала заказов, и они не знают, кому какой контейнер везти, а также в какое время это делать. У них нет доступа к плану отгрузки и, быть может такой же груз, который он хочет взять, уже был отправлен адресату днем ранее и везти его уже не требуется.

В нашей космической компании эти задачи решает контрольная станция. У нее есть большой магнитный кран для погрузки контейнеров на судна-грузовики, специальные портовые команды сопровождения, которые решают разные мелкие задачи, связанные с погрузкой и большой командный центр, где специальные отделы

занимаются распределением, сопровождением и мониторингом перевозки контейнеров своим конечным потребителям.

Такой командный корабль в Kubernetes - это `master node`. На нем расположены контрольные компоненты, т.е. те командные центры, абордажные команды и краны по аналогии с космической станцией.

Этот главный узел отвечает за управление кластером, он хранит информацию о состоянии всех компонентов кластера, занимается планированием контейнеров на ноды, мониторит состояние нод и контейнеров в них. Вся информация стекается сюда и здесь формируются управленческие решения. За это ответственен набор компонентов, которые все вместе называются `controlplane`.



Теперь давай посмотрим на каждый из этих компонентов.

Ок, в течение дня через наш порт проходит множество кораблей и на каждом из них много контейнеров. Какие-то загружаются, какие-то выгружаются.
Нам нужно вести учет всех этих операций, иначе мы очень скоро перестанем понимать, что у нас находится в порту, а что уже улетело к заказчику.
Т.е. мы должны знать какой контейнер сейчас находится в каком корабле, в какое время его загрузили и т.п.

В Kubernetes данные о контейнерах хранятся в высокодоступном хранилище типа `ключ-значение` под названием ETCD. ETCD - это NoSQL база данных, популярный фреймворк, который хорошо работает под высокой нагрузкой, умеет масштабироваться и т.д. Мы более подробно поговорим о том, что такое кластер ETCD, как он хранит данные и как его установить в следующих лекциях.

Когда корабли-перевозчики прибывают, мы загружаем на них контейнеры с помощью крана. Эти специальные краны, определяют правильный корабль, на который требуется погрузить выбранный контейнер. Кран определяет нужное судно на основе его размера, вместимости, количества контейнеров, уже находящихся на борту, или любых других условий, нарпимер пункта назначения судна или типа контейнеров, разрешенных к перевозке на этом судне.

Итак, в кластере Kubernetes у нас этим краном будет `scheduler`. Этот планировщик определяет правильную ноду для развертывания там контейнера.
Он принимает во внимание требования приложения к ресурсам, емкость рабочих нод или любые другие политики или ограничения, такие как `taints` и `tolerations`, правила `node affinity`.

Мы рассмотрим их более подробно с примерами и практическими тестами позже в этом курсе.
На самом деле важная важная тема и у нас есть целый раздел только о скедулинге.

Еще на нашей станции есть разные портовые команды, которым назначаются специальные задачи.
Содержанием этих команд и нарезкой им задач руководит офис в этой огромной пушке.

Например, операционная группа заботится об управлении движением транспортных судов и т. д.
Они реагируют и пытаются решить проблему перевозчика если вдруг изменился маршрут или что-то нехорошее произошло с судном.

Команда транспортировки грузов заботится о контейнерах, когда они вдруг при транспортировке повреждаются или выпадают из корабля.
Еще они следят за тем, чтобы новые контейнеры были доступны.
Также есть сервисная группа, которая заботится о связи между всеми кораблями.

Точно так же в Kubernetes у есть контроллеры, которые заботятся о разных областях.

`Node-controller` заботится об узлах.
Он несет ответственность за подключение новых нод к кластеру, обрабатывает ситуации, когда узлы становятся недоступными или уничтожаются, а `replication controller` гарантирует, что желаемое количество контейнеров постоянно будет работать в отдельной группе репликации.
Компонент Kubernetes `kube-controller-manager` объединяет все эти контроллеры.

Итак, мы видели разные компоненты, такие как большие орудия и портовые команды, разные корабли, краны, хранилища данных.

Но как они общаются друг с другом?

Как одна команда сможет согласовать свои действия с другой и кто управляет этим общением на высоком уровне?

`Kube-apiserver` - это основной компонент управления в Kubernetes. Сервер kube-api отвечает за оркестрацию всех операций в кластере.

Он предоставляет Kubernetes API, который используется различными контроллерами для мониторинга состояния кластера и внесения необходимых изменений по мере необходимости. С помощью этого API рабочие ноды связываются с сервером, отчитываясь о своем состоянии. Также благодаря API внешние пользователи могут выполнить операции управления в кластере.

Сейчас мы живем и работаем в контейнерном мире. Контейнеры есть везде, поэтому нам нужно, чтобы все было совместимо с контейнерами.

Наши приложения представляют собой контейнеры различных сервисов, расположенные на рабочих нодах.

И на мастер-узлах мы тоже можем разместить компоненты всей системы управления в контейнерном виде.

Сетевые решения, вроде служб DNS, также могут быть развернуты в виде контейнеров.

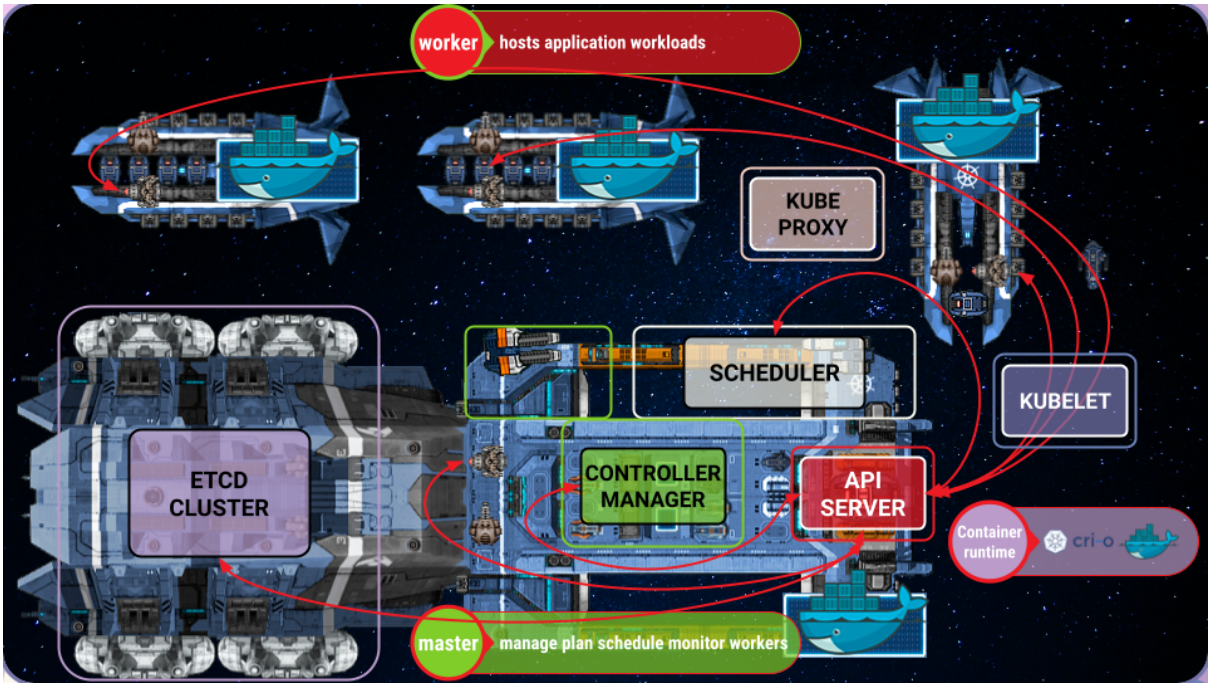
Итак, нам нужно программное обеспечение, которое может запускать контейнеры. Это среда выполнения контейнеров - `container runtime`. Популярным является Docker.

Поэтому нам нужен Docker или его поддерживаемый эквивалент, установленный на всех узлах кластера, включая и главные узлы.

Но это в том случае, если ты хочешь разместить компоненты controlplane как контейнеры.

И еще, это не обязательно должен быть Docker, более того, в скором времени Docker не будет использоваться в Kubernetes.

Kubernetes поддерживает другие движки контейнеров, такие как ContainerD или Cri-o.



Теперь давай сосредоточимся на грузовых кораблях.

Итак, у каждого корабля есть капитан.

Капитан отвечает за управление всей деятельностью на этих суднах-грузовиках.

Он несет ответственность за поддержание связи с контрольной станцией, начиная с сообщения адмиралу, что корабль заинтересован в присоединении к нашей космической компании, до получения информацию о контейнерах, которые будут загружены на корабль и отсылке отчетов о своем полете.

Он должен загружать соответствующие контейнеры на борт по мере необходимости, отправляя отчеты адмиралу о статусе самого корабля и статусе перевозимых контейнеров и подобные вещи.

Как ты понял, капитаном корабля является в Kubernetes является `kubelet`. Kubelet - это агент, который запускается на каждой ноде кластера.

Он слушает инструкции от сервера kube-api и при необходимости развертывает или уничтожает контейнеры на своем узле.

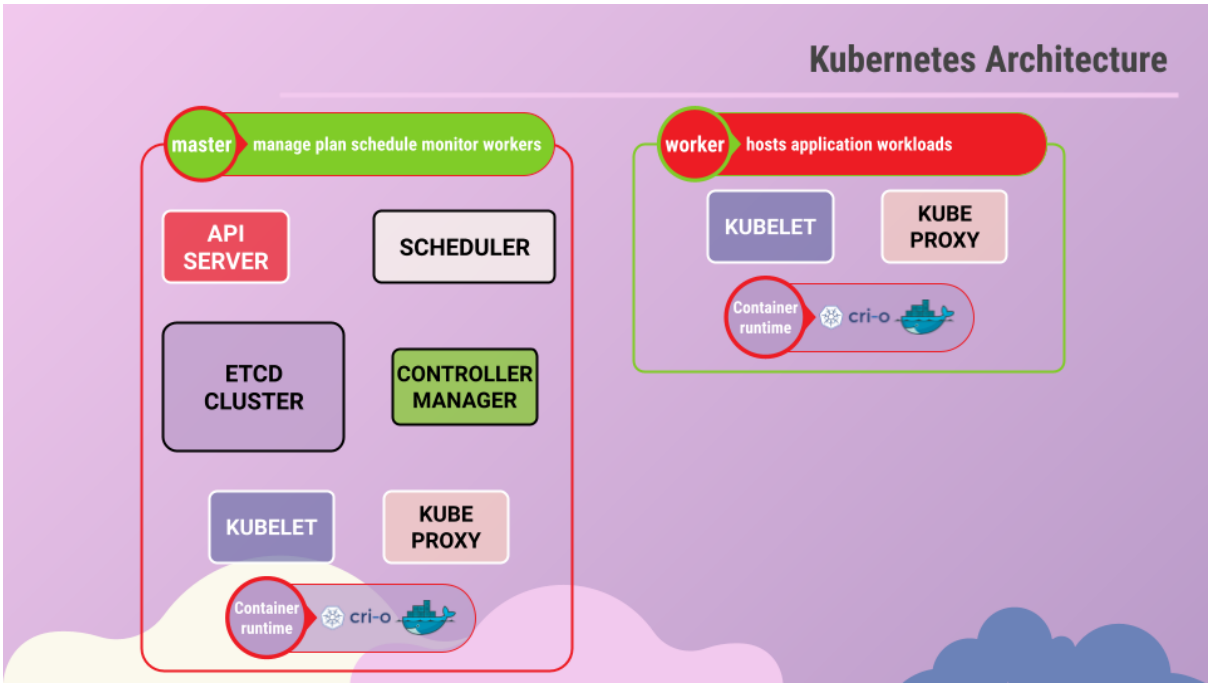
Сервер kube-api периодически получает отчеты о состоянии из kubelet, чтобы отраженная картина состояния всех нод и контейнеров на них была актуальной.

В общем, kubelet скорее капитан корабля, который управляет не кораблем, а контейнерами на корабле.

Двигаемся дальше. Приложения, работающие в контейнерах на рабочих нодах, должны иметь возможность взаимодействовать друг с другом.

Например, у нас может быть веб-сервер, работающий в одном контейнере на одном из узлов, и сервер базы данных, работающий в другом контейнере на другом узле. Как веб-сервер достигнет сервера базы данных на другой ноде?

Связь между всеми узлами кластера обеспечивается другим компонентом, известным как `kube-proxy`. Эта служба запускается на каждом узле кластера и обеспечивает репликацию на узлах таких правил, позволяющих нужным контейнерам достигать друг друга.



Итак, давай резюмируем полученную информацию:

- у нас есть главные и рабочие узлы в кластере

- у нас есть kube-apiserver, который отвечает за оркестрацию всех операций в кластере

- у нас есть кластер из ETCD-экземпляров, который хранит информацию о кластере Kubernetes

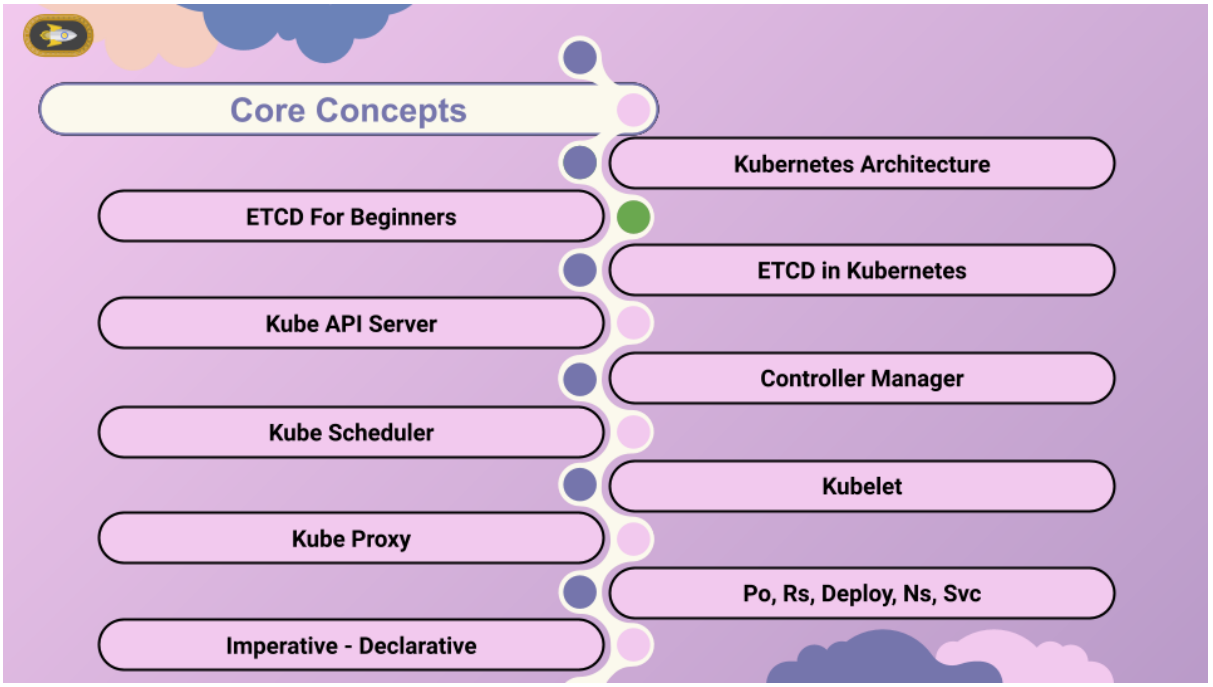
- у нас есть kube-scheduler, который отвечает за планирование приложений или контейнеров на узлы

- У нас есть разные контроллеры, которые выполняют различные функции, такие как управление нодой, репликация нагрузки и т. д.

- у нас есть kubelet, который слушает инструкции от kube-apiserver и управляет контейнерами, а также kube-proxy, который помогает в обеспечении связи между службами в кластере, ну и еще подходящий рантайм для контейнеров.

Последний пункт не такой строгий. Для ставшей уже традиционной установки кластера с помощью `kubeadm` он выполняется.
Если ставить кластер вручную, что называется `the hard way`, kubelet и kube-proxy на мастер-ноде не потребуются.

Итак, это общий обзор различных компонентов.
Мы подробно рассмотрим каждый из них в следующих лекциях.
На этом пока все, увидимся на следующей лекции.



Привет и добро пожаловать на лекцию.

Здесь мы кратко поговорим о ETCD. Это будут самые азы, и если ты знаком с этой базой данных, смело переходи к следующему видео, где мы будем обсуждать работу этого решения, как части Kubernetes.

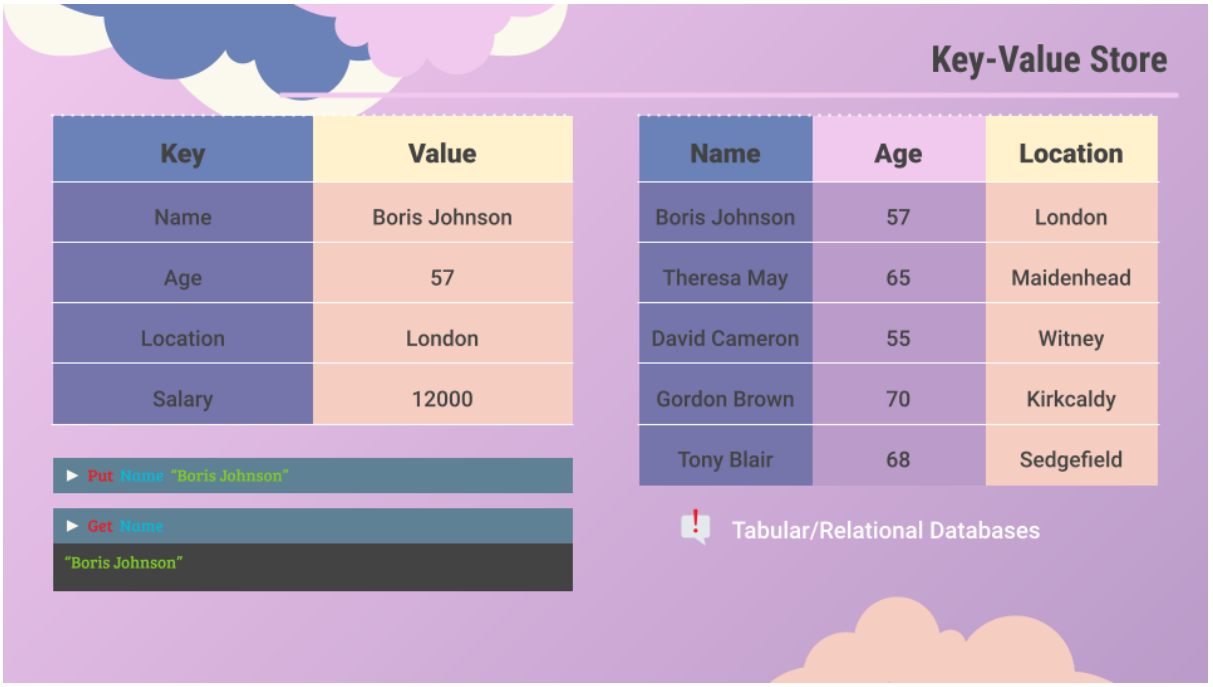
Позже в этом курсе, когда мы рассмотрим высокую доступность, мы обсудим, что значит быть распределенной системой, как ETCD работает в клиентском режиме, что такое протокол RAFT и как наши кластера могут сойти с ума из-за раздвоения личности. Также я расскажу о лучших практиках в отношении количества узлов в кластере ETCD.

Мы начнем с базового введения, а именно с того, в чем преимущества NoSQL баз данных и чем они отличаются от традиционных реляционных СУБД. Далее рассмотрим, как быстро начать работу с ETCD и попробуем поработать с ней руками в консоли.

Итак, что же такое ETCD?
Это надежное распределенное хранилище типа "ключ-значение".
Оно простое, безопасное и быстрое.

Но что такое хранилище типа "ключ-значение"?

Давай разберемся.



Традиционно базы данных проектировались для хранения данных в табличном формате.

Ты, должно быть, слышал об SQL или реляционных базах данных.
Они хранят свои данные в виде строк и столбцов.

Например, вот таблица, в которой хранится информация о нескольких людях. Строка представляет каждого человека, а столбец - тип хранимой информации.

Хранилище "ключ-значение" или NoSQL база, как их еще называют, хранит информацию в формате ключа, обратившись за которым ты сможешь получить значение, которое было помещено в этот ключ.

Чтобы записать, ты сообщаешь базе название ключа и его содержимое, оно записывается
и прямо так хранится в базе данных.

Затем ты запрашиваешь ключ, и он возвращает значение. Т.о в подобных базах данных не может быть дублирующихся ключей.

Как видишь, эти решения не используются в качестве замены обычной табличной базы данных. Они хороши для других задач, а именно для использования при хранении и извлечении небольших фрагментов данных, таких как данные конфигурации, которые требуют быстрого чтения и записи.
В этих приложениях реляционные базы сильно уступают NoSQL.

ETCD легко установить и начать работу с ней.

Просто загрузи и извлеки исполняемый бинарник, а потом запусти его.

Они разные для разной ОС, посмотри какой нужен тебе, выбери и загрузи архив из соответствующего из релиза из github, это бесплатно.

Распакуй командой и запусти исполняемый файл ./etcd.

Когда мы запускаем ETCD, она запускает службу, которая прослушивает порт 2379 по умолчанию.

У ETCD есть несколько клиентских приложений для взаимодействия с базой, через которые можно сохранять и получать хранимые данные.

По умолчанию клиент, который идет в поставкие ETCD это etcdctl.

Этот клиент является утилитой командной строки и мы можем использовать его для хранения и извлечения пар "ключ-значение" из ETCD.

Для того, чтобы сохранить пару "ключ-значение" нужно указать etcdctl команду put, далее название ключа - у меня это key1, а после значение для этого ключа, которое будет сохранено. Здесь это value1.

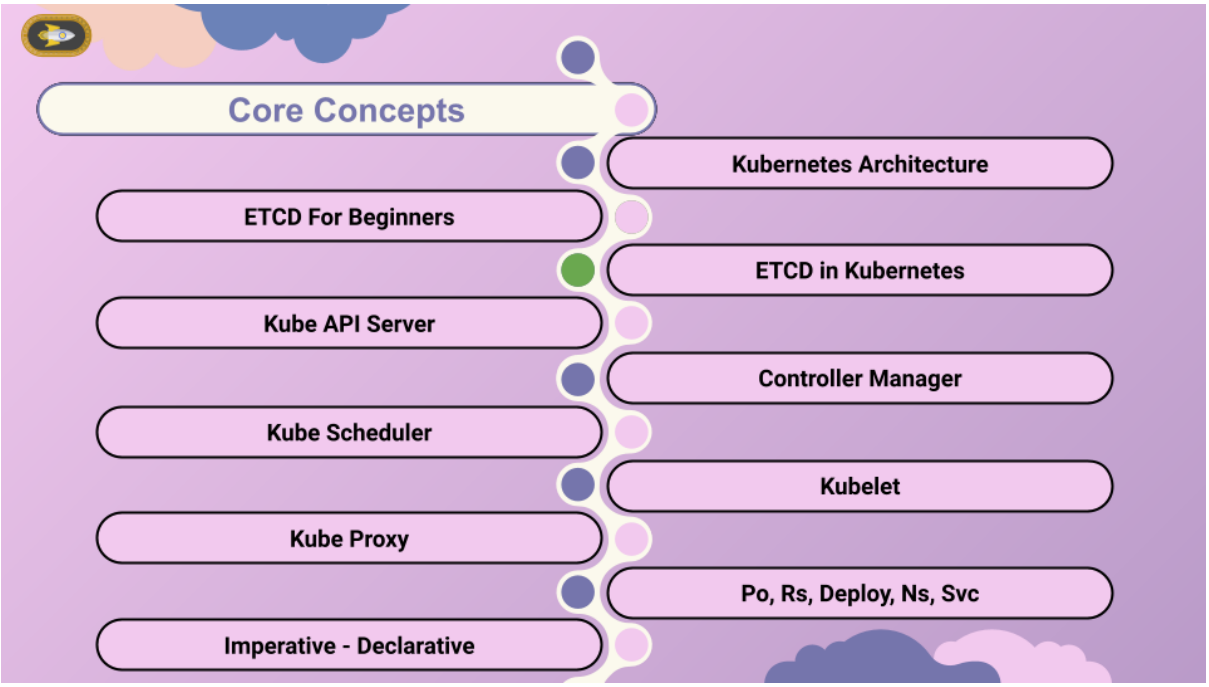
Это создаст в базе данных запись с информацией.

Для извлечения сохраненных данных, запускаем команду etcdctl get и название ключа. Больше не требуется никаких параметров.

Чтобы узнать, что еще может этот клиент запусти etcdctl без параметров и ты получишь листинг команд. На самом деле просто и быстро.

Что ж, надеюсь я создал тебе первое представление о ETCD. Это действительно популярная, мощная и продуманная штука, которая много где используется кроме Kubernetes.

Сейчас мы сделали краткий обзор, далее посмотрим, как ETCD себя чувствует в Kubernetes, а еще далее в курсе мы вернемся к ней, когда будем разбирать ее конфигурирование в режиме высокой доступности и лучшие практики, связанные с этим.
Жду тебя на следующей лекции!



Привет и добро пожаловать на эту лекцию.

В предыдущей лекции мы обсудили основы ETCD, сейчас мы поговорим о роли ETCD в Kubernetes.

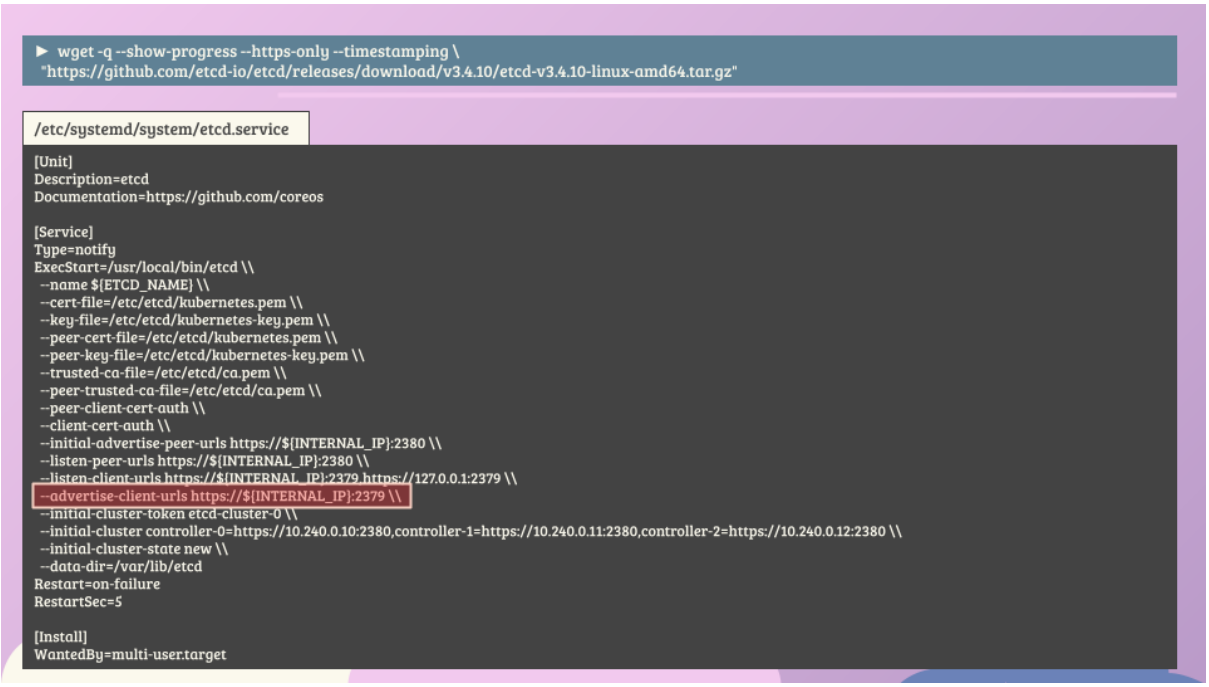
База данных ETCD хранит информацию о кластере. Эти данные описывают состояние нод, PODs, конфигураций, чувствительных данных, а также учетные записи, роли, привязки и другие.

Вся информация, которую ты видишь при запуске команды kubectl get, поступает с сервера ETCD. Каждое изменение, которое ты вносишь в свой кластер, например добавление дополнительных нод, развертывание PODs или ReplicaSets, обновляется на сервере ETCD. Изменение считается завершенным только после фиксации этого обновленного состояния на сервере ETCD.

В зависимости от того, как мы настраиваем свой кластер, ETCD развертывается по-разному.

В этом разделе мы обсуждаем два типа развертывания Kubernetes. Первый - развернуть все с нуля, другой - с помощью утилиты kubeadm.

Позже в этом курсе, мы настроим учебный кластер с помощью kubeadm, и также я покажу, как сделать это с самого низа руками. Для экзамена достаточно уметь через kubeadm, но я советовал бы сделать это с нуля, т.к. полезно самому прочувствовать разницу между двумя методами.



Если ты настраиваешь свой кластер с нуля, то ETCD развертывается через загрузку бинарников, это нужно сделать самому, устанавливая исполняемые файлы и настраивая ETCD как службу на своей мастер-ноде.

В эту службу передается множество опций, здесь основные, их может быть больше.

Как видишь, некоторые из опций относятся к сертификатам. Многих сертификаты пугают, но не волнуйся, мы узнаем больше об этих сертификатах, как их создавать и как их настраивать. Это будет позже в этом курсе в разделе `security`, там у нас есть целый раздел о сертификатах TLS.

Остальные параметры посвящены настройке ETCD как кластера. Мы рассмотрим эти возможности, когда будем разбирать высокую доступность в Kubernetes, но на один вариант все же стоит обратить внимание.

Это рекламируемый URL клиента - `--advertise-client-urls`.

Это адрес, по которому слушает ETCD. Он находится на IP-адресе сервера на порту 2379, который является портом по умолчанию, который занимает ETCD.

Этот URL-адрес, нам нужно сообщить kube-apiserver, чтобы он смог общаться с сервером ETCD.

Если ты настраиваешь свой кластер с помощью kubeadm, то kubeadm развертывает для тебя сервер ETCD в виде POD в пространстве имен kube-system.

Ты можешь изучить данные из ETCD с помощью утилиты etcdctl, запустив ее в POD базы данных. Чтобы вывести список всех ключей, хранящихся в Kubernetes, выполни команду etcdctl get следующим образом.

Kubernetes хранит данные в определенной каталожной структуре, где корневой каталог - это `registry`, в котором есть различные Kubernetes-примитивы, такие как nodes, pods, replicasets, deployments, roles, secrets и т. д.

В среде высокой доступности у нас будет несколько мастер нод в кластере.
Таким образом, у нас появится несколько экземпляров ETCD, распределенных по главным узлам.

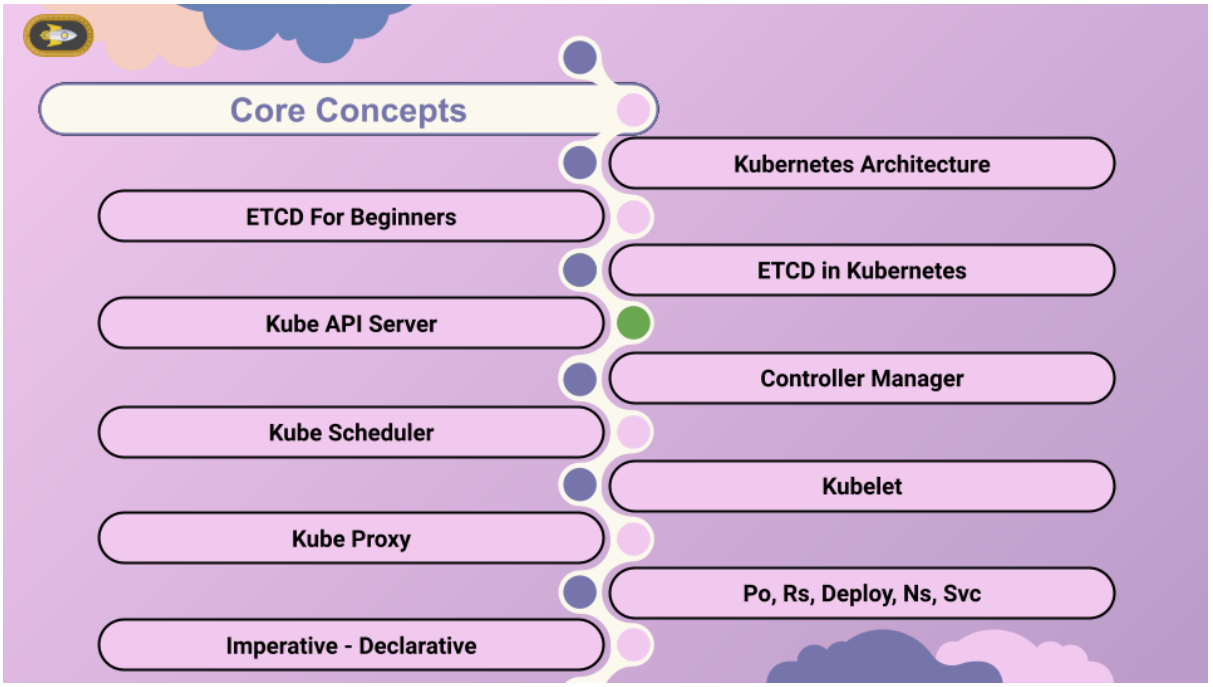
В этом случае не забудь сообщить всем ETCD-инстансам, что они не одни.

Чтобы экземпляры ETCD знали друг о друге, установи правильный параметр в конфигурации служб ETCD. Параметр `--initial-cluster` - это адреса тех мест, где будут жить члены ETCD-кластера.

Мы поговорим о высокой доступности позже, но когда мы туда доберемся, ты скорее всего уже столкнешься с

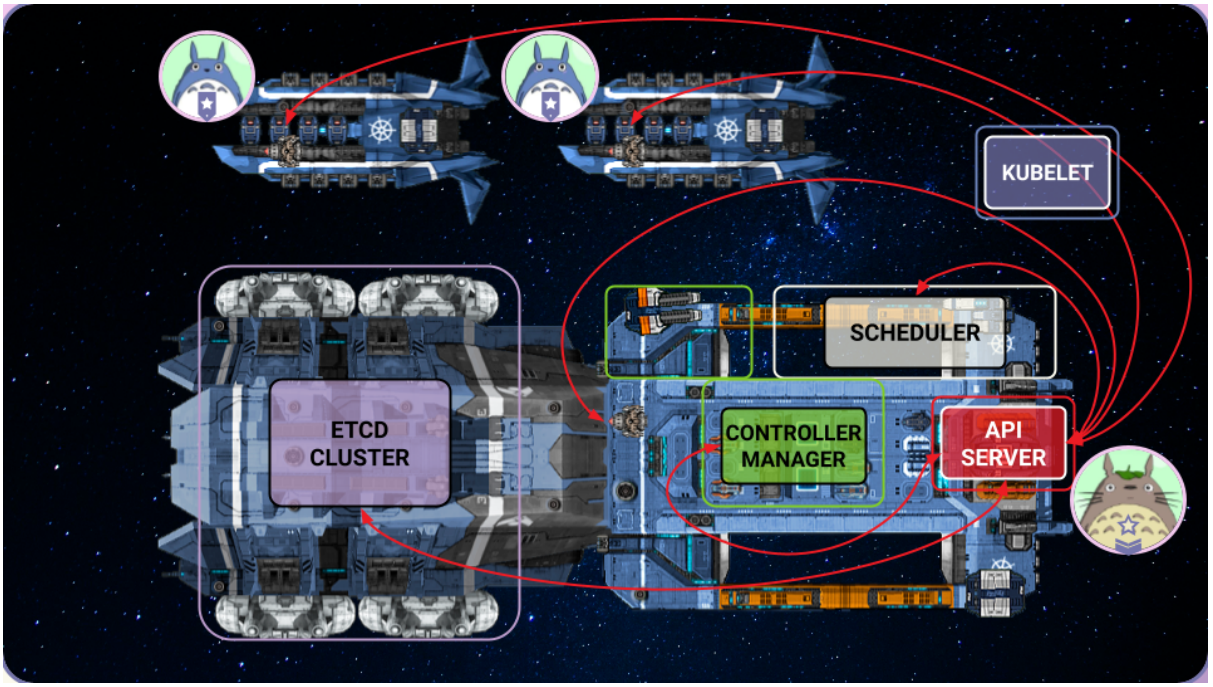
конфигурационными файлами компонентов Kubernetes.
И они уже будут немного привычными и в это проще будет вникнуть.

Ну вот и все в этой лекции, увидимся на следующей.



Привет и добро пожаловать на лекцию. В этой лекции мы поговорим о kube-apiserver в Kubernetes.

Ранее мы обсуждали, что API-сервер Kubernetes - это основной компонент управления в кластере.



Как ты помнишь, все компоненты координируют свое поведение через kube-api и только он может писать в ETCD. Таким образом этот сервер главное связующее звено системы.

С ним общаются как компоненты controlpane, так и агенты на рабочих нодах, постоянно обновляя информацию о себе в общем хранилище.

Раз этот компонент такой важный, не станет ли он быть единой точкой отказа? Нет Kubernetes может работать в режиме высокой доступности одновременно с несколькими API-серверами, мы поговорим об этом позже.

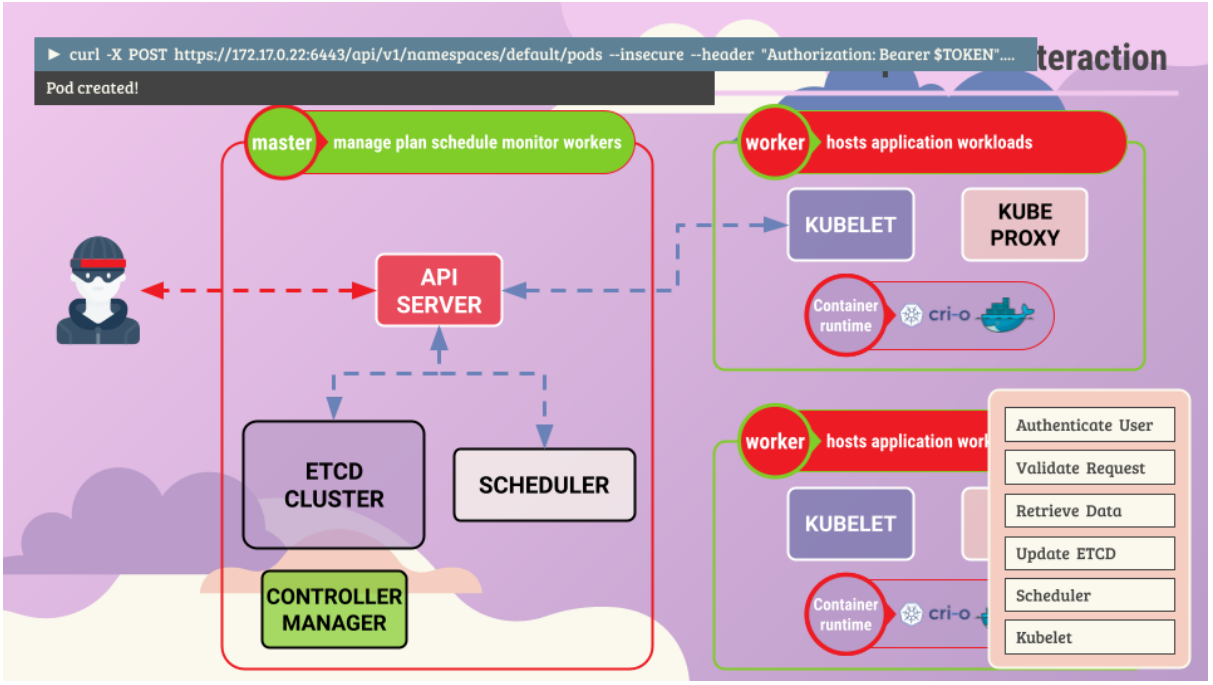
Когда ты запускаешь команду в kubectl, эта утилита фактически напрямую обращается в kube-apiserver.

Kube-api сначала аутентифицирует и валидирует твой запрос. Затем он извлекает данные из ETCD-кластера и посылает тебе в ответ эту информацию.

Итак, запишем для себя, аутентификация, проверка запроса, извлечение данных.

На самом деле тебе не нужно использовать утилиту командной строки kubectl. Вместо этого ты также можешь вызывать API напрямую, отправив вот такой POST-запрос.

Давай рассмотрим пример создания POD таким манером.



Как и в первый раз, наш запрос сначала аутентифицируется, а затем проверяется.

Следующим шагом сервер API создает объект POD, не назначая его никакой ноде, а лишь записывает информацию на сервер ETCD, что в системе есть POD без ноды. А далее он уведомляет пользователя о том, что POD был создан.

Scheduler непрерывно контролирует сервер API и понимает, что существует новый POD без узла. Планировщик определяет правильный узел для размещения этого

нового POD и передает свое решение, о том какую ноду он выбрал, обратно в kube-apiserver.

Затем API-сервер обновляет информацию в кластере ETCD. Следующий шаг - сервер API передает информацию с данными POD в kubelet на соответствующей рабочей ноде.
С этой информацией kubelet создает POD у себя на ноде и инструктирует движок выполнения контейнеров развернуть нужный образ приложения.

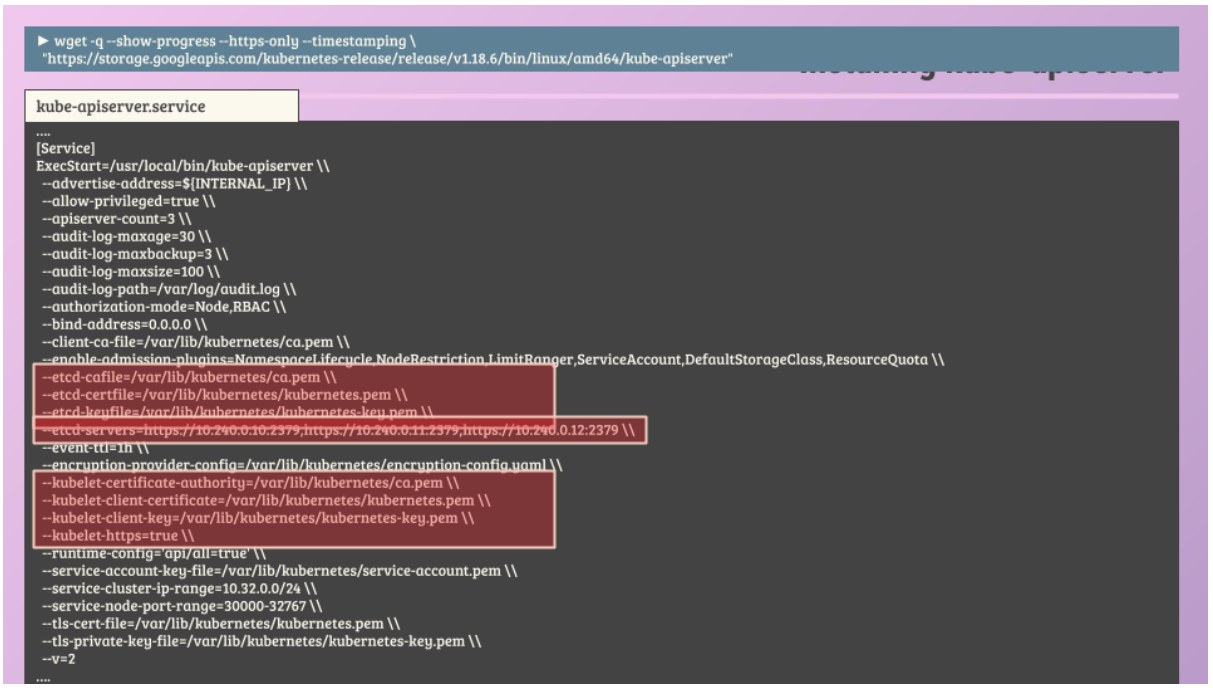
После этого kubelet отправляет результат обратно в kube-apiserver, а тот в свою очередь, обновляет данные о состоянии POD в кластере ETCD.

И подобный шаблон используется каждый раз, когда запрашивается изменение.

Kube-apiserver всегда учавствует во всех задачах кластера, он находится в центре всех событий, ведь от него зависит, что изменения в кластере будут запомнены и утверждены в базе.
Т.о. без API-сервера невозможно произвести изменения на уровне кластера.

Подведем итог: сервер kube-api отвечает за аутентификацию и проверку запросов, извлечение и обновление данных в хранилище данных ETCD.

Фактически, этот сервер API является единственным компонентом, напрямую взаимодействующим с хранилищем данных ETCD.
А другие компоненты, такие как scheduler, kube-controller-manager и kubelet, используют API-сервер для выполнения обновлений в кластере, в части областей, за которые они сами несут ответственность.



Если ты готовил свой кластер с помощью инструмента kubeadm, тебе это не понадобится, но если мы пробуем `the hard way`, то kube-apiserver доступен для скачивания в виде запакованого бинарника на github-странице релизов Kubernetes. Загрузи его и настрой для работы в качестве службы на своем мастер-узле.

Как ты видишь здесь kube-apiserver запускается с множеством параметров.
В этом разделе курса мы просто смотрим, как устанавливать и настраивать эти отдельные компоненты архитектуры Kubernetes.

Тебе не обязательно глубоко вникать в это прямо сейчас и понимать все варианты настройки. Но я думаю, что знакомство на высоком уровне с некоторыми из этих вещей, упростит задачу позже, когда мы будем настраивать весь кластер и его компоненты с нуля. У тебя будут припоминаться какие-то уже знакомые вещи и это сильно помогает.

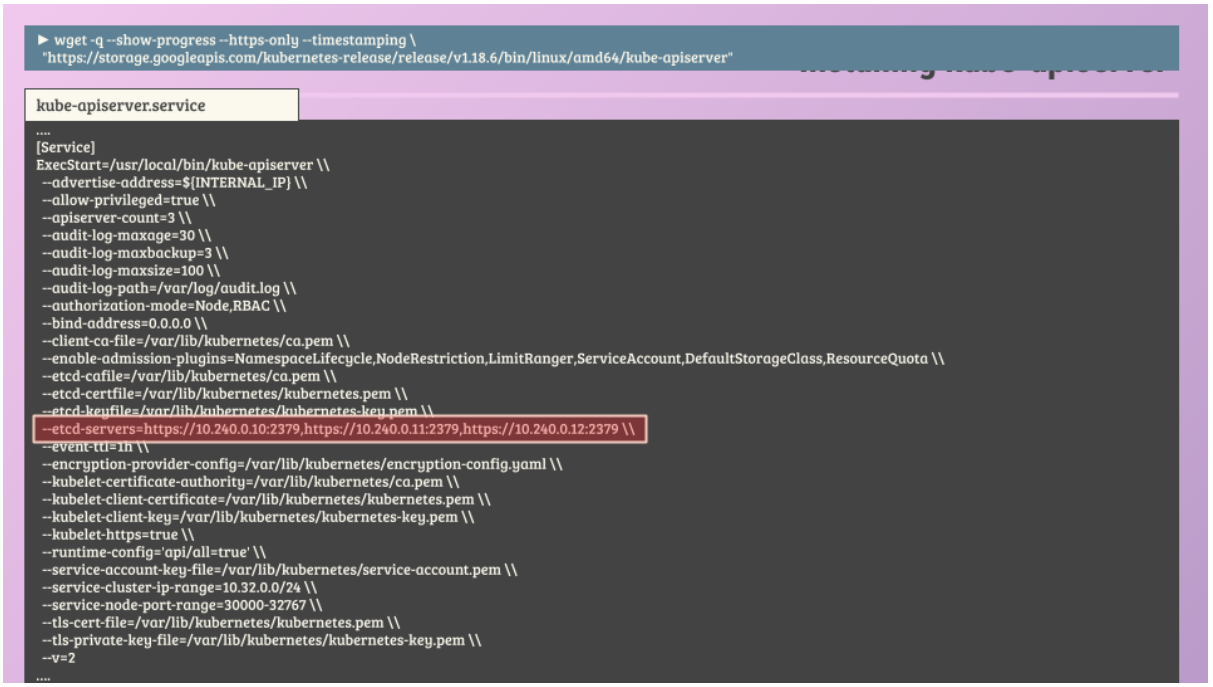
Архитектура Kubernetes состоит из множества различных компонентов, работающих друг с другом, разговаривающих друг с другом по-разному, поэтому всем им нужно знать, где находятся другие компоненты. Существуют разные режимы аутентификации, авторизации, шифрования и безопасности. Вот почему у нас так много вариантов настройки, и когда мы пройдемся по соответствующему разделу курса, мы снова откроем этот файл и будем разбирать его опции с новых позиций понимания всей системы.

А пока мы рассмотрим несколько важных.

Многие из них - это указания на сертификаты. Эти сертификаты используются для защиты связи между различными компонентами.
С сертификатами почти у всех проблемы и мы рассмотрим их более подробно, когда попадем в раздел security. Там присутствует целая серия лекций о SSL/TLS, это будет позже в этом курсе.

Так что нам не надо много об этом думать.

Просто помни, что эти разные компоненты системы изначально друг другу не доверяют. Поэтому все компоненты, которые мы рассмотрим в этом разделе, будут иметь связанные с собой сертификаты.



Опция `--etcd-servers` - это те места, где проживают наши экземпляры ETCD-серверов. Благодаря этим параметрам сервер kube-api знает куда подключаться к серверам etcd.

Ок, а как нам посмотреть параметры kube-api сервера в уже существующем кластере?

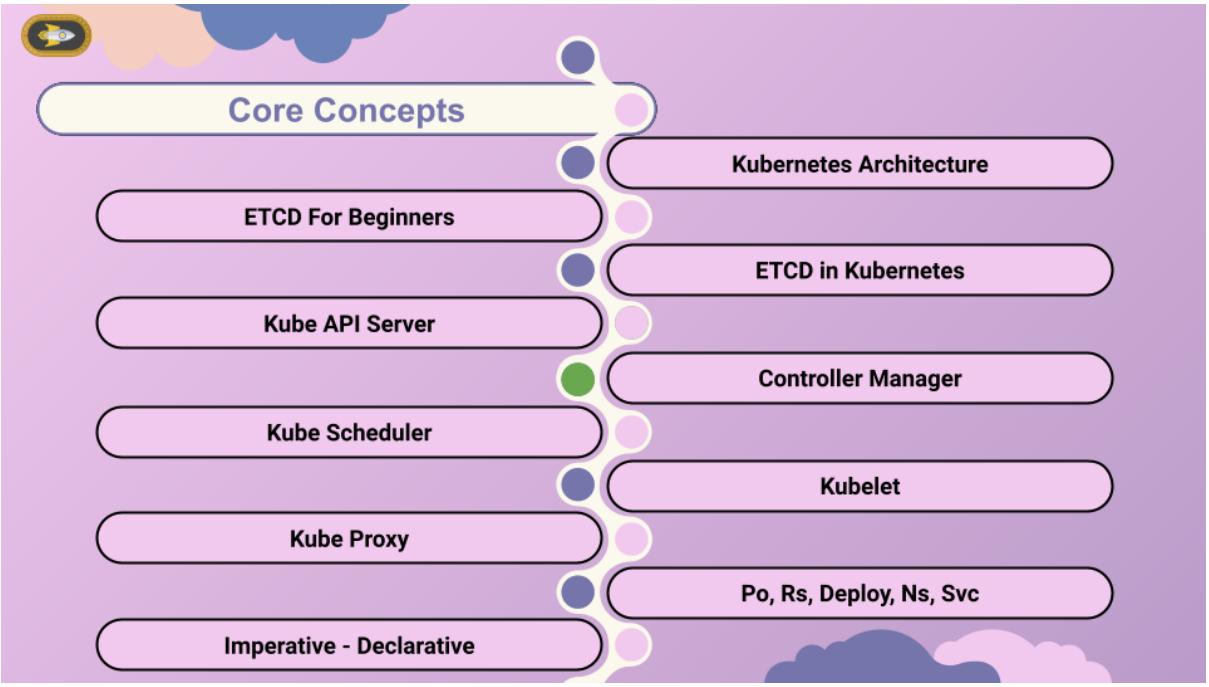
Это зависит от того, как мы провижинили свой кластер.

Если мы создали его с помощью инструмента kubeadm, то kube-apiserver будет запущен как POD в пространстве имен `kube-system` на мастер-ноде.

Ты можешь посмотреть опции запуска в файле определения POD, расположенном в размещении `/etc/kubernetes/manifestests`.
Это папка манифестов Kubernetes.

В случае ручного развертывания у тебя будет служба kube-apiserver, расположенная по адресу `/etc/systemd/system/kube-apiserver.service`.
Ты также можешь увидеть выполняющийся процесс и те опции, с которыми он запущен, вызвав листинг процессов и отфильтровав по запросу `kube-apiserver`.

Ну вот и все в этой лекции, увидимся на следующей!



Привет и добро пожаловать на лекцию. В ней мы будем говорить о kube-controller-manager.

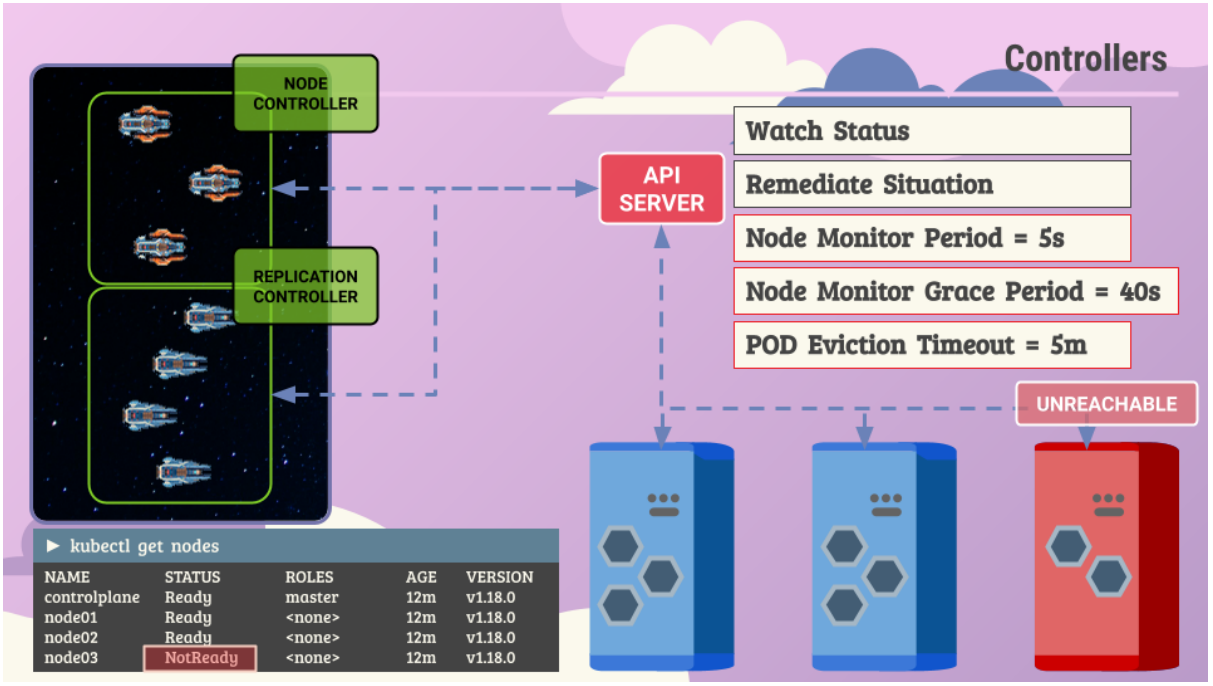
Как мы обсуждали ранее, kube-controller-manager управляет различными контроллерами в Kubernetes.
Контроллер похож на портовую команду или группу небольших кораблей, у которых есть собственный набор обязанностей.

Например такие, как эта команда на специальных маленьких звездолетах, которые производят сканирование и внешний осмотр грузовых кораблей и принимают некоторые решения, если видят по этой части отклонения.

Они на страже каждый раз, когда прибывает новый грузовой корабль, или когда корабль покидает пределы базы или вдруг он разрушиться прямо около базы за дело возьмется бригада, которая специально обучена реакции на такие ситуации.

Также есть другие виды этих маленьких корабликов, маленькие буксиры, который помогают капитану корабля. Они заботятся о контейнерах которые повреждены, а также помогут, если вдруг корабль слишком сильно нагрузили.

Итак, эти звенья кораблей с оранжевыми крыльями постоянно следят за состоянием контейнеров на грузовиках, а также принимают необходимые меры для исправления ситуации.



В терминах Kubernetes `controller` - это процесс, который непрерывно отслеживает состояние различных компонентов в системе и работает над приведением всей системы в желаемое функциональное состояние.

Например, `node-controller` отвечает за мониторинг состояния нод и выполнение необходимых действий для поддержания работы приложения.

Все это осуществляется через сервер kube-api. Node-controller проверяет состояние нод кластера каждые 5 секунд.

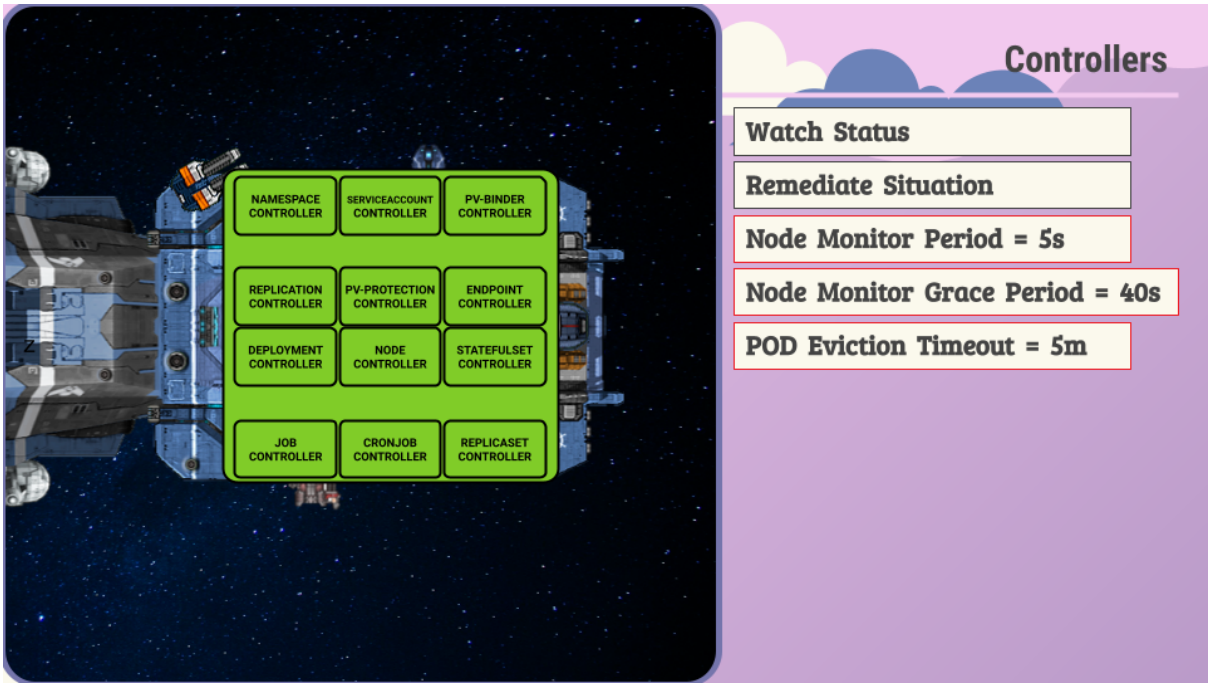
Таким образом, node-controller может отслеживать состояние ноды, если он перестает получать контрольные сообщения (`heartbeat`) от узла, нода помечается как недоступная.

Далее контроллер ждет 40 секунд, прежде чем пометить ноду как недоступную (`unreachable`). Уже после того, как он пометил ноду как недоступную, он дает пять минут, чтобы нода вернулась в строй.
Если этого не случилось, он удаляет PODs, назначенные на эту ноду, и отправляет их на развертывание в работоспособные рабочие узлы кластера.

Если PODs являются частью replicaset, то за этими подами будет приглядывать другая группа корабликов сопровождения, а именно контроллер репликации.

Он отвечает за мониторинг состояния replicasets и обеспечение того, чтобы желаемое количество PODs всегда было доступно в пределах этого replicaset.

Если POD умирает, контроллер создает другой.



Это были всего лишь два примера контроллеров. В Kubernetes доступно гораздо больше таких контроллеров.

Какие бы концепции мы ни видели в Kubernetes, такие как deployments, services, namespaces, persistent volumes, как и все другие интеллектуальные способности, которыми обладают эти примитивы Kubernetes, они реализуется через эти различные контроллеры.

Как ты понимаешь, это своего рода мозг, стоящий за многими вещами в Kubernetes. Теперь, ты представляешь что это за контроллеры, но где они расположены в нашем кластере?
Все они объединены в единый процесс, известный как `kube-controller-manager`.

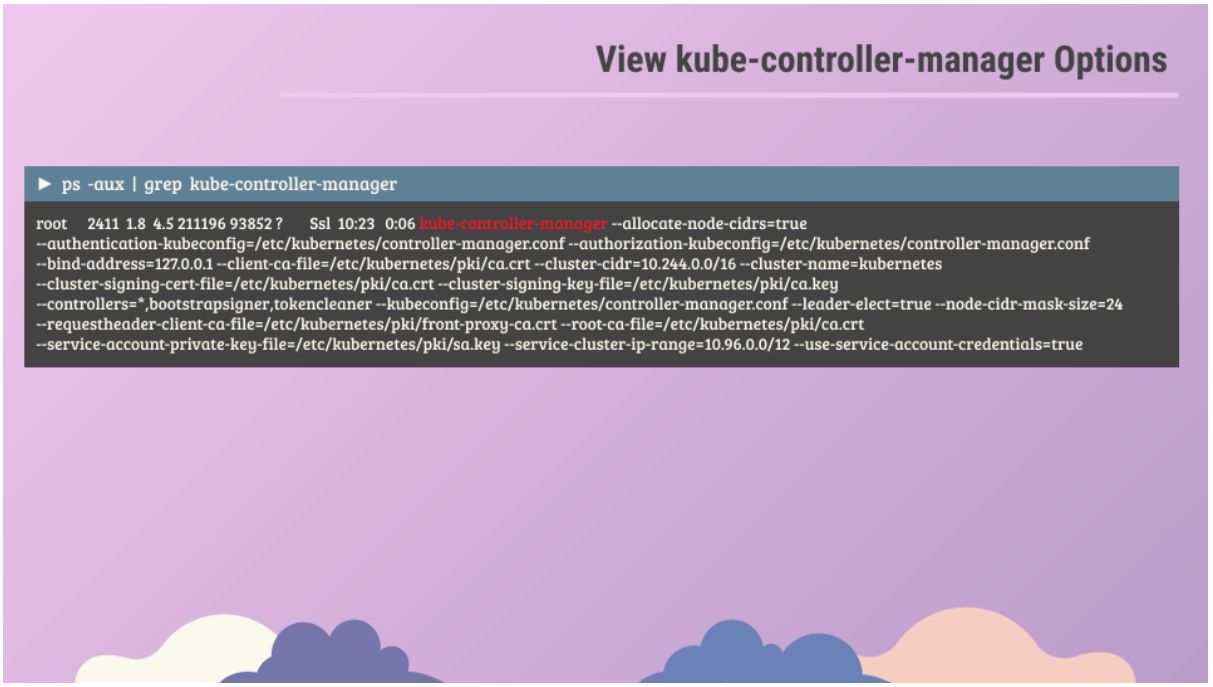
Когда ты устанавливаешь его в систему, также устанавливаются и различные контроллеры.

Итак, как установить и просмотреть kube-controller-manager в Kubernetes?

Как и в случае с предыдущим компонентом, загрузи kube-controller-manager со страницы релизов Kubernetes. Потом извлеки и запусти как службу.

Когда мы запускаем его, как видишь, у него есть список параметров. Это то место, где ты можешь дополнительно настроить контроллеры под свои задачи и особенности этого отдельного кластера.

Как ты наверное заметил, у нас тут есть несколько дефолтных параметров, о которых мы говорили несколько слайдов назад - это `--node-monitor-period`, `--node-monitor-grace-period` и `--pod-eviction-timeout`.
Здесь они у нас проходят как опции запуска.



Существует также дополнительная опция, называется `--controllers`, с помощью которой мы можем указать, какие контроллеры будут включены.

По умолчанию все контроллеры включены, но в некоторых случаях требуется их отключить и у нас есть такая возможность.

Имей в виду, если в кластере какой-либо из контроллеров не работает или не существует, то будет хорошей отправной точкой для начала траблешутинга.

Итак, как нам просмотреть параметры, с которыми выполняется kube-controller-manager?

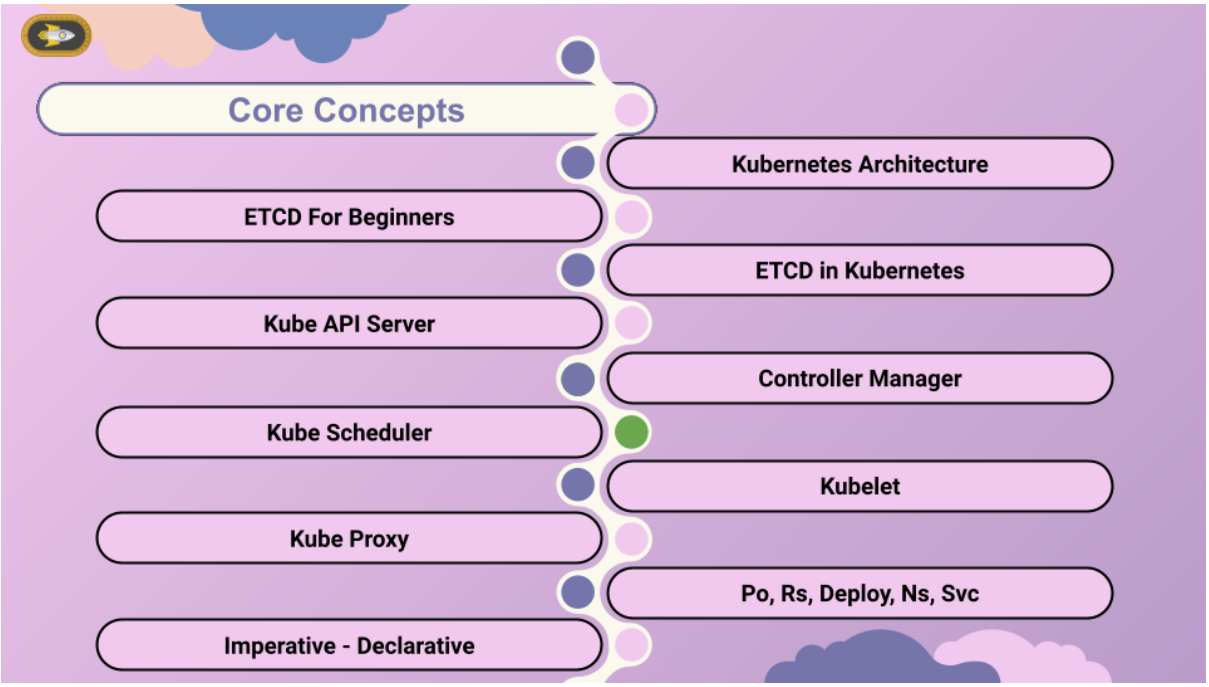
Опять же, это сильно зависит от того, как мы создавали свой кластер.

Если мы провижинили его с помощью инструмента kubeadm, то kube-controller-manager будет развернут как POD в namespace kube-system на мастер-ноде.
Ты можешь увидеть эти опции в файле определения POD, который расположен в папке /etc/kubernetes/manifestests.

В сетапе, отличном от kubeadm, ищи параметры в конфигурационном файле службы kube-controller-manager, который лежит в каталоге с другими службами.

Ты также можешь увидеть эффективные параметры исполняющегося процесса, сделав листинг процессов на мастере и сделав поиск по запросу kube-controller-manager.

Это все в этой лекции, увидимся в следующей.



Привет и добро пожаловать на эту лекцию. В ней мы поговорим о kube-scheduler.

Ранее мы уже обсуждали, что scheduler отвечает за назначение PODs на ноды.

Но все-таки у планировщика в Kubernetes есть отличия от нашего космического крана, а именно в том, что scheduler фактически не отправляет POD. Он лишь решает, какая нода наиболее подходит для этого POD в данный момент. В действительности POD на ноде создает kublet. Kubelet - это капитан транспортного корабля и его задача заниматься созданием POD, назначенного на его ноду. А вот решение принимает scheduler. Давай взглянем поближе, как он это делает.

А зачем вообще планировщик, ведь kubelet умеет запускать контейнеры, может быть отдать все им?
Когда у тебя много кораблей и много разных контейнеров, нужно быть опытным логистом, чтобы разместить их по правильным кораблям, а капитаны кораблей для этого не очень подходят.

Например у нас есть на станции несколько кораблей, два из них могут везти небольшие контейнеры, а третий берет на борт лишь один, но огромный.

Наш крановщик в кране-планировшике должен быть уверен, что корабли после загрузки не только смогут взлететь, но и прибыть в место назначения. Для этого он должен быть осведомлен как о характеристиках контейнера, так и о типах кораблей, имеющихся в распоряжении, их текущей загрузке, а также состояния торговых путей. Так, например, зачем грузить контейнер с маркировкой `на Сатурн` на местные корабли, если скоро туда полетит большой грузовой корабль.

Как видишь это большая и сложная задача. Поэтому ее решает отдельный компонент. Планировщик решает какие ноды возьмут PODs по различным критериям. Это могут быть вычислительные ресурсы, необходимые приложению, это может быть особенность отдельной ноды, которая выделена под такой тип нагрузки, другие параметры, но все они оцениваются и играют роль в конечном решении.

Как scheduler назначает PODs?
Он проверяет данный ему POD и пытается найти лучшую для него ноду. Например, вот эти три PODs.
У каждого есть набор требований по CPU и памяти.

Желтый самый жирный, ему необходимой 10 единиц CPU.

Наш scheduler пройдет два этапа, чтобы подыскать дом этому POD.
Сначала он отфильтрует подходящие ноды из доступных, на которых это приложение сможет поместиться.
Как ты видишь, на этом этапе две первые ноды выбывают, т.к. на них физически нет требуемых для этой рабочей нагрузки 10 CPU.
Т.е. планировщик отфильтровал два маленьких узла и больше они в соревновании не участвуют, т.к. у них не хватило ресурсов для будущего POD.

Теперь наступает второй этап, этап ранжирования узлов. Каждый узел будет оценен по нескольким направлениям, и который наберет наибольшее количество очков, тому и размещать у себя POD.

Но как именно планировщик выберет один из двух, как он оценит ранги нод, чтобы выбрать лучшую ноду?
Он воспользуется шкалой от 0 до 10 и выставит оценки в нескольких номинациях.

Например, scheduler вычислит количество ресурсов, которые будут свободны на узлах после размещения на них POD.

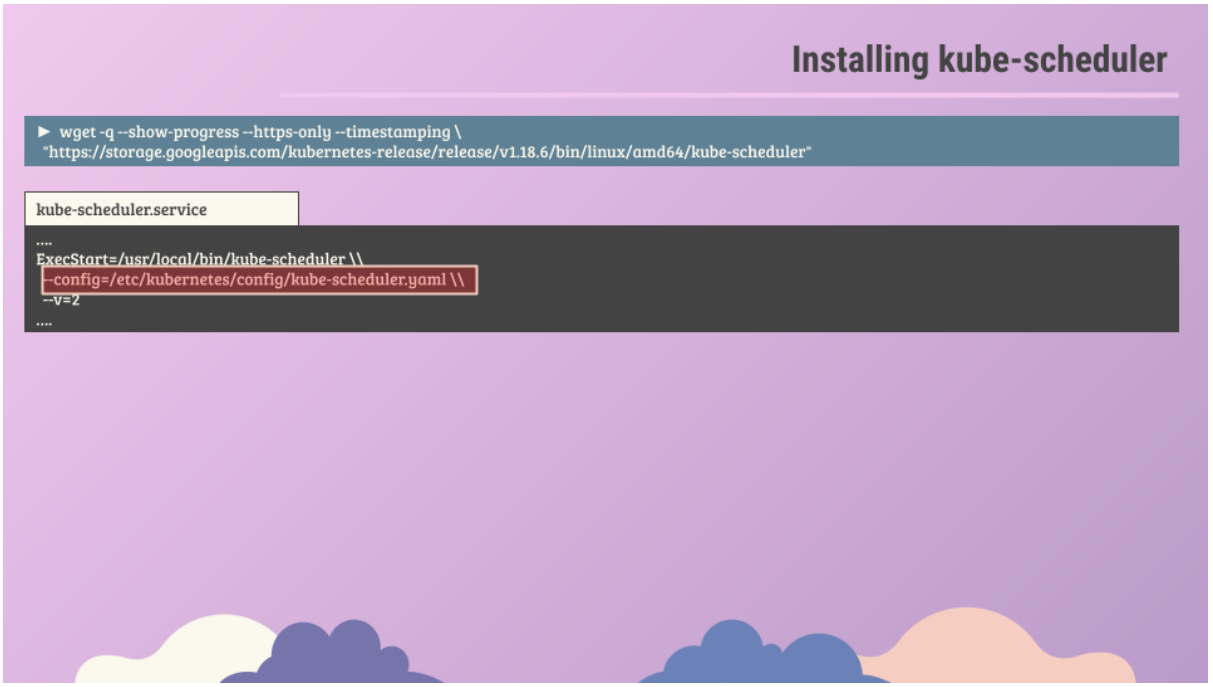
В этом случае у того, что справа станет 6 свободных CPU, если на него будет назначен POD, что на 4 больше, чем у другого. Поэтому рейтинг правой ноды будет выше. Также sheduler будет рассуждать и по другим номинациям. В этом примере мы выбрали только один параметр - центральный процессор, поэтому побеждает самая толстая нода.

Вот как работает планировщик на высоком уровне.

И, конечно же, все планировщики можно настроить, а еще ты можешь написать и подключить свой собственный планировщик.

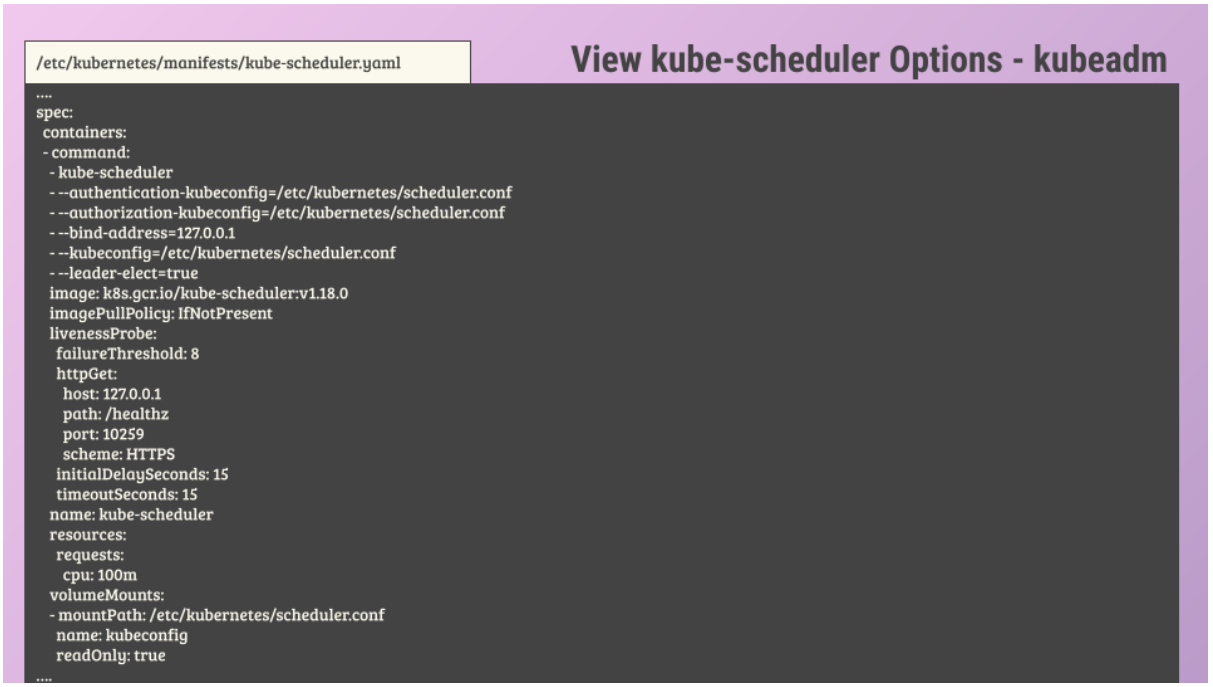
Алгоритм назначения сложный, в нем много критериев: требования к ресурсам, лимиты, taints and tolerations, node selectors, правила affinity и т.д.

Вот почему у нас есть целый раздел, посвященный планированию нод, который будет в этом курсе уже очень скоро, и где мы погрузимся в каждую из опций и разберем гораздо подробнее.
В этой лекции мы лишь обсуждаем сам процесс на высоком уровне.



Итак, как тебе поставить kube-scheduler?
Скачай его из релизов Kubernetes с github, распакуй и установи как службу.
В строке запуска службы укажи путь к файлу конфигурации планировщика.

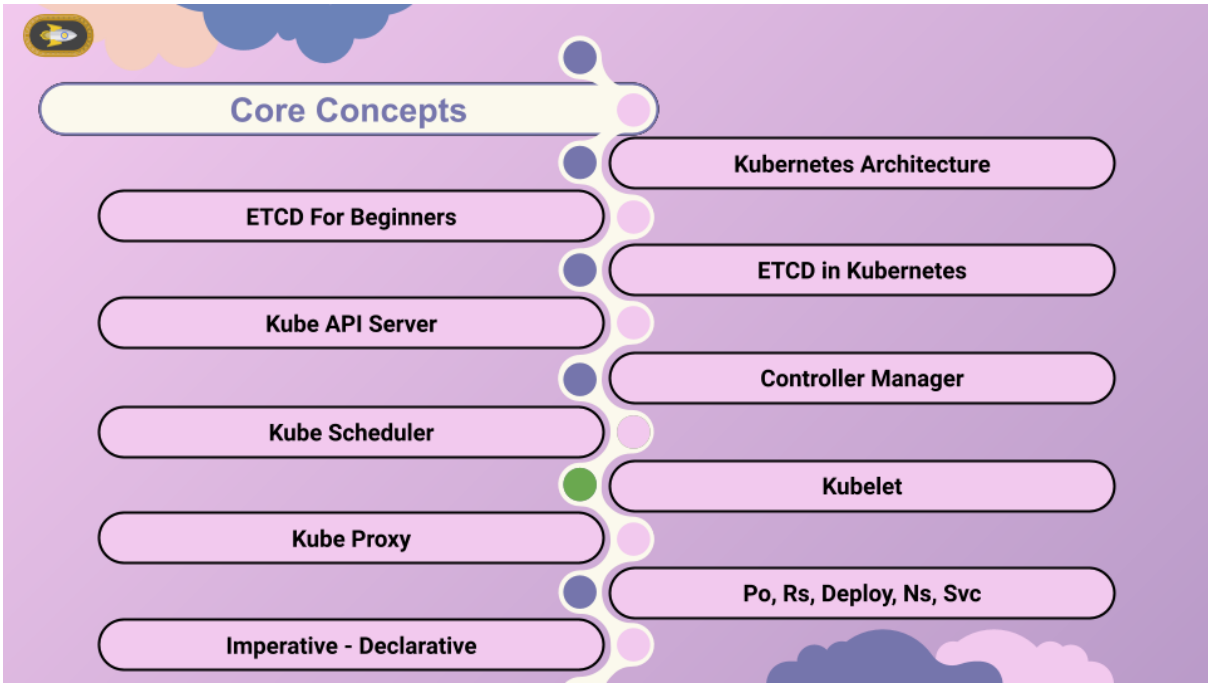
А как узнать, с каким параметрами kube-scheduler запущен в данный момент?



Если ты настроил кластер при помощью инструмента kubeadm, то kubeadm развернет kube-scheduler на мастере в качестве POD в системном namespace, как ты знаешь это пространство имен `kube-system`. Ты можешь увидеть параметры в файле определения POD, расположенном в папке /etc/kubernetes/manifestests.

Также можно просмотреть выполняющийся процесс и его эффективные параметры, сделав `ps` мастер-ноде и отфильтровав по слову `kube-scheduler`.

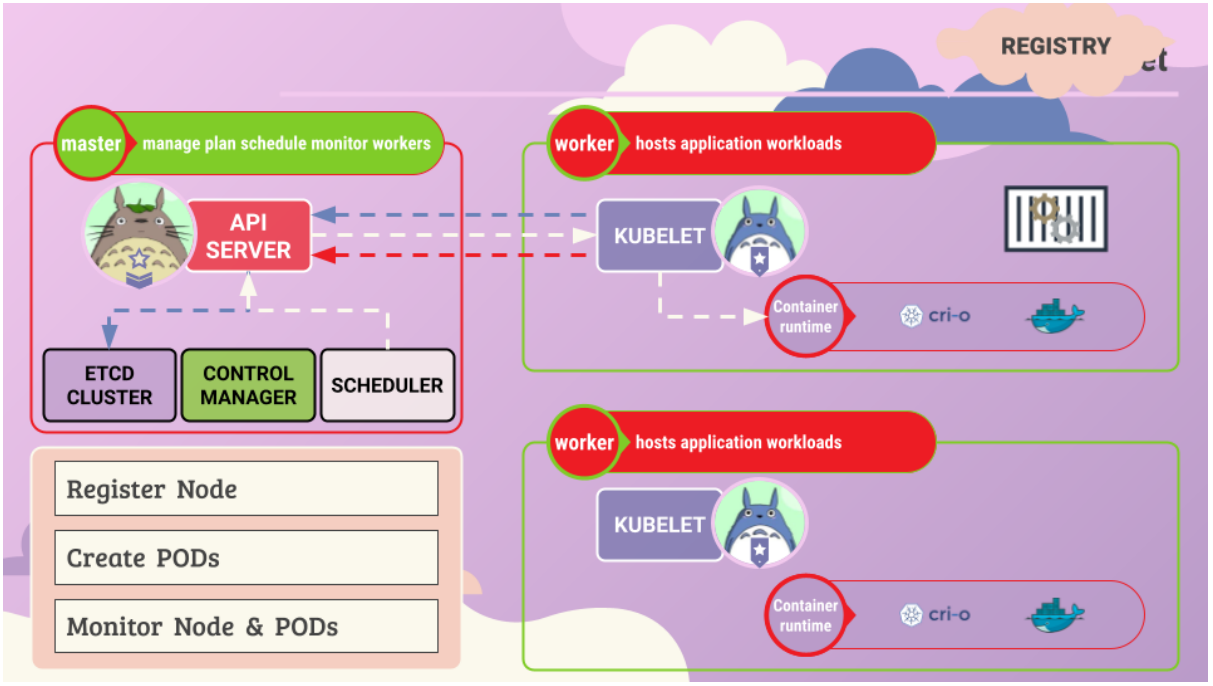
Ну вот и все для этой лекции, увидимся на следующей.



Привет и добро пожаловать на лекцию о компоненте `kubelet`.

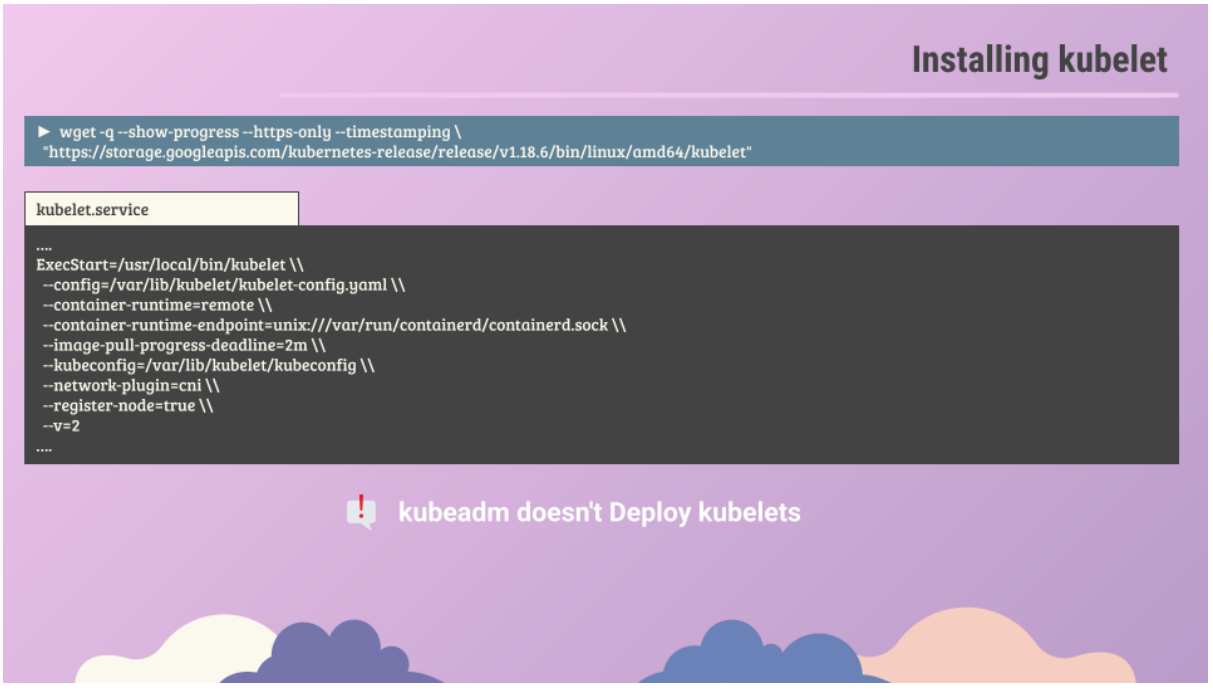
Ранее мы уже говорили, что kubelet можно сравнить с капитаном космического корабля.

Капитан является ответственным за свой корабль-ноду и управляет всеми активностями на ней. Он решает вопросы с работой контейнеров на своем узле. Также на нем лежит работа по подготовке ноды к вступлению в кластер. Он общается с контрольной станцией, получая через нее инструкции от планировщика, и действует сообразно им, если требуется загрузить или выгрузить контейнер. Также он регулярно отправляет отчеты о состоянии корабля и контейнерах на его борту.



Ок, еще раз о важных функциях kubelet:

- kubelet, расположенный на рабочей ноде, регистрирует свою ноду в кластере
- получая инструкцию по запуску POD с контейнером, он связывается с средой выполнения контейнеров, например Docker, и дает команду, чтобы тот скачал нужный образ и запустил его экземпляр с нужными параметрами.
- kubelet постоянно мониторит состояние POD и его контейнеров, и периодически посылает эту информация в kube-apiserver.



Как нам поставить kubelet в систему?

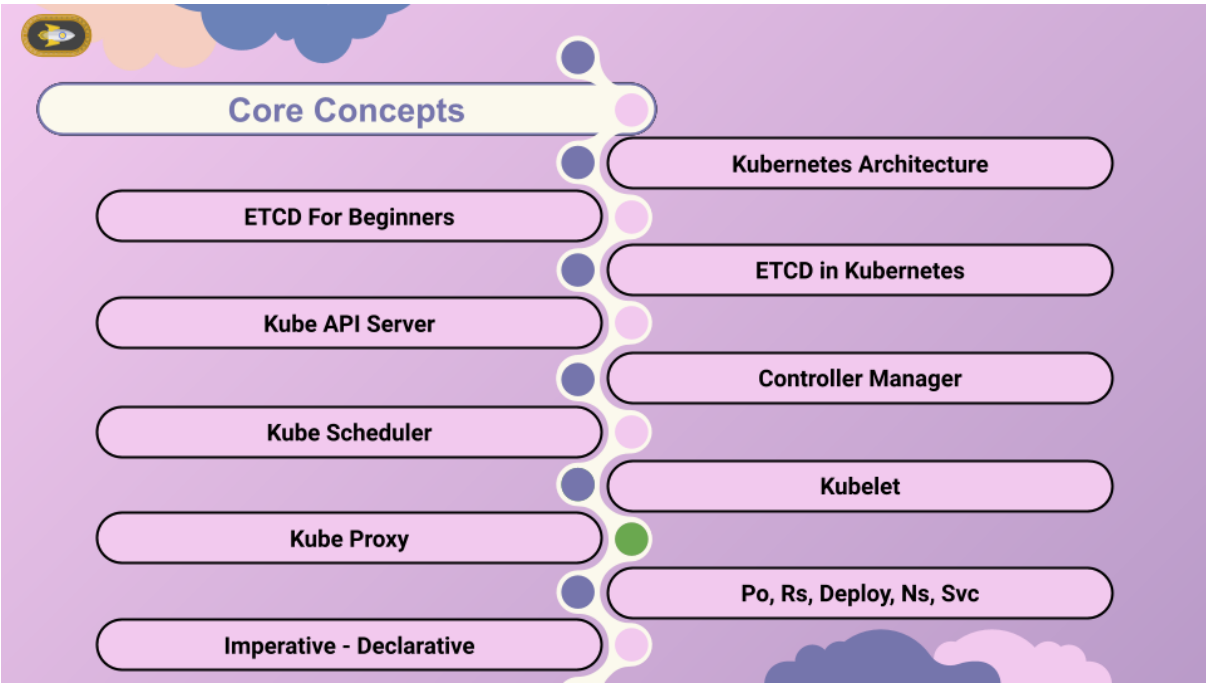
Здесь есть отличие от тех, что мы рассматривали раньше.
Тебе придется всегда вручную производить процесс инсталляции kubelet на свои рабочие ноды. Скачать инсталлятор, распаковать и настроить службу.

Итак еще раз, этот компонент не ставит kubeadm, ты ставишь его сам.

Чтобы посмотреть работающий kubelet и его параметры вызови листинг процессов на рабочем узле и отфильтруй по `kubelet`.

Kubelet - это большой компонент, как выполняемым задачам, так и по размеру своего бинарника. Мы будем подробно рассмтривать, как конфигурировать, создавать сертификаты и присоединять его в кластер позже в этом курсе.

А сейчас это все, жду тебя в следующей лекции!



Привет и добро пожаловать на лекцию.
Мы поговорим о Kube Proxy.

В кластере Kubernetes каждый POD может подключиться к любому другому POD.

Это достигается путем развертывания в кластере специального сетевого решения, которое называют сетью POD (`POD network`). Сеть POD - это внутренняя виртуальная сеть, охватывающая все узлы кластера, и к этой специальной сети подключаются все PODs. Через эту сеть обеспечивается возможность всем объектам и компонентам кластера общаться друг с другом. Существует множество решений для развертывания такой сети.

Посмотрим на мой пример. В этом случае у меня есть веб-приложение, развернутое на одной ноде, и приложение базы данных, развернутое на второй. Веб-приложение может подключиться к базе данных, просто используя IP-адрес POD с базой данных.

Но здесь нет гарантии, что IP-адрес базы данных всегда останется прежним.
Если смотрел лекцию о службах, из моего курса для начинающих, то знаешь, что лучший способ для веб-приложения получить доступ к базе данных - использовать службу - `service`.

Итак, мы создаем service для предоставления доступа приложению к базе данных в кластере. Теперь веб-приложение может получить доступ к базе данных, используя имя службы db. Служба также получает назначенный ей IP-адрес.

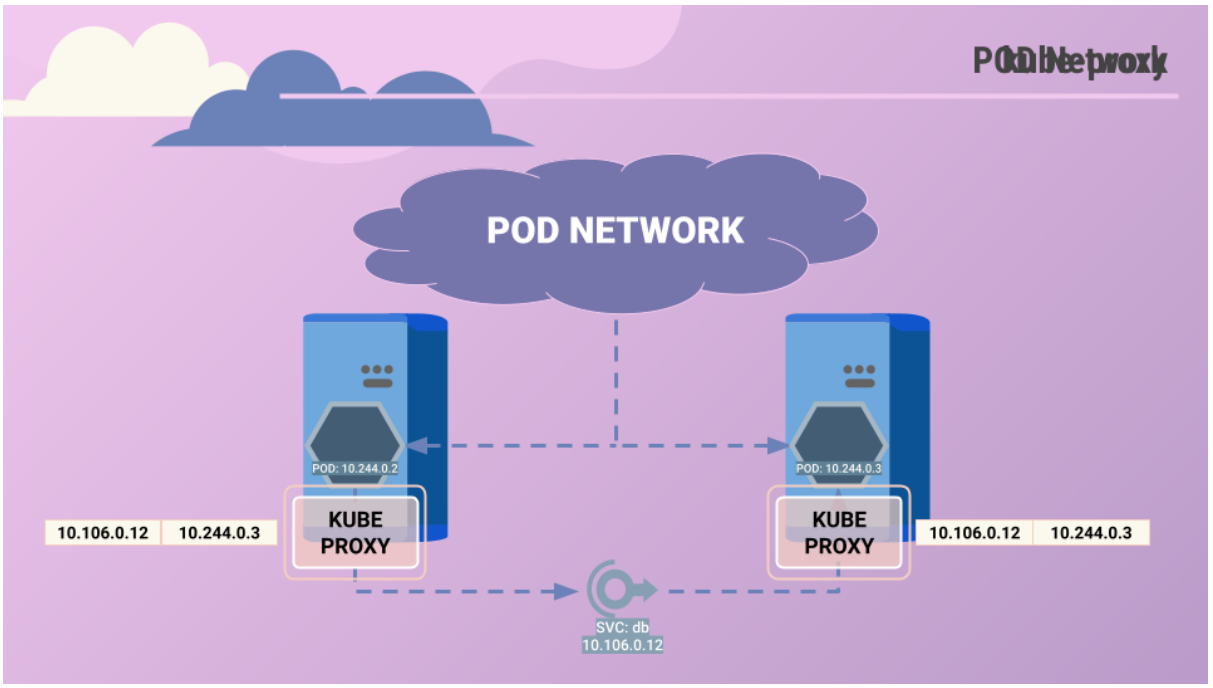
Теперь всякий раз, когда POD пытается обратиться к службе, используя ее IP-адрес или имя, трафик его запроса перенаправляется в POD, на который ссылается этот service. В этом случае на POD с базой данных.

Но что из себя представляет служба и как она получает IP? Присоединяется ли служба к той же сети POD?

Служба не может присоединиться к POD-network, потому что service объект не совсем настоящий. Здесь нет контейнера, как в случае POD, и поэтому у нее нет интерфейсов или активного слушающего процесса, к которому можно обратиться. Services - это виртуальные объекты, это наборы правил, которые живут в памяти нод нашего кластера.

Ранее мы сказали, что службы должны быть доступны в кластере из любого источника. Как же этого добиться?

Вот здесь нам на выручку приходит компонент kube-proxy.



Kube-proxy - это процесс, который выполняется на каждом узле kubernetes-кластера.

Его задача - следить за новыми службами, и каждый раз, когда создается новый service, создавать соответствующие правила.

Таким образом он обеспечивает процесс перенаправления трафика от этих служб к их бэкендам в POD.

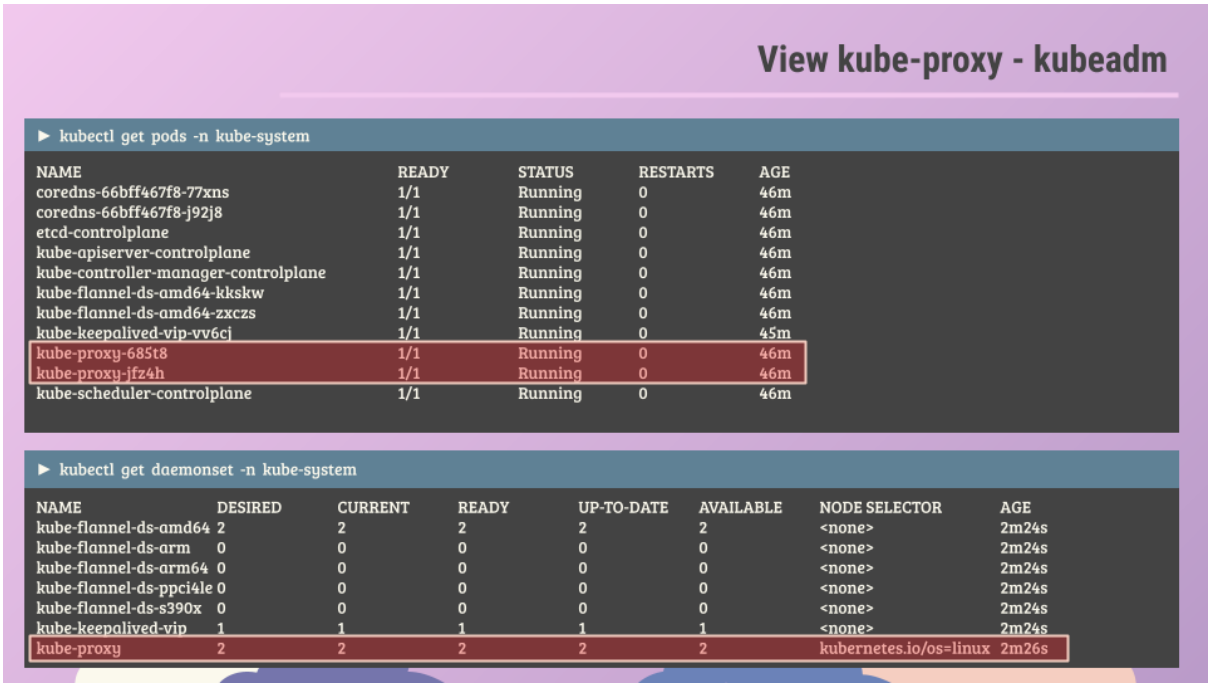
Один из способов сделать это - использовать правила IPTABLES.

В этом случае kube-proxy создает IPTABLES-правило на каждой ноде в кластере. Оно будет осуществлять пересылку трафика, направляемого на IP-адрес службы, равный 10.106.0.12, на фактический IP POD, то есть 10.244.0.3.

Ок, а как kube-proxy настраивает эти службы?

Мы подробнее поговорим об этом в разделе networking, где обсудим сети PODs, и работу служб kube-proxy с сетевой перспективы. Это будет в весьма обширном разделе вниз по течению этого курса. На данный момент наша задача - это общий обзор.

Теперь посмотрим, как установить kube-proxy. Загрузи двоичный файл kube-proxy со страницы релизов Kubernetes. Извлеки его и запусти как службу.



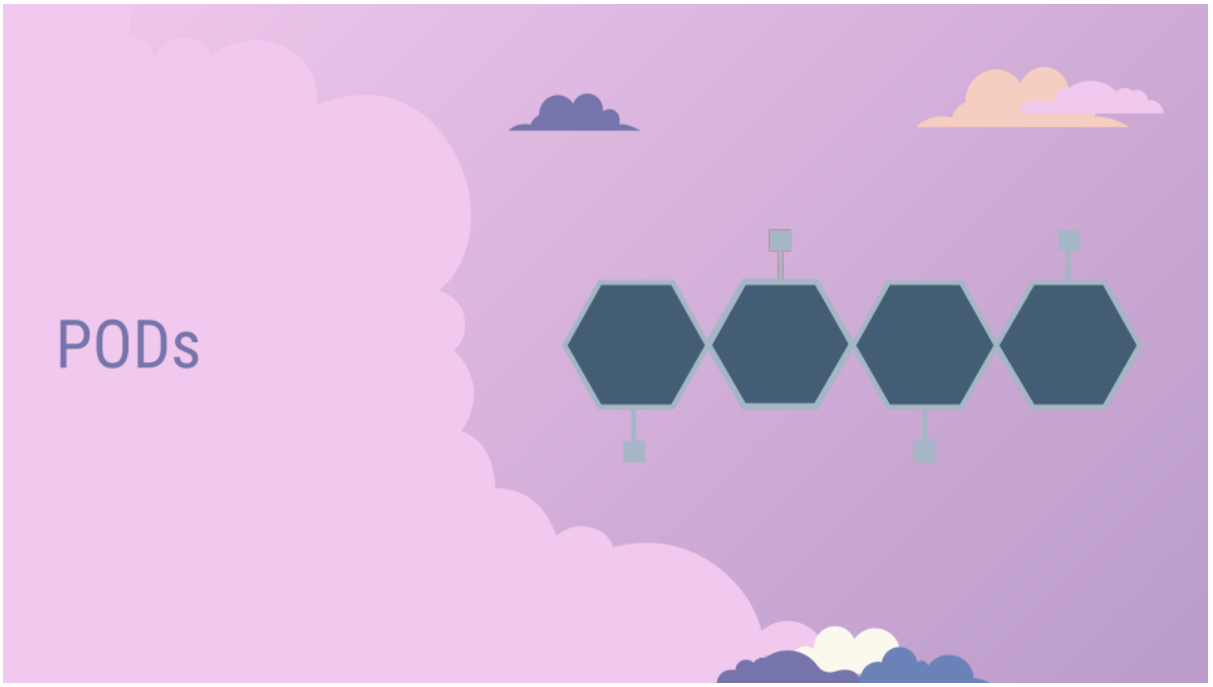
Инструмент kubeadm развертывает kube-proxy как POD на каждой ноде кластера. Фактически он развертывает их как `daemonset`, поэтому на каждом узле кластера всегда развертывается один такой POD.

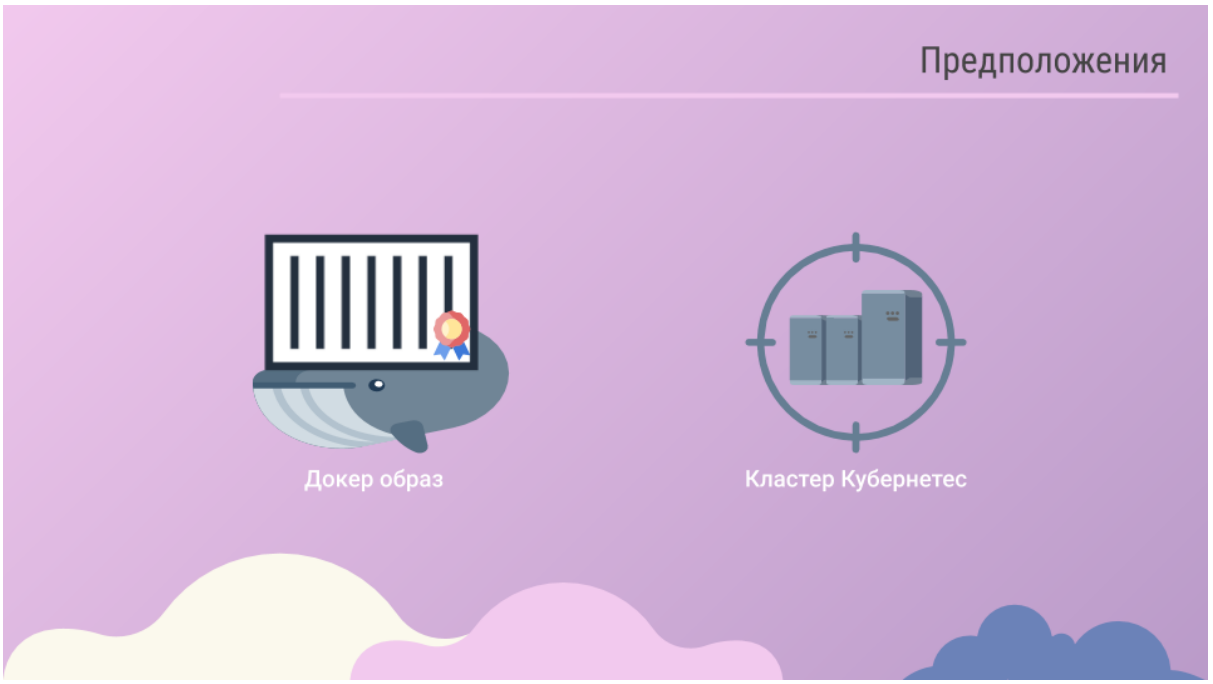
Если ты еще не знаешь о daemonset, не волнуйся, у нас в курсе будет лекция об этом.

Ок, мы сделали общий обзор различных компонентов из области controlplane Kubernetes.

Как уже упоминалось, мы рассмотрим некоторые из них более подробно в следующих разделах этого курса.

Это все в этой лекции.
Увидимся в следующей!

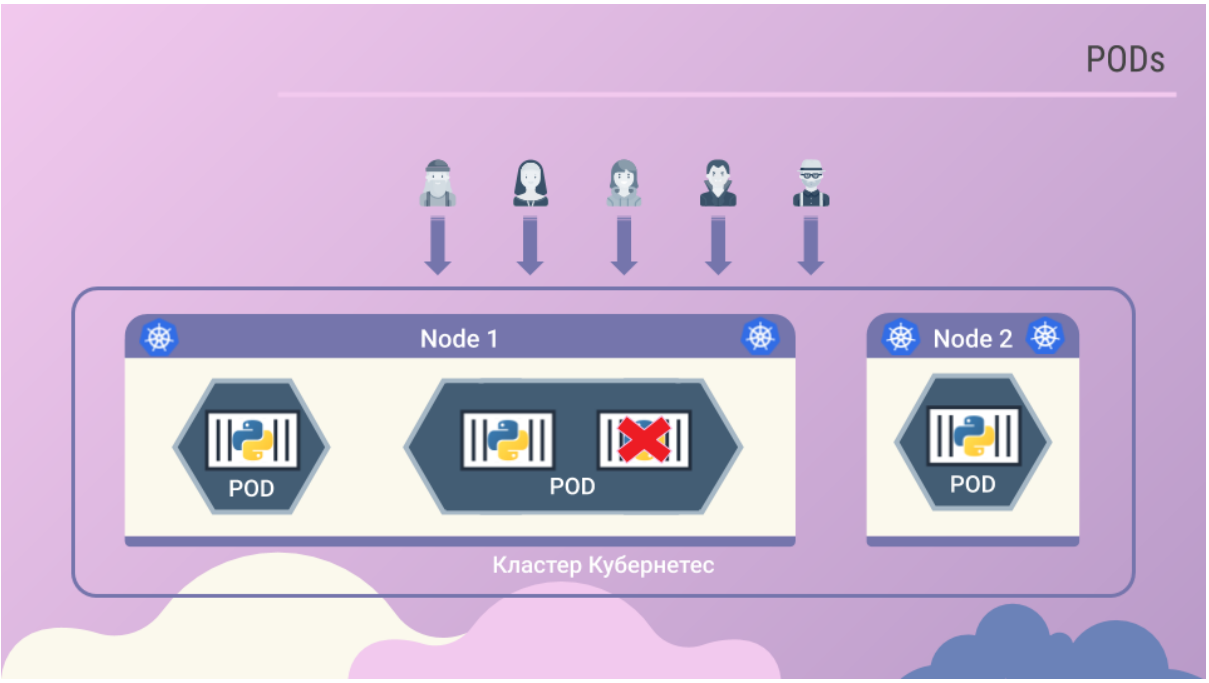




Перед тем, как мы окунемся в PODs, давай предположим, что у нас соблюдены некоторые условия.
А именно, у нас есть приложение, уже разработанное и собранное в докер-образ. Оно доступно нам в докер-репозитории, вроде Dockerub, и Kubernetes может его оттуда забрать. Также мы предположим, что кластер уже установлен, запущен и работает. Он может быть однонодным или иметь большее количество узлов, это неважно. Все сервисы кластера запущены и работают как надо.

Вот здесь появляется первая абстракция Kubernetes с которой мы познакомимся, а именно PODs.

Ранее мы узнали, что конечная цель Kubernetes - развернуть наше приложение в форме контейнеров на наборе машин, которые сконфигурированы как воркер-ноды в кластере.
Однако, Kubernetes не развертывает контейнеры непосредственно на ноды. Он инкапсулирует их в объекты Kubernetes, которые называются PODs или ПОДЫ. POD это отдельный экземпляр приложения. Также POD это самый маленький объект, который ты можешь создать в Kubernetes, самая маленькая деталь этого конструктора.
Тут мы видим очень простой случай: одноузловой кластер Kubernetes с одним экземпляром приложения, работающим в одном контейнере Docker, обернутый в POD. Количество пользователей неизбежно вырастает и тебе требуется отмасштабировать приложение.

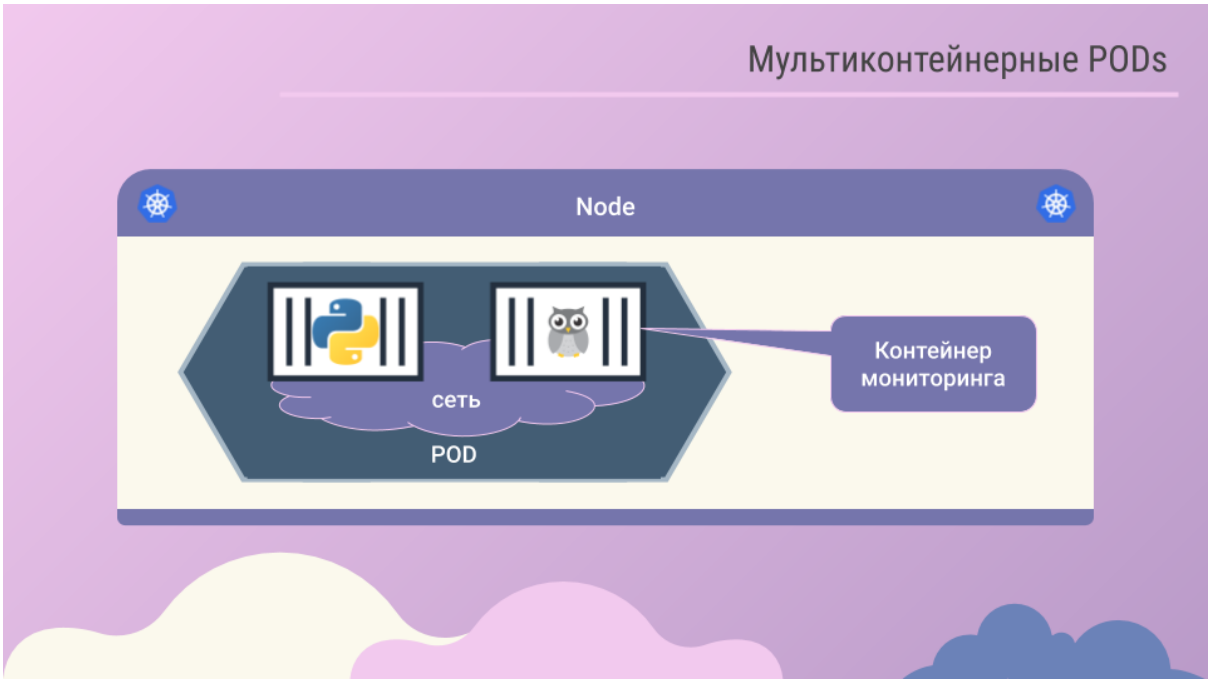


Это значит, что нужно добавить еще экземпляров своего приложения и разделить нагрузку между ними. Как бы ты развернул дополнительные инстансы? Создал бы дополнительный контейнер в том же POD?
Нет. Мы создадим новый под с новым экземпляром этого приложения

Как видишь, у нас теперь два инстанса нашего веб приложения, запущенного в двух разных участках ноды. Что произойдет, если нагрузка вырастет еще больше и текущая нода не сможет вместить в себя еще один дополнительный POD? Ты всегда можешь разместить этот POD на новой ноде кластера. Т.е. увеличить физическую емкость кластера, добавив новую ноду.

Основная мысль здесь, что в POD обычно один контейнер, в которых выполняется приложение, а чтобы отмасштабироваться вверх создай новые PODs, чтобы соскейлится вниз удали существующие.

Не стоит добавлять дополнительные контейнеры в существующие PODs для масштабирования приложения. Принципы реализации масштабирования, балансировки нагрузки между контейнерами и т.д. мы более пристально рассмотрим в дальнейших лекциях.
В данный момент мы просто усваиваем основные принципы.



Только что мы сказали, что в POD обычно один контейнер, но есть ли ограничение на запуск нескольких контейнеров в одном POD?
Нет, в одном POD может быть несколько контейнеров, но эти контейнеры не должны быть одного типа. Как мы только что говорили, если нам нужно масштабировать наше приложение, то для этого мы создадим дополнительные PODs.

В некоторых случаях требуется создать контейнер-помощник, который нес бы функции поддержки приложения из основного контейнера, такие как препроцессинг введенных пользователем данных, обработку файлов, загруженных пользователем или фиксация поведения приложения. Главное требование для таких контейнеров, что время их жизни совпадает с продолжительностью работы основного контейнера с приложением. В этом случае оба контейнера будут частью одного POD, и когда будет создан новый контейнер приложения, с ним вместе будет создан помощник, а когда контейнер приложения умрет, помощник тоже будет уничтожен вместе с подом. Эти контейнеры могут общаться друг с другом напрямую, обращаясь как к localhost, они находятся в одном сетевом пространстве. Также они могут иметь единое пространство для хранения.

Если у тебя есть какое-то непонимание или сомнения в этой теме, я тебя прекрасно понимаю.
Когда я в первый раз изучал эти концепции имел такой же опыт. Мы сейчас посмотрим на это под другим углом.

Давай на минутку отойдем от Kubernetes и поговорим о простых Docker-контейнерах. Допустим, мы разрабатываем процесс или скрипт для развертывания своего приложения на хосте Docker. Тут мы разворачиваем приложение только с помощью команды Docker run Python app, и приложение прекрасно запускается, и наши пользователи видят его. Нагрузка возрастает, и мы разворачиваем больше инстансов приложения через такие же команды Docker run много-много раз. Все вроде бы хорошо и нас не беспокоит.

Но наше приложение не стоит на месте, оно дорабатывается, претерпевает архитектурные изменения, растет и усложняется, и чтобы понимать его работу, мы решили использовать мониторинг. Для этого нам нужен агент-помощник, который будет забирать логи приложения и отсылать их на наш сервер мониторинга. Для работы в паре с контейнером приложения, мы расшарим папку с логами приложения, чтобы агент имел к ним доступ.

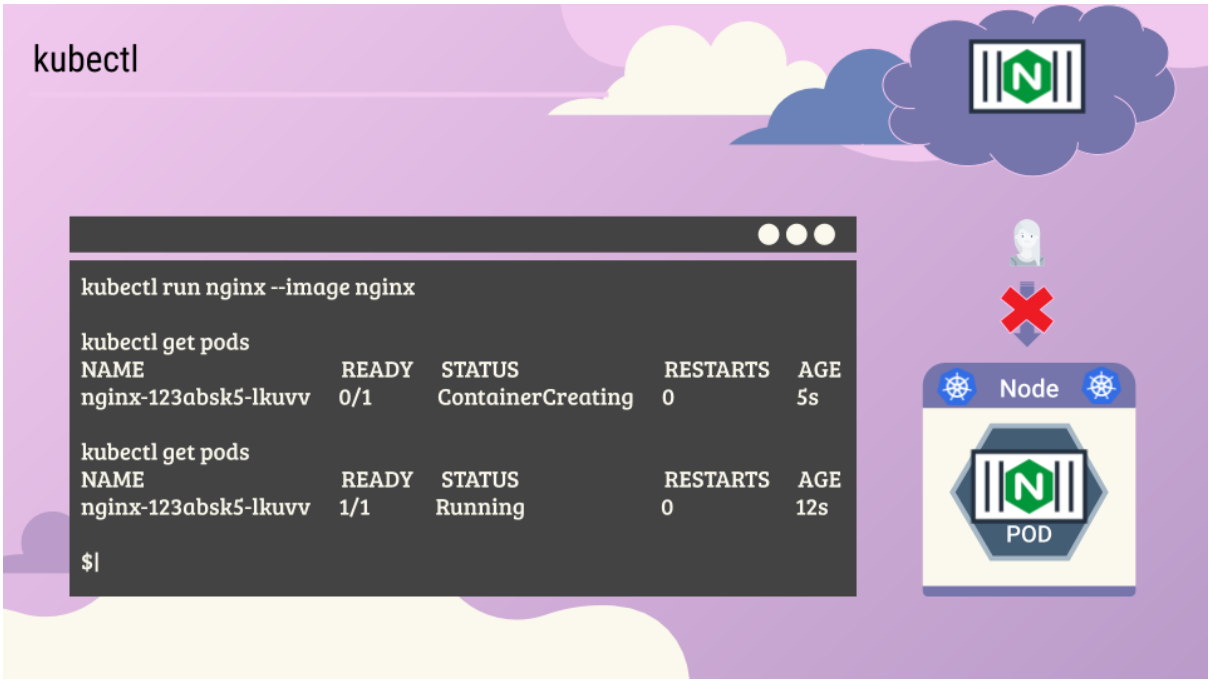
Теперь нам нужно будет поддерживать карту того, как приложения, агенты и общие папки связаны друг с другом. Нам нужно настроить сетевые соединения между соответствующими контейнерами, используя links и custom networks. Также создать нужные volumes и прокинуть их в соответствующие контейнеры.

Чтобы все работало, нужно поддерживать эту карту в рабочем состоянии. И очень важно проследить, чтобы при остановке контейнера приложения, агент мониторинга умирал вместе с ним, а не продолжал исправно мониторить пустоту. При этом, когда мы разворачиваем новый контейнер приложения, мы должны убедится, что не забыли развернуть и контейнер-агент.

С подами Kubernetes делает для нас это все автоматически.
Мы просто указываем, какие контейнеры должны быть в POD.

А они автоматически получают доступ к единому хранилищу, оказываются в одном сетевом пространстве и у них появляется единый жизненный цикл, т.е. они вместе будут созданы и уничтожены.

Даже если наше приложение не очень сложное, и мы могли бы ограничится одним контейнером, Kubernetes все равно требует создавать PODs. Но это хорошо в долгосрочной перспективе, т.к. теперь приложение готово к архитектурным изменениям и масштабированию, которые произойдут в будущем.
Мультиконтейнерные PODs не так распространены, как одиночные контейнеры в POD, и на протяжении курса мы будем разбирать в основном один контейнер на один POD.



Давай посмотрим как создаются PODs. Мы уже знаем о команде kubectl run.
Эта команда создает POD, развертывая в нем контейнер Docker.

Это автоматически создаст POD и развернет экземпляр Nginx из Docker-образа
Откуда возьмется образ приложения? Для этого используется параметр --image
Образ приложения, в данном случае образ Nginx будет скачан из репозитория Docker Hub.

Docker Hub, как мы говорили, публичный репозиторий где находятся последние Docker образы различных приложений и утилит. Ты можешь настроить Kubernetes, чтобы он забирал образы из публичного Docker Hub или приватного репозитория своей организации.

После того, как POD создан, как нам узнать, какие PODs нам доступны?
Команда kubectl get pods поможет нам в этом случае, она покажет нам листинг подов.

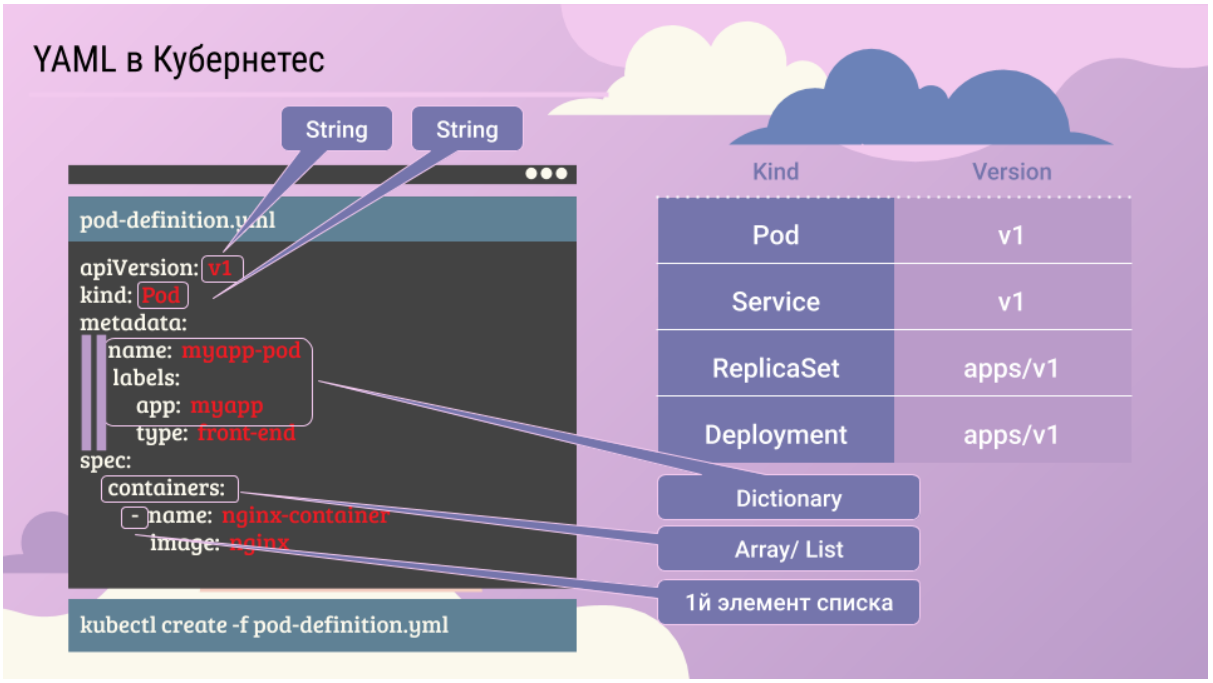
Сейчас POD в статусе создания (creating) и быстро меняется в рабочее состояние (running),когда он действительно запустился.

Мы пока не говорили о способах того, как пользователь может получить доступ к веб-серверу Nginx и в данный момент сервер не доступен для внешних пользователей. Однако внутри ноды ты имеешь доступ к нему и можешь с ним взаимодействовать.

Сейчас мы узнали как развернуть POD. В следующих лекциях мы познакомимся с сетевым взаимодействием и службами и научимся давать доступ к приложению для конечных пользователей.

В предыдущей лекции мы узнали о файлах YAML вцелом.
Теперь мы узнаем как разрабатывать файлы YAML специально для Kubernetes.

Kubernetes использует файлы YAML в качестве входных данных для создания своих объектов,
таких как PODs, ReplicaSets, Deployments, Services и т. д.
Все они имеют сходную структуру.



Манифест Kubernetes всегда содержит 4 верхнеуровневых поля:

- apiVersion
- kind
- metadata
- spec

Это самый верхний или корневой уровень свойств. Считай их братьями и сестрами, детьми одного и того же родителя. Все эти поля ОБЯЗАТЕЛЬНЫЕ, поэтому они ДОЛЖНЫ присутствовать в твоем файле конфигурации.

Давай посмотрим на каждое из них.
Первое - это apiVersion. Это версия API Kubernetes, который мы используем для создания объекта. В зависимости от того, что мы пытаемся создать, мы должны указывать ПРАВИЛЬНЫЙ apiVersion.
Пока мы работаем над POD, мы установим apiVersion как v1. Несколько других возможных значений для этого поля: apps / v1beta1, extensions / v1beta1 и т. д. Позже в курсе мы увидим, для чего они нужны.

Следующий kind. Kind относится к типу объекта, который мы пытаемся создать, в данном случае POD. Ок, мы пишем Pod. С большой буквы. Возможны другие значения, например ReplicaSet, Deployment или Service, я показал это в таблице справа.

Следующее - metadata. Метаданные - это данные об объекте, такие как его имя, метки и т. д.
В отличие от первых двух, имеющих строковое значение, этот - dictionary. Так все, что относится к метаданными, будет являться дочерними элементами, т.е. по правилам YAML сдвинутыми немного вправо. Поэтому мы сдвигаем поля name и labels. Количество пробелов перед этими двумя свойствами не имеют значения, но они должны быть такими же, как и у их братьев. Если у свойства labels слева будет больше пробелов, чем у name, то labels станут дочерним элементом свойства name, а не его братом, что будет неверно. По этой же логике оба свойства должны иметь БОЛЬШЕ пробелов, чем их родительский элемент metadata и располагаться правее родителя. Если же все трое имеют одинаковое количество пробелов перед собой, они становятся братьями, что тоже неверно. В metadata поле name представляет собой строковое значение, поэтому пишем сюда название для POD - myapp.

Labels - это dictionary, т .е labels - это словарь в словаре метаданных. И он может иметь любые пары "ключ-значение" по твоей необходимости. Я добавил в labels параметр app со значением myapp. Точно так же можно добавлять и другие метки по своему усмотрению, которые позволят отслеживать созданные объекты позже во время работы POD. К примеру есть 100 PODs с приложениями, работающими в качестве фронтенда, и 100 PODs запускающие бекенд или базу данных. Будет очень СЛОЖНО сгруппировать эти PODs после их развертывания. Но если пометить их при создании как фронтенд, бекенд или базой данных, то можно легко фильтровать PODs на основе этой метки type.

ВАЖНО отметить, что в metadata ты можешь указать только name, labels или другие поля, которые Kubernetes относит к метаданным. НЕЛЬЗЯ добавлять другие поля по своему усмотрению. Однако под labels МОЖЕТ быть любой вид пар "ключ-значение" по твоему усмотрению. Поэтому ВАЖНО понимать, какой из параметров Kubernetes ожидает увидеть.

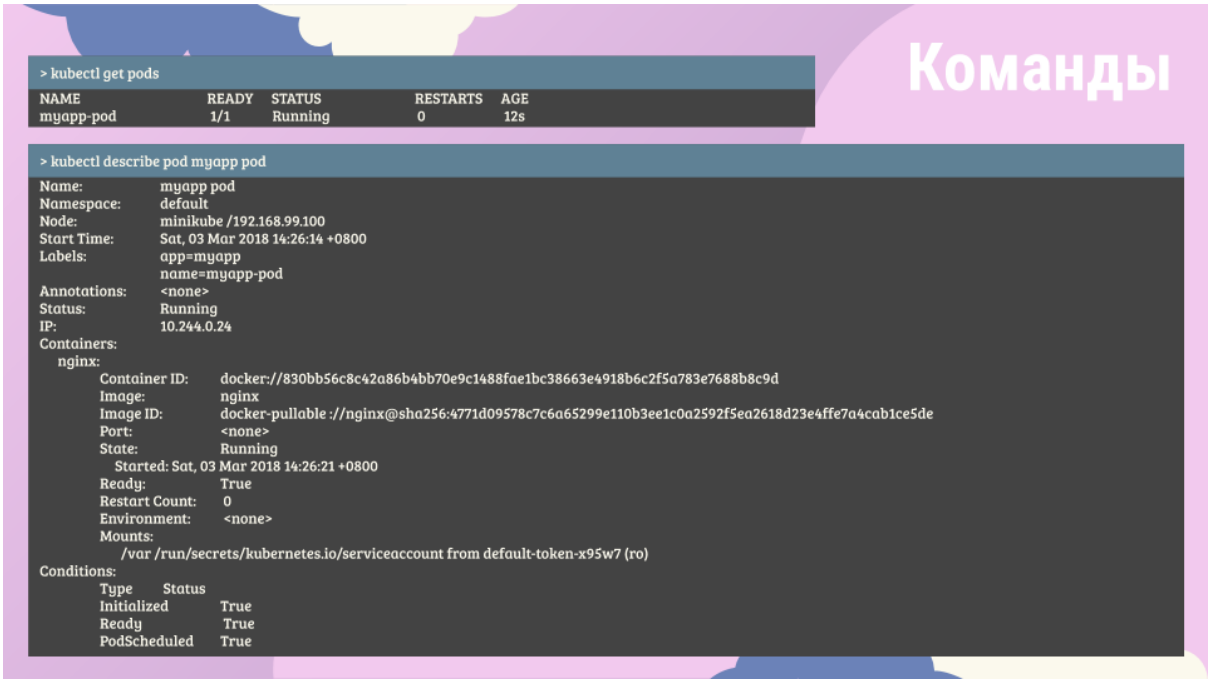
До сих пор мы упоминали только тип и имя объекта, который нам нужно создать. Это POD с названием myapp-pod.Но пока мы не указали контейнер или образ, который нам нужно запускать в POD. Последний раздел в файле конфигурации - это спецификация, пишется как spec.

В зависимости от объекта, который собираемся создать, мы предоставляем Kubernetes дополнительную информацию, относящуюся к этому объекту. Она разная для разных типов объектов, поэтому важно запомнить или обратиться к разделу документации, чтобы указать правильный формат. Поскольку мы создаем POD только с одним контейнером, это сделать легко.

Spec - это dictionary, в нем есть свойство containers, которое представляет собой list. Причина, по которой это свойство является списком, состоит в том, что POD может иметь несколько
контейнеров внутри, как мы узнали из лекции ранее. В данном случае мы добавим только один элемент в список, так как нам нужен только один контейнер в POD. Элемент в list является dictionary, в него пишем свойства name и image. Значение для образа - nginx, это имя образа в Docker репозитории.

После сохранения файла запусти команду kubectl create -f далее имя созданного файла, у нас это файл pod-definition.yml, и Kubernetes создаст POD.

Ок, в качестве итога, еще раз вспомним 4 корневых свойства.
apiVersion , kind, metadata и spec. Начинаем с них, устанавливая значения в зависимости от создаваемого нами объекта.



Итак, мы создали POD. Как нам увидеть, что произошло дальше?

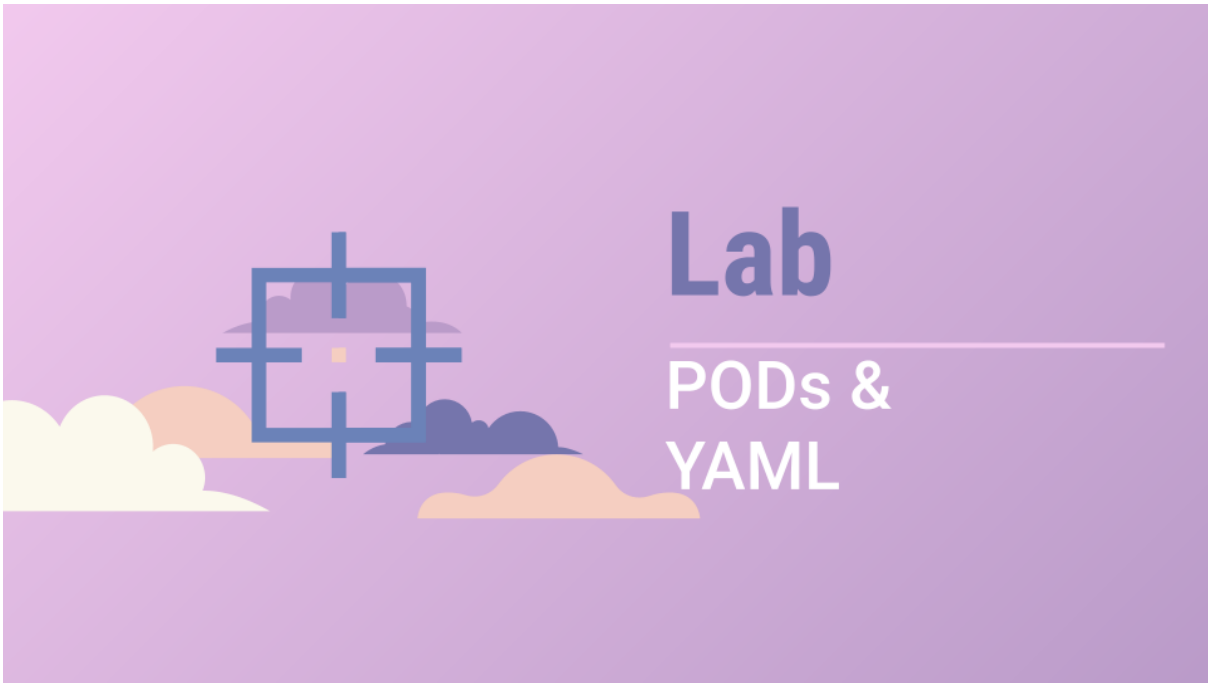
Использовать команду kubectl get pods, которая покажет тебе список всех доступных PODs. В данном случае всего один.

Demo

Lab Environment

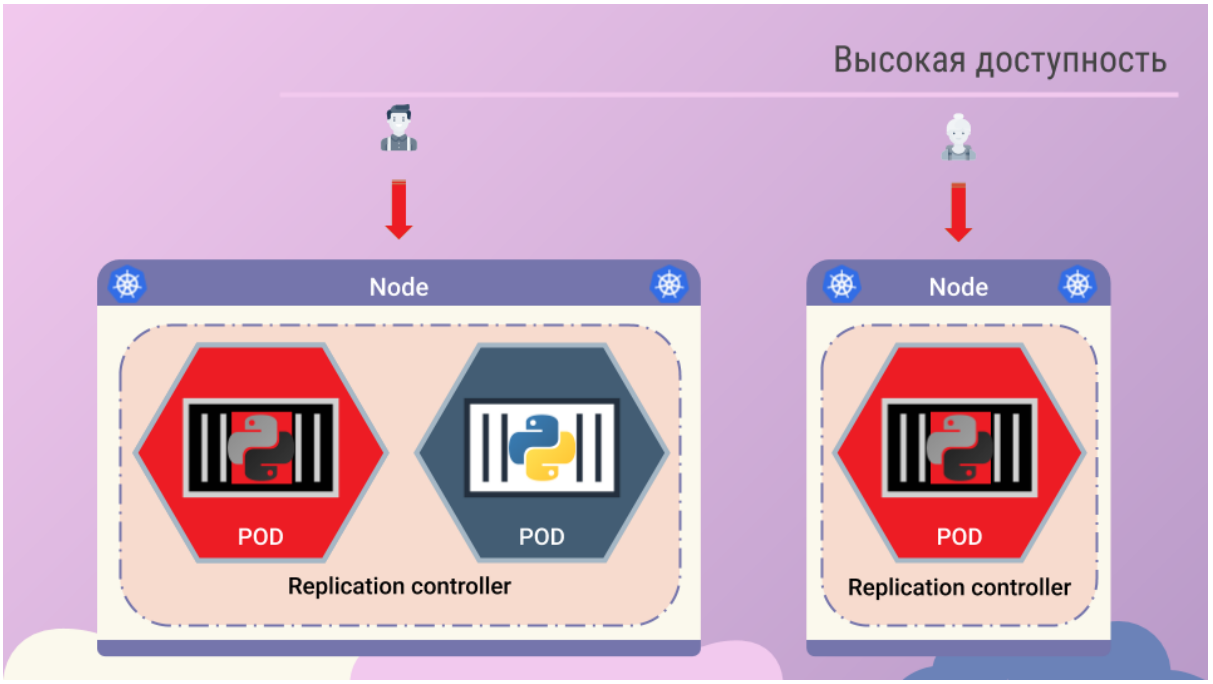
Demo

IDE & YAML


Lab
PODs &
YAML

На этой лекции мы начнем обсуждать контроллеры Kubernetes. В общем смысле контроллеры - это мозг Kubernetes. Это процессы, которые отслеживают состояние объектов и реагируют соответствующим образом.

Сейчас мы разберем один из контроллеров, а именно контроллер репликации. Итак, что такое реплика и зачем нам нужен контроллер репликации?

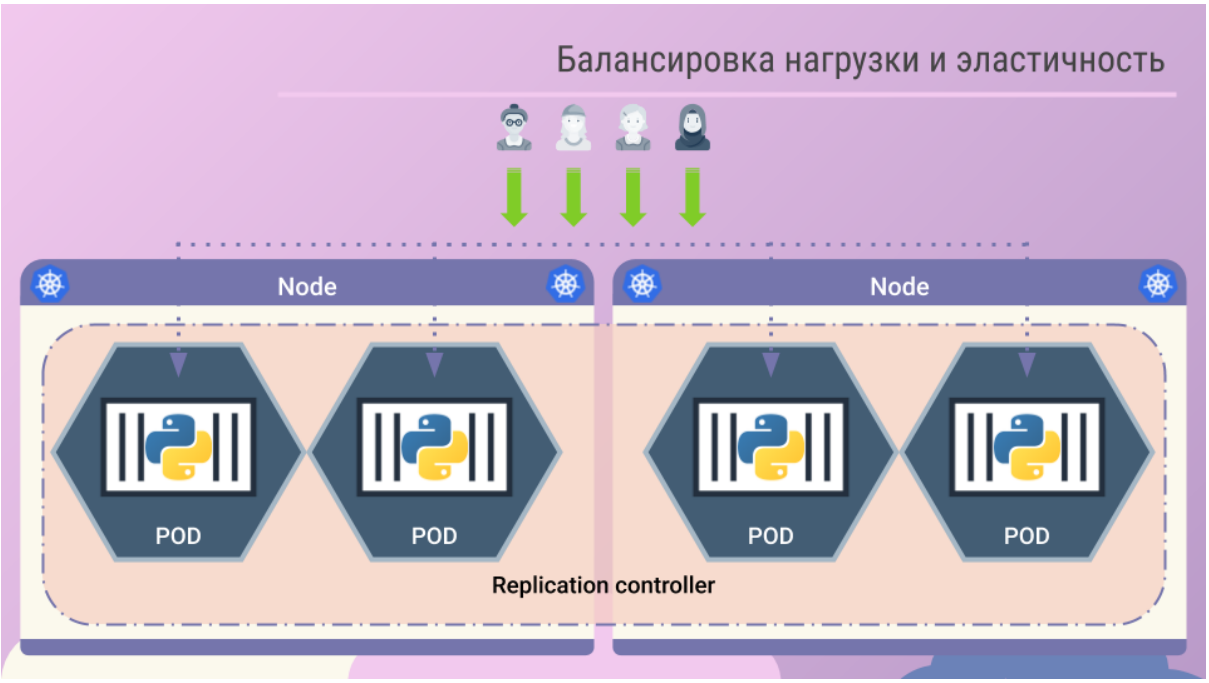


Вернемся к нашему первому сценарию, где приложение запускалось на одном POD. Что произойдет, если вдруг наше приложение вылетит с ошибкой и POD выйдет из

строя? Очевидно, что пользователи больше не смогут получить доступ к нашему приложению.

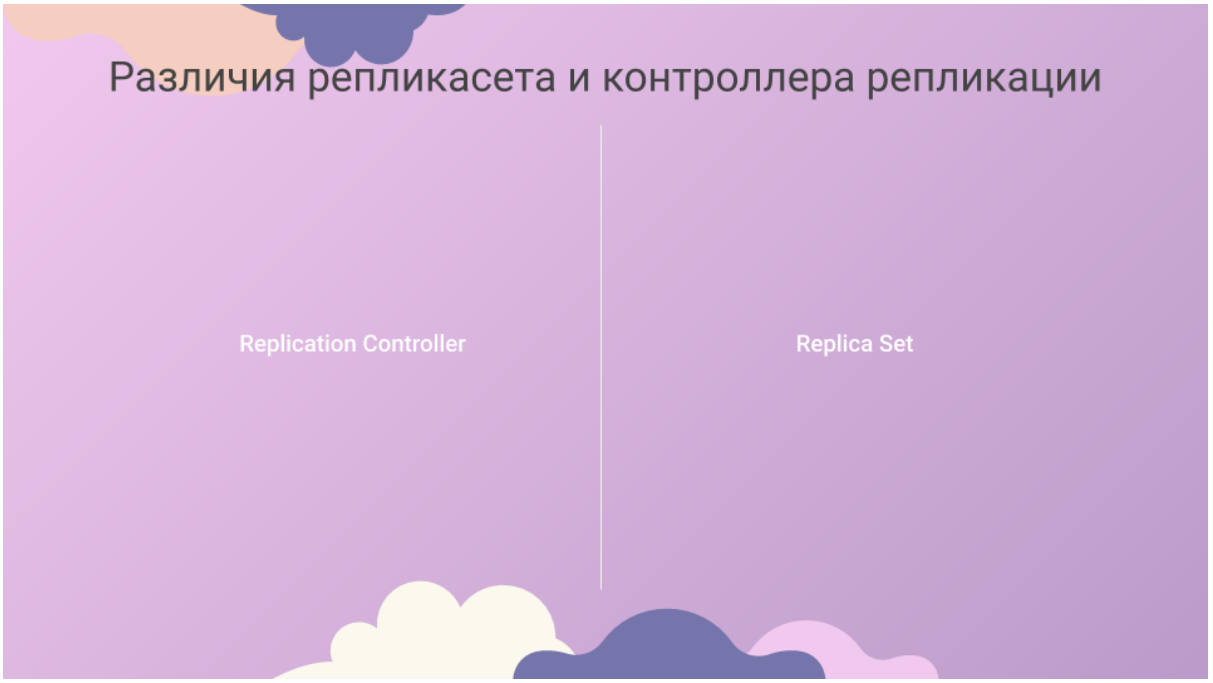
Чтобы пользователи не потеряли доступ к приложению, мы хотели бы, чтобы одновременно работало более одного экземпляра или POD. Таким образом, если один POD выйдет из строя, то приложение все еще будет доступно из других экземпляров.

Контроллер репликации помогает нам запускать несколько экземпляров POD в кластере Kubernetes, тем самым обеспечивая высокую доступность. Значит ли это, что тебе не стоит пользоваться контроллером репликации если POD всего один? Нет! Даже если у тебя один POD, контроллер репликации может помочь, автоматически запуская новый POD при выходе из строя существующего. Таким образом, контроллер репликации гарантирует, что указанное количество исправных PODs будет постоянно. Будь это всего лишь 1 POD или сотня.



Другая причина его использования - распределение нагрузки между несколькими PODs. К примеру, в этом простом сценарии у нас есть один POD и некоторое количество пользователей. Когда число пользователей увеличивается, мы разворачиваем дополнительный POD, для балансировки нагрузки. Если нагрузка продолжает расти, то в какой-то момент мы исчерпаем вычислительные ресурсы на первой ноде. Тогда мы можем развернуть дополнительные PODs на других нодах кластера.

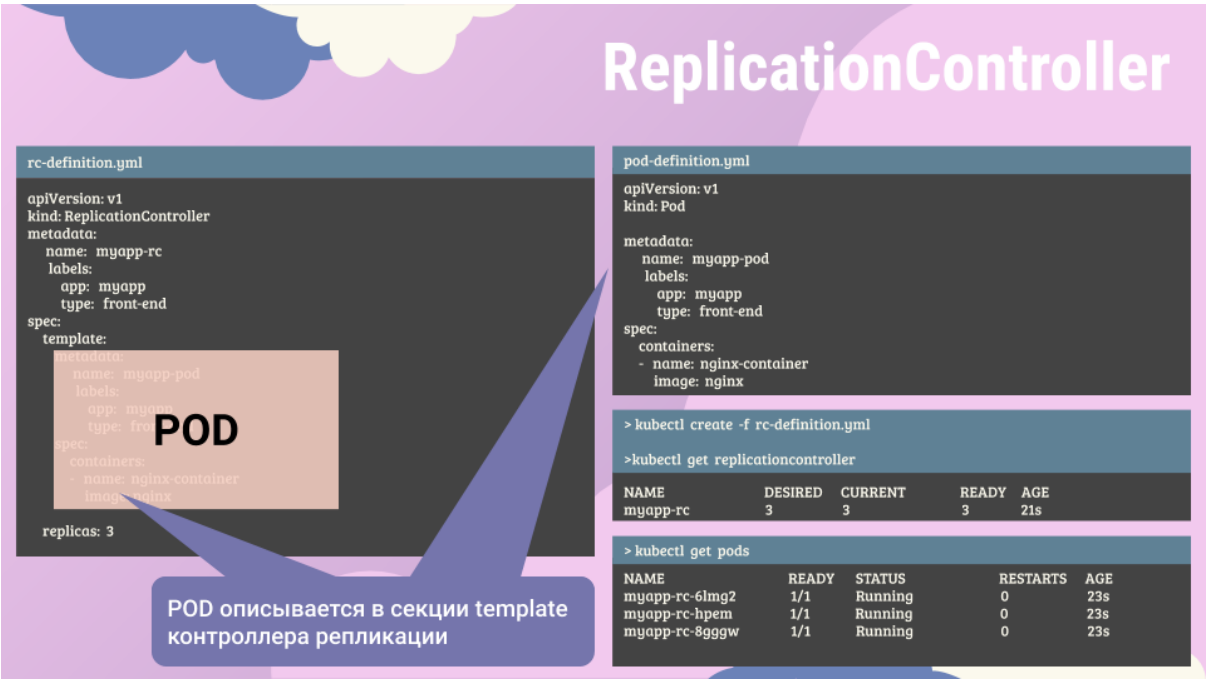
Как видишь, контроллер репликации охватывает сразу несколько узлов кластера. Это помогает нам сбалансировать нагрузку между несколькими PODs на разных узлах, а также масштабировать наше приложение при увеличении спроса.



Важно отметить, что в Kubernetes есть два похожих объекта: ReplicationController и ReplicaSet.
У обоих одна и та же цель, но они не одинаковы.

ReplicationController - это более старая технология, которую заменяет ReplicaSet.
ReplicaSet - это новый рекомендуемый способ настройки репликации.

Однако всё, что мы обсуждали ранее о контроллерах репликации, применимо к обеим этим технологиям. Есть незначительные различия в способах работы каждого из них, и мы рассмотрим это чуть позже. В дальнейшем мы будем придерживаться технологии ReplicaSet во всех наших демонстрациях.



Давай теперь посмотрим, как создается контроллер репликации. Как и в предыдущей лекции, мы начнем с создания файла определения контроллера репликации.
Назовем его rc-definition.yml.
Как и в любом файле определения kubernetes будет базовых 4 раздела.
ApiVersion, kind, metadata и spec.

ApiVersion зависит от того, что мы создаем. Объект ReplicationController kubernetes поддерживает apiVersion v1. Поэтому запишем v1.

Kind будет ReplicationController.

В Metadata мы добавим name и со значением myapp-rc. Также добавим несколько labels, чтобы в будущем понимать, что это за приложение и какая у него роль.
Пока это очень похоже на создание POD.

Следующая и самая важная часть манифеста - это спецификация, элемент spec. В Kubernetes YAML файле раздел спецификации определяет, что находится внутри объекта, который мы создаем.
Функция контроллера репликации это создание нескольких экземпляров POD. Но что именно за POD это будет?

Мы создаем раздел template в разделе spec, чтобы предоставить шаблон POD, который будет использоваться контроллером репликации для создания реплик. Ок, как мы ОПРЕДЕЛЯЕМ шаблон POD? Это не так сложно, потому что мы уже делали это в предыдущем упражнении.
Вспомни, как в предыдущем упражнении мы создали файл определения POD. Мы можем повторно использовать содержимое того же файла для заполнения раздела

template. Перемести все содержимое файла pod-definition.yml, за исключением первых двух строк - apiVersion и kind в раздел template контроллера репликации.

Помни, что все, что мы перемещаем, должно находиться В СЕКЦИИ template. Это значит, что они должны быть расположены справа и иметь больше пробелов перед собой, чем строка template.

Посмотрим, что у нас получилось. У нас теперь есть два раздела metadata, один для ReplicationController, а другой для POD. Еще у нас есть два раздела spec, по одному на каждый. Мы вложили один файл определений в другой, определение ReplicationController стало родительским, а определение POD - дочерним.

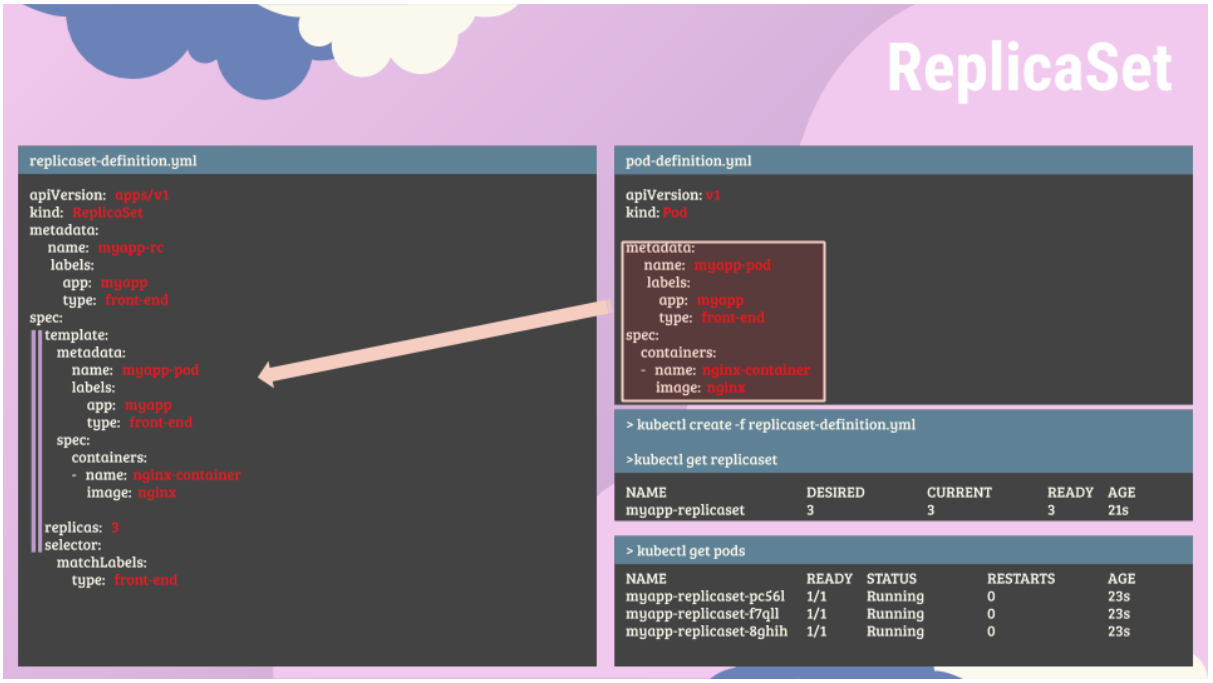
Так, чего-то еще не хватает. Мы забыли, сколько реплик нам нужно в ReplicationController. Для этого добавим в спецификацию еще одно свойство, называемое replicas, и укажем необходимое количество реплик.

Запомни, что шаблон и реплики являются прямыми потомками раздела спецификации.
Они братья и должны находиться на одной вертикальной линии, перед ними должно быть одинаковое количество пробелов.

Когда файл будет готов, запусти команду kubectl create и укажи файл, используя параметр -f.
Создастся ReplicationController. Когда ReplicationController начинает работать, он создает POD используя шаблон из поля template. И делает это столько раз, сколько мы указали, в данном случае это 3.

Для просмотра списка активных ReplicationController воспользуйся командой kubectl get replicationcontroller, ты получишь их список. Тут присутствует информация о желаемом количестве реплик, текущее количество реплицированных PODs и сколько из них готово.

Чтобы увидеть PODs, созданные ReplicationController, выполни команду kubectl get pods. Как мы видим запущены 3 PODs. Обрати внимание, что имена начинаются с названия ReplicationController, которым является myapp-rc, это означает, что они созданы автоматически контроллером репликации.



Мы только что рассмотрели ReplicationController. Давай теперь посмотрим на ReplicaSet. Он очень похож на ReplicationController.

Как обычно, сначала идет apiVersion, kind, metadata и spec. Однако apiVersion будет другим. Небольшое отличие: apps / v1. Для ReplicationController было просто v1. Если ты укажешь неправильно, то, вероятно, получишь ошибку, которая выглядит примерно так. Kubernetes говорит, что нет совпадений kind для ReplicaSet, это потому, что указанная версия api kubernetes не поддерживает ReplicaSet.

Kubernetes растет и развивается, и некоторые решения могут перемещаться по версиям api и даже быть одновременно в двух группах.

Kind будет ReplicaSet, также добавим имя и метки в метаданные. Раздел спецификации очень похож на аналогичный из ReplicationController. В нем есть раздел template, где как и раньше, мы предоставляем определение POD.

Итак, давай скопируем содержимое из файла определения POD, количество реплик равно 3. Однако есть одно важное различие между ReplicationController и ReplicaSet. Для ReplicaSet требуется определение селектора. Раздел selector помогает ReplicaSet понять, какие PODs ему принадлежат.

Но зачем указывать, какими PODs владеет ReplicaSet, если вы прописали сам манифест POD внутри определения ReplicaSet?

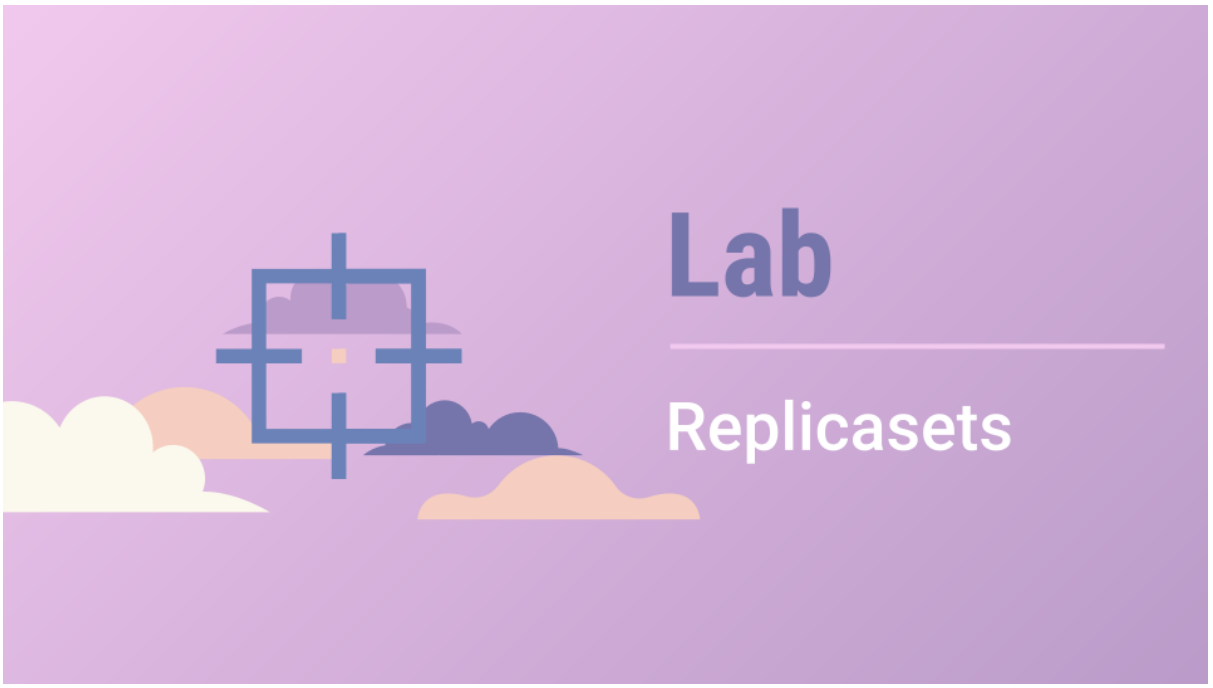
ПОТОМУ ЧТО, ReplicaSet ТАКЖЕ может управлять PODs, которые не были созданы как часть ReplicaSet.

Допустим, были созданы PODs ДО создания ReplicaSet, у них есть labels, значения которых совпадают с указанными в элементе selector ReplicaSet. ReplicaSet начнёт учитывать ЭТИ PODs при создании своих реплик.

Мы подробнее рассмотрим этот процесс совсем скоро. Но прежде чем мы перейдем к этому, я хотел бы упомянуть, что selector - одно из основных различий между ReplicationController и ReplicaSet. Selector не является обязательным полем в случае ReplicationController, но он доступен.
Если мы опустили его, как мы это делали раньше, предполагается, что он совпадает с метками, указанными в манифесте POD.

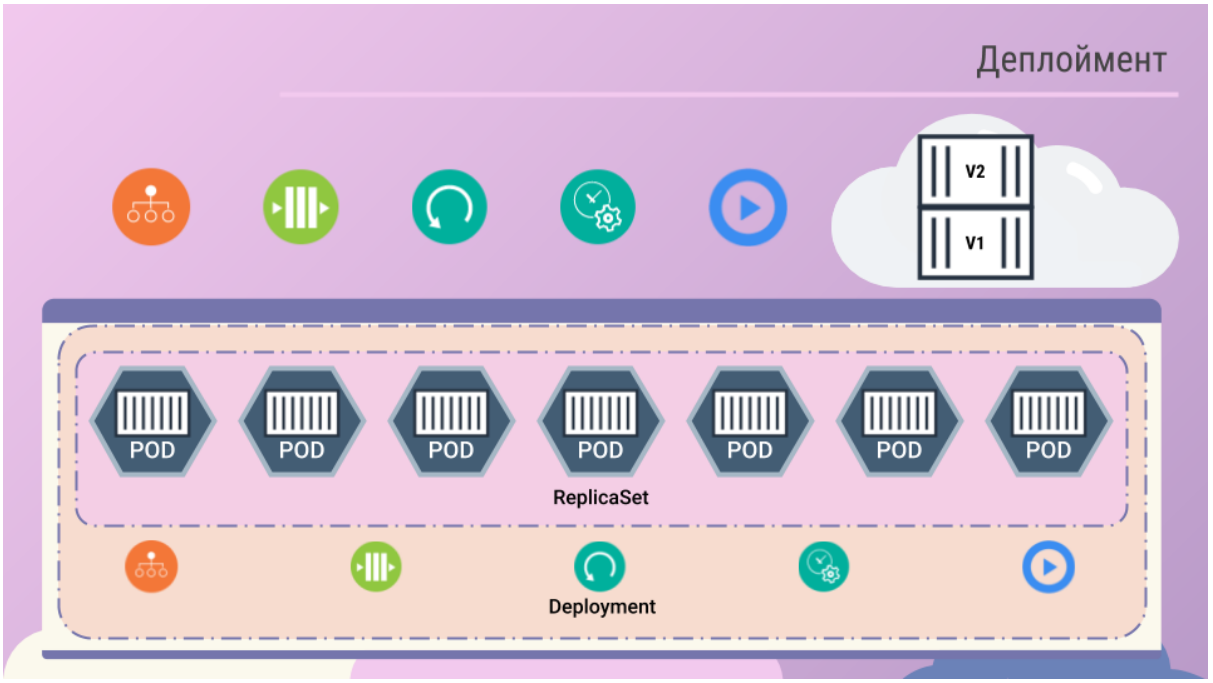
В случае ReplicaSet, определить selector необходимо явно и это должно быть записано в виде matchLabels, как показано здесь. Селектор matchLabels просто сопоставляет принадлежащие ему метки с полем labels из POD. Selector ReplicaSet также предоставляет множество других вариантов сопоставления меток, которые не были доступны в ReplicationController.

И, как всегда, для создания ReplicaSet запусти команду kubectl create, указав файл определения в качестве входных данных. Для просмотра созданных ReplicaSet выполни kubectl get replicaset. Чтобы получить список PODs, запусти команду kubectl get pods.



Давай ненадолго отвлечемся от PODs, replicaSets и прочих концепций Kubernetes и поговорим о вариантах развертывания приложений в продакшене.

Допустим у нас есть веб-сервер, который необходимо развернуть в производственной среде. Очевидно, что нам нужен не ОДИН, а много таких экземпляров. Идет время и когда в докер-реджистри станут доступны новые версии сборки приложения, нам потребуется бесшовно ОБНОВИТЬ экземпляры работающих контейнеров. Нам не подойдет вариант, когда PODs обновляются все сразу, поскольку это может привести к простою и повлияет на наших пользователей.



Нам нужен процесс, который обновит контейнеры один за другим. Такое обновление называется Rolling Update.

Предположим, что одно из выполненных нами обновлений привело к непредвиденной ошибке, и нас попросят отменить последнее действие. Нам нужна возможность откатиться от недавно внесенных изменений.

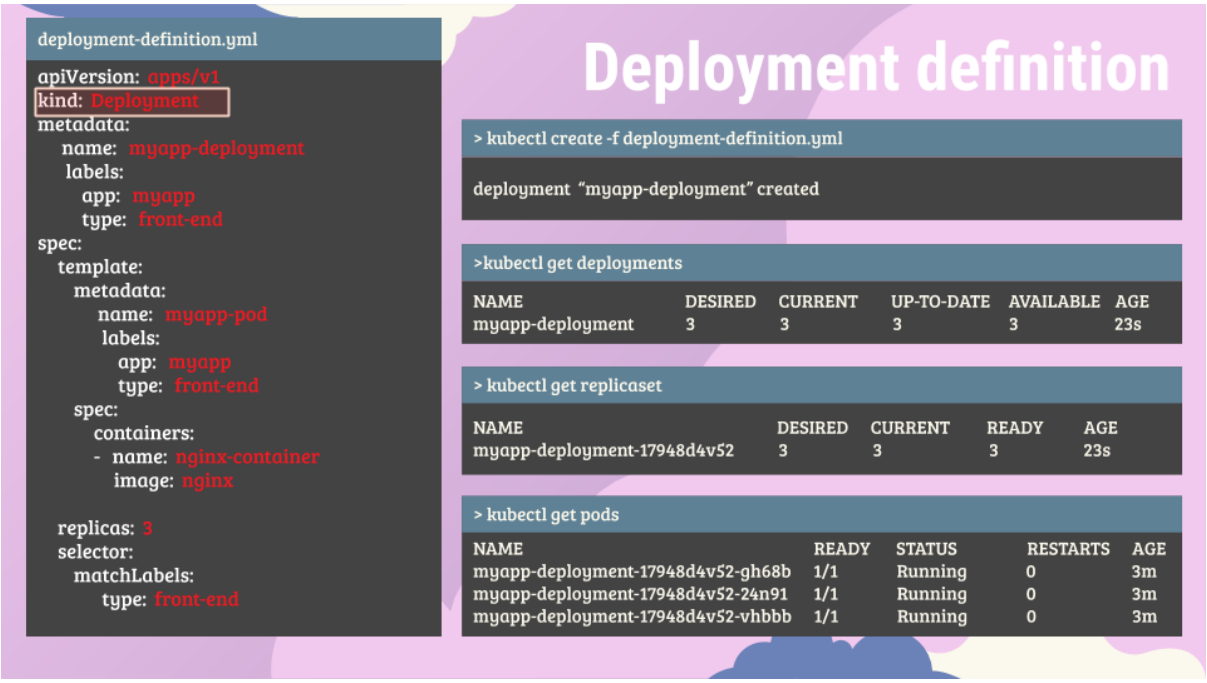
Наконец, если нам требуется внести изменения в свою среду, например, пропатчить ОС ноды или изменить распределение ресурсов вкупе с масштабированием, или какие-то другие вещи.

Мы не хотим применять каждое изменение сразу после исполнения команды, а предпочитаем поставить нагрузку на паузу, произвести требуемые изменения и затем возобновить работу, чтобы все изменения развертывались вместе.

Все эти возможности доступны в deployments Kubernetes. До сих пор в этом курсе мы обсуждали PODs, которые развертывают отдельные экземпляры нашего приложения, такие как веб-приложение в данном случае. Каждый контейнер был инкапсулирован в POD. Несколько таких POD развертываются с использованием ReplicationControllers или ReplicaSets.

Их старший товарищ называется deployment, он представляет собой объект Kubernetes, который находится выше в иерархии и обладает всеми навыками PODs и Replicasets.

Deployment также дает нам возможность обновить принадлежащие ему экземпляры плавно, используя процесс непрерывного обновления (rolling update), позволяет отменять изменения, а также приостанавливать и возобновлять изменения в развертываниях.

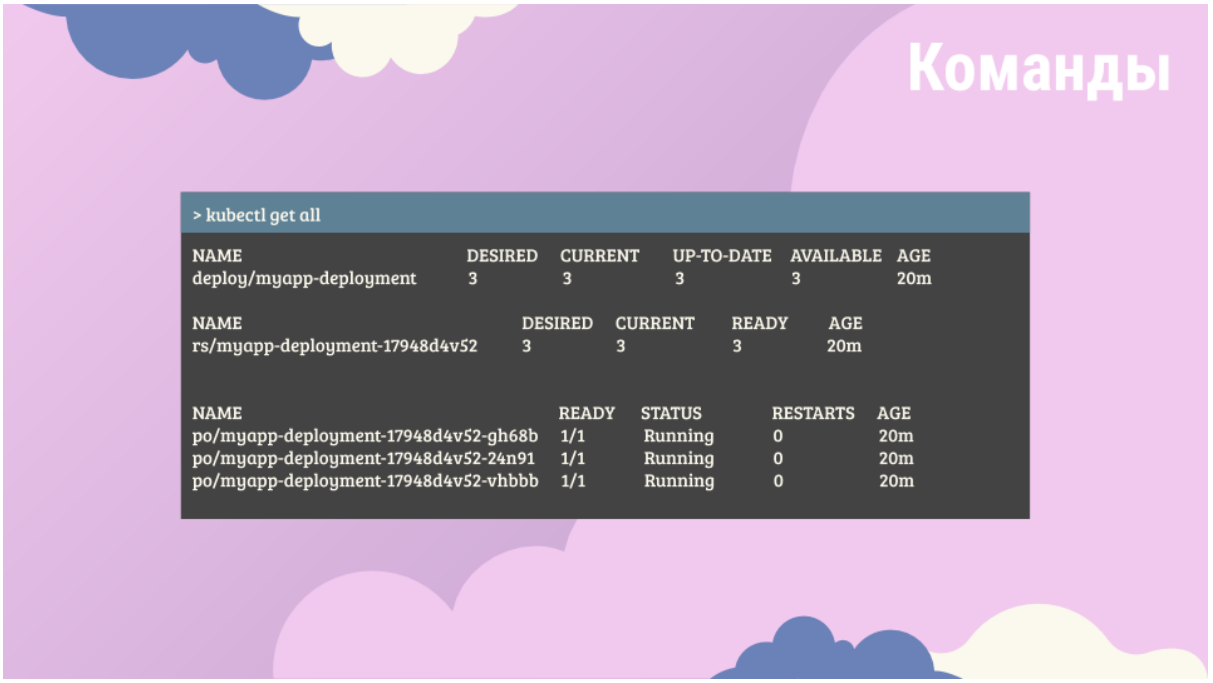


Итак, как нам создать deployment. Как и в случае с предыдущими объектами, мы сначала создаем файл определения deployment. Содержимое манифеста развертывания точно такое же, как и манифеста Replicaset, за исключением того, что значение kind теперь Deployment.

Пройдемся по содержимому файла.
У него есть apiVersion, который представляет собой apps / v1, metadata с name и labels, поле spec с секцией template, а также replicas и selector. Элемент template это как обычно определение POD.

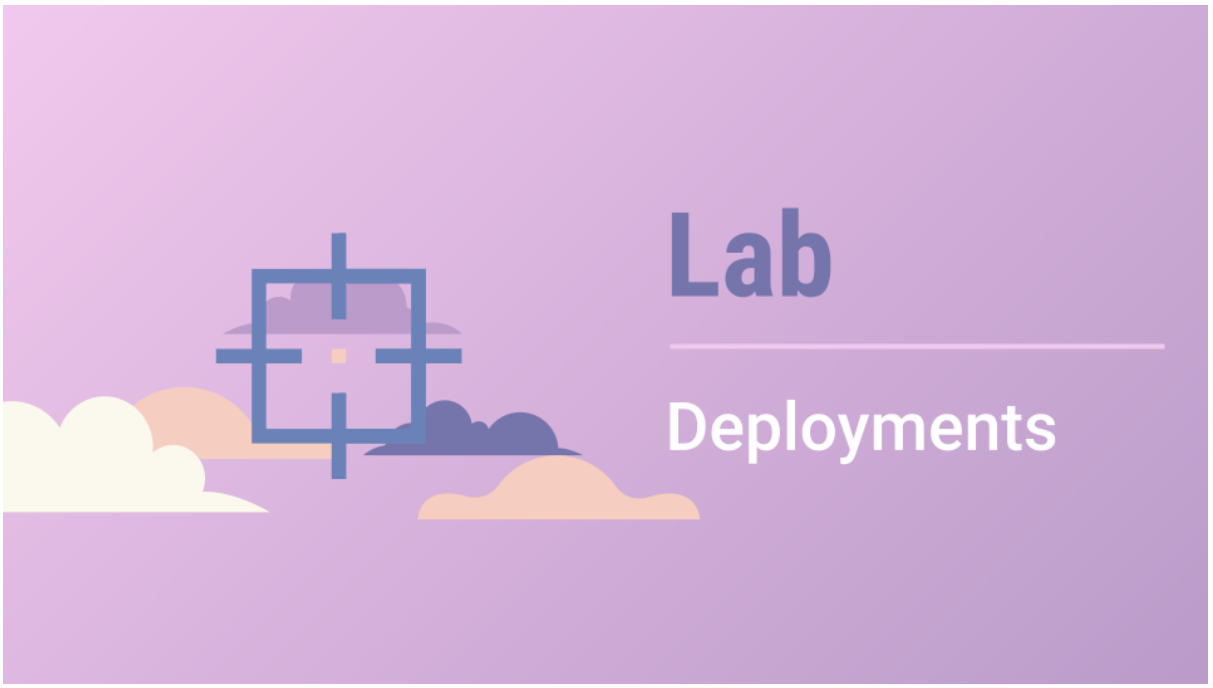
Когда файл будет готов, запусти команду kubectl create и укажи созданный файл определения deployment. Затем запусти команду kubectl get deployments, чтобы увидеть только что созданный deployment. Deployment автоматически создает ReplicaSet. Таким образом, запустив команду kubectl get replcaset, мы увидим новый ReplicaSet в названии deployment. ReplicaSet в конечном итоге создают PODs. Поэтому, если запустить команду kubectl get pods, мы увидим что в имени POD отражено название Deployment и ReplicaSet.

Разница между Deployment и ReplicaSet в данный момент не очевидна, за исключением того, что эти объекты Kubernetes называются по-разному. Немного терпения, в ближайших лекциях мы увидим как использовать преимущества развертываний, используя их широкие возможности, о которых мы говорили в начале лекции.



В конце лекции я хочу показать тебе команду, которая покажет все объекты за один раз.

Это kubectl get all. Используй ее чтобы увидеть список всех Deployments, ReplicaSets и PODs созданных в системе.



Привет и добро пожаловать. Мы продолжаем знакомиться с основными примитивами Kubernetes. В этой лекции мы познакомимся с пространствами имен в Kubernetes.

Начнем с аналогии.

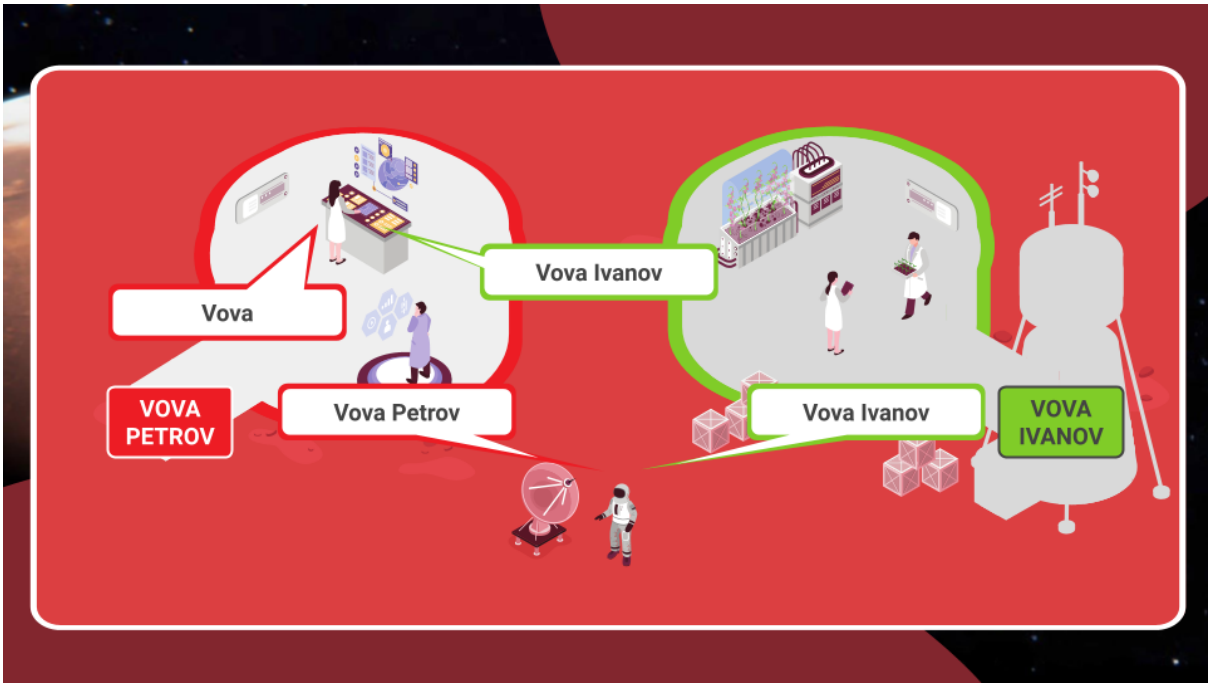
Предположим, на Марсе есть поселение с двумя лабораториями. Одна занимается межпланетной связью, а другая биологическая. В них есть ученые по имени Вова. Один работает в одной лаборатории, другой в соседней.

Чтобы отличать их друг от друга, мы называем их по фамилиям: Иванов и Петров.

В каждой лаборатории есть другие сотрудники: лаборанты, специалисты, обслуживающий персонал. Внутри лаборатории эти люди обращаются друг к другу просто по имени.

Например, доктор Варя из первой лаборатории обращается к Вове Петрову просто как к Вове.

Однако, если Варе нужно обратиться к кому-то за пределами своей лаборатории, она будет использовать полное имя, например Вова Иванов. Также, если кто-то за пределами этих лабораторий захочет вызвать Вову по рации, ему нужно будет уточнить к кому он обращается, использовав полное имя для обозначения вызываемого.



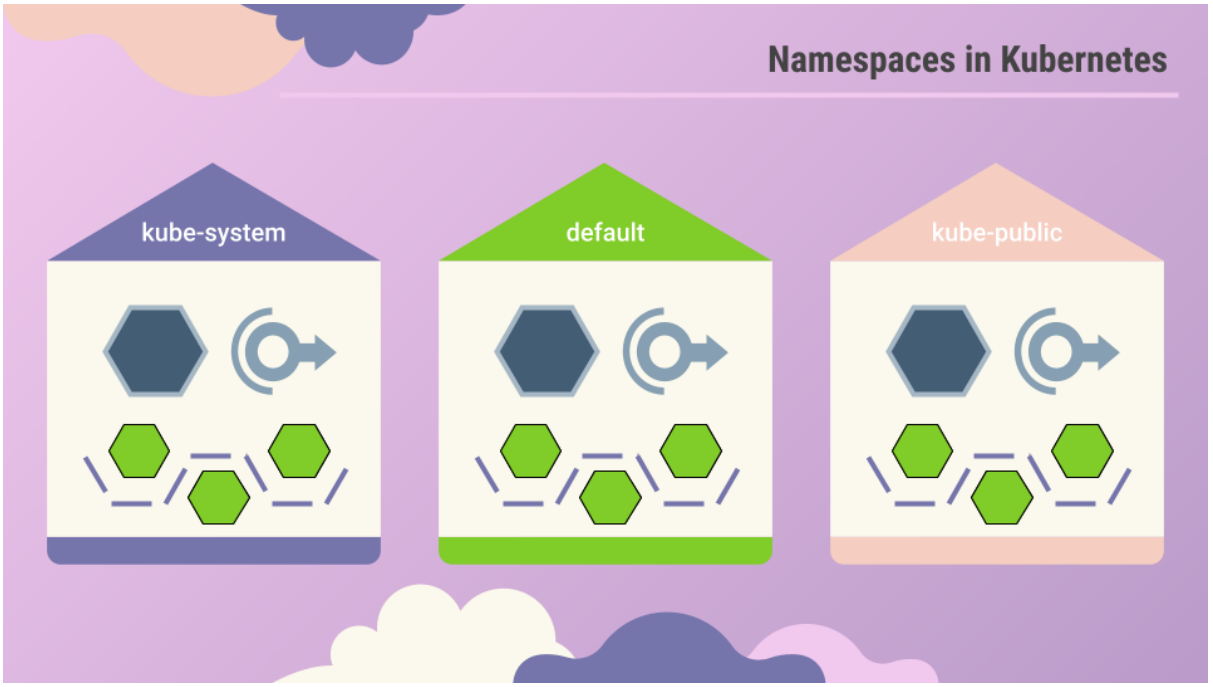
Каждая из этих лабораторий имеет свой собственный набор правил, определяющих, кто чем занимается. Также каждая из них имеет свой собственный набор ресурсов и оборудования, которые они могут используют для своих целей.

Теперь вернемся к Kubernetes: эти здания соответствуют пространствам имен (`namespaces`) в Kubernetes. До сих пор в этом курсе мы создавали такие объекты, которые развертывали PODs и services в нашем кластере.Тем не менее, все, что мы делали, мы делали в пространстве имен.

Все это время мы были внутри одной из лабораторий, и это пространство имен известно как `default` namespace. Это пространство присутствует всегда. Его автоматически создает Kubernetes, когда кластер впервые настраивается.

Это нужно для того, чтобы Kubernetes при создании своих внутренних механизмов, таких как POD-network, DNS-службы и т.п. мог разделить нагрузки. Это разделение предотвращает случайное удаление или модификацию системных компонентов, перенося их в специальное пространство имен - `kube-system`.

Третий namespace, автоматом создающийся при настройке кластера - `kube-public`. Здесь находятся ресурсы, необходимые всем пользователям кластера.



Если твоя среда мала или ты просто изучаешь и играешь с небольшим кластером - тебе не стоит сильно беспокоиться о пространствах имен. Ты можешь можешь продолжить работу в default namespace.

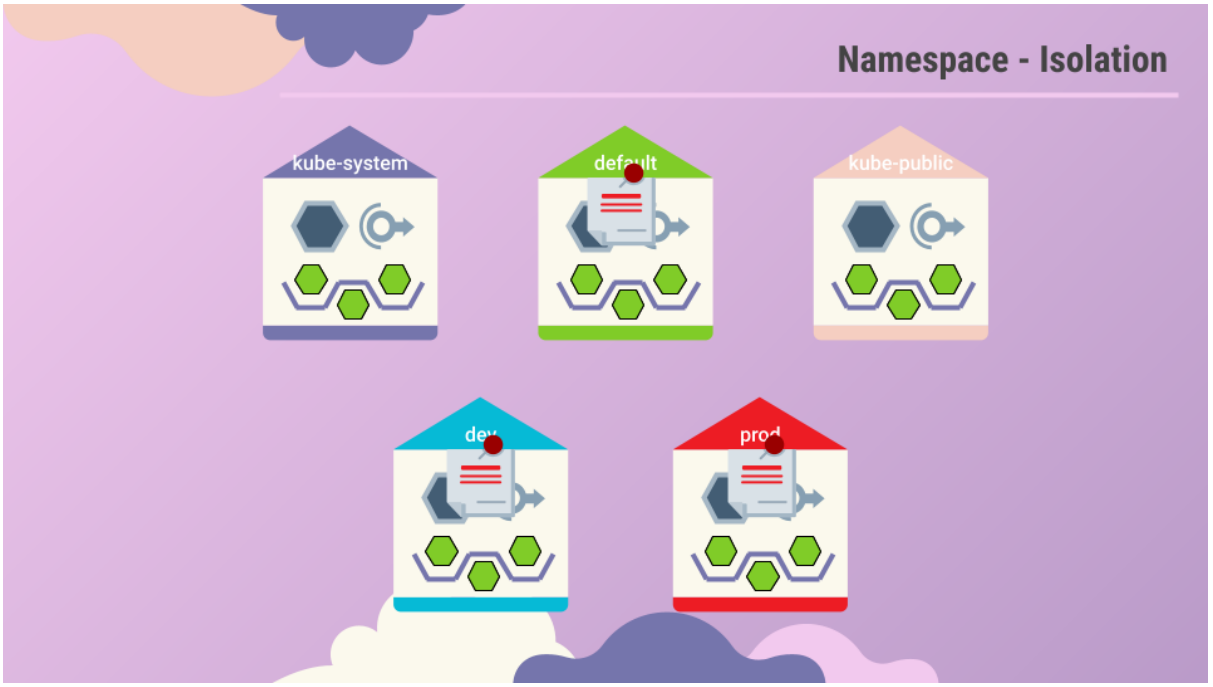
Однако по мере роста использования кластера в корпоративных или производственных сетапах тебе потребуется развести разные нагрузки по разным углам своего кластера. Т.е. тебе нужно будет создать свои собственные пространства имен.

Например, если ты хочешь использовать один и тот же кластер как среду для разработки и одновременно для размещения производственной среды.

Чтобы изолировать ресурсы между ними, ты можешь создать разные namespaces для каждого из них.

Таким образом, работая в среде `dev`, пользователи случайно не изменят ресурсы в продакшене.

Каждое из этих пространств имен может иметь свой собственный набор политик, определяющих, кто и что может делать.



Мы также можем назначить квоту ресурсов для каждого из этих именованных пространств, чтобы каждый namespace имел гарантированный объем ресурсов, но не использовал больше, чем разрешено.

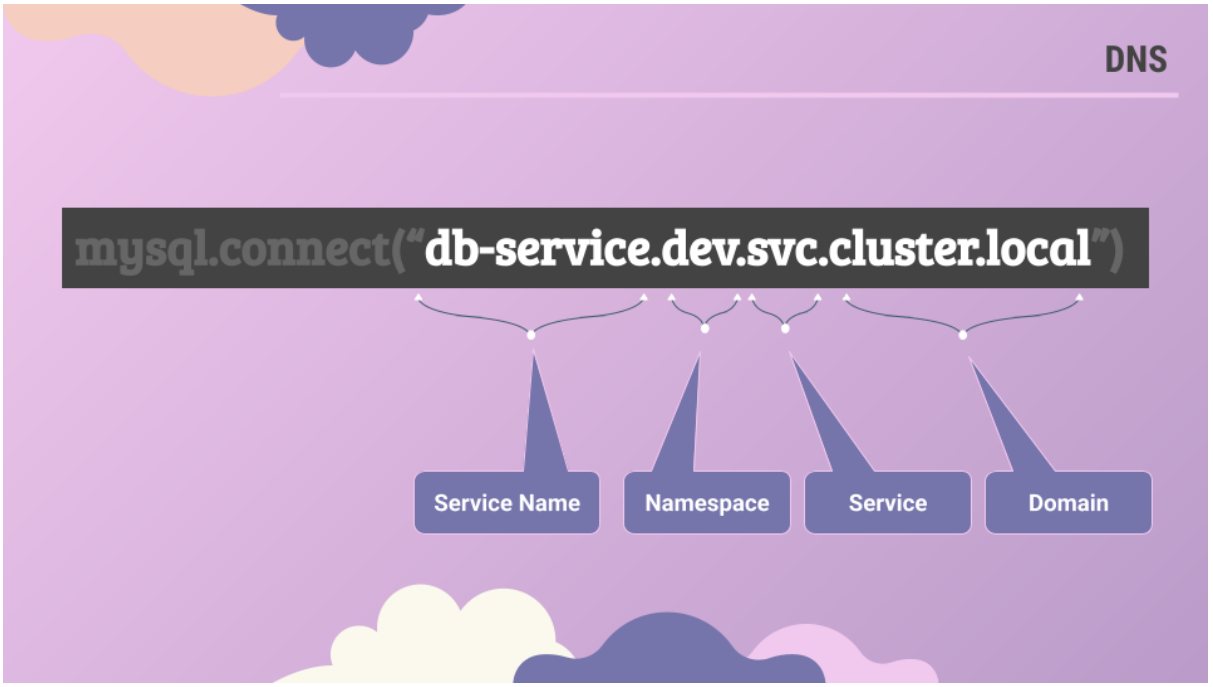
Теперь вернемся в наш default namespace. Ресурсы внутри пространства имен могут обращаться друг к другу и ссылаться друг на друга просто по своим именам.

В этом случае веб-приложение может подключиться к службе db просто используя имя хоста `db-serivce`.

POD веб-приложения также может подключаться к службе в другом пространстве имен.
Для этого нам нужно добавить название этого namespace к имени службы.

Например, веб-POD находится в пространстве имен default и ему требуется подключиться к базе данных в среде разработки в dev namespace.

Используй формат `servicename.namespace.svc.cluster.local`, в котором имя службы будет `db-service`, а namespace - `dev`.

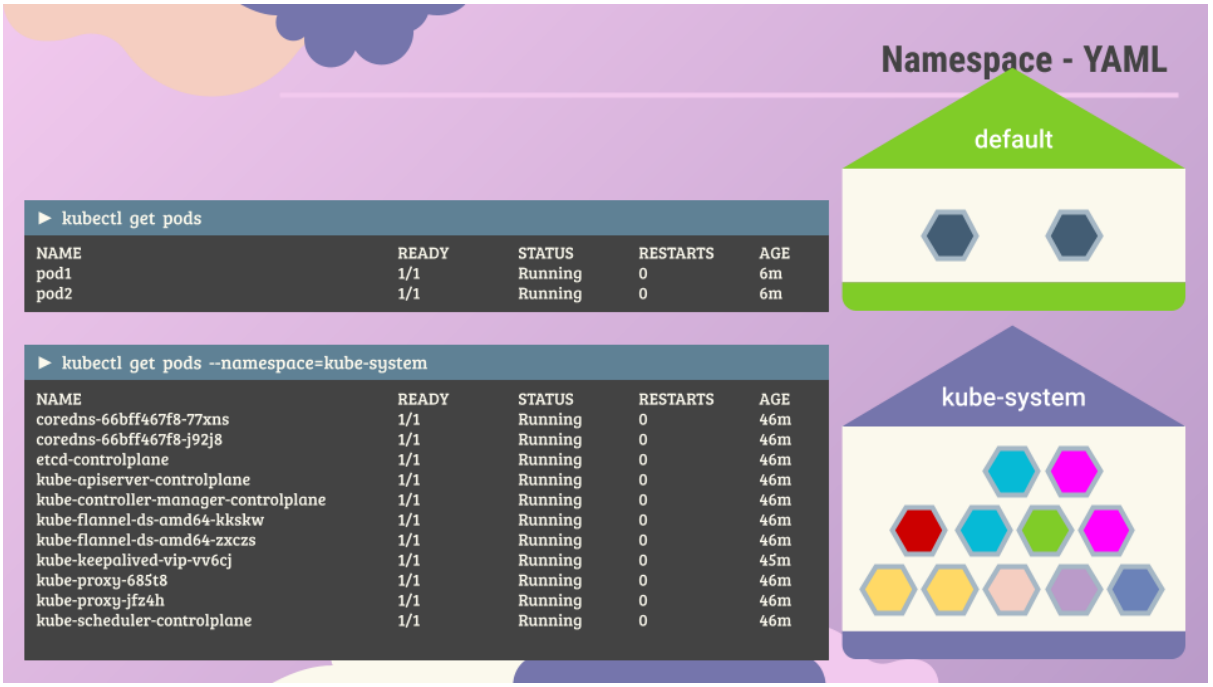


T.e. `db-service.dev.svc.cluster.local`. Мы можем сделать это, потому что при создании службы запись DNS добавляется автоматически в этом формате.

Давай внимательно присмотримся к DNS-имени службы.

Последняя часть `cluster.local` - это доменное имя по умолчанию для кластера Kubernetes. `svc` - это поддомен для службы, за которым следует пространство имен, а затем имя самой службы.

Давай теперь посмотрим на некоторые операционные аспекты пространств имен.



Начнем с команд kubectl.

Например, эта команда используется для вывода списка всех PODs, но она перечисляет только те PODs, которые размещены в пространстве имен default.

Чтобы сделать листинг PODs в другом namespace используй параметр `--namespace` в команде и укажи требуемое пространство имен. В данном случае это kube-system.

Ок, теперь у меня есть файл определения POD. Когда мы создадим POD с помощью этого файла, он будет создан в пространстве имен по умолчанию.

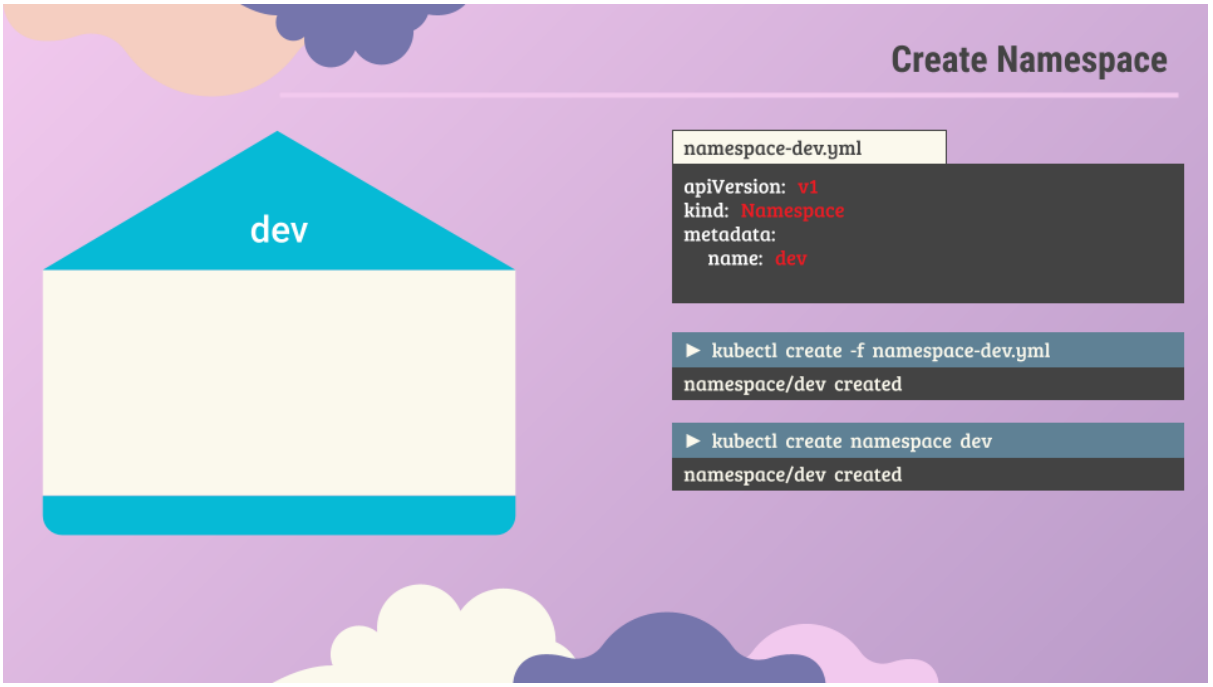
Чтобы создать POD в другом пространстве имен из этого же файла определений используй опцию `--namespace`. Таким образом наш POD будет создан в dev-namespace.

На самом деле вовсе не обязательно указывать пространство имен в командной строке. Гораздо удобнее переместить это определение пространства имен в файл определения POD и положить его в секцию метаданных.

Это хороший способ гарантировать, что твои ресурсы всегда создаются в одном и том же namespace.

Итак, как создать новое пространство имен?

Также как и остальные объекты в Kubernetes.



Используй файл определения пространства имен. Здесь:

- apiVersion: v1
- kind: Namespace
- metadata: name: dev

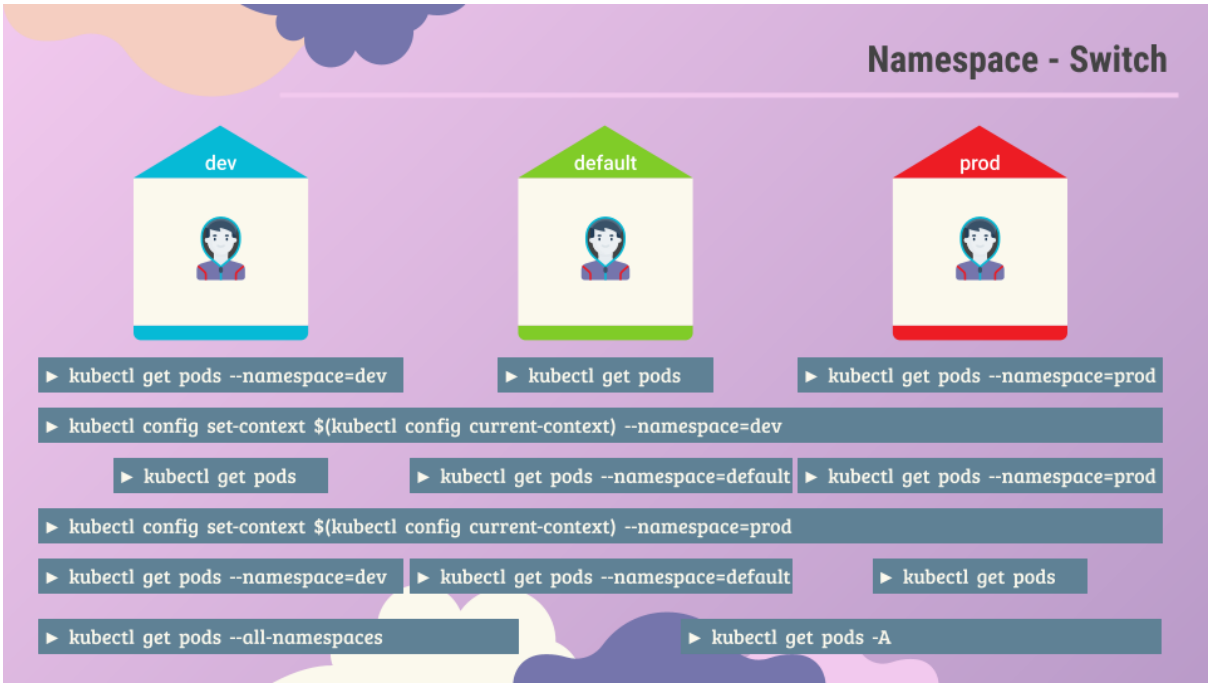
Теперь сделаем `kubectl create -f namespace-dev.yml`, чтобы создать это пространство имен.

Другой способ создать пространство имен - просто запустить команду `kubectl create namespace dev`

Ок, теперь у нас есть три namespaces.
Как же нам сказать kubectl, что мы хотим работать в именно этом из тех трех пространств?

Как мы обсуждали ранее, по умолчанию мы находимся в default namespace, поэтому мы можем видеть ресурсы внутри пространства имен по умолчанию, а с помощью команды `kube control get pods --namespace=dev` и просматривать их и в пространстве имен dev.

Да, мы можем использовать параметр --namespace, но что, если мы хотим навсегда переключиться на пространство имен dev, чтобы нам больше не приходилось каждый раз указывать эту опцию?



В этом случае используй команду `kubectl config set-context`, чтобы установить namespace dev основным для себя.

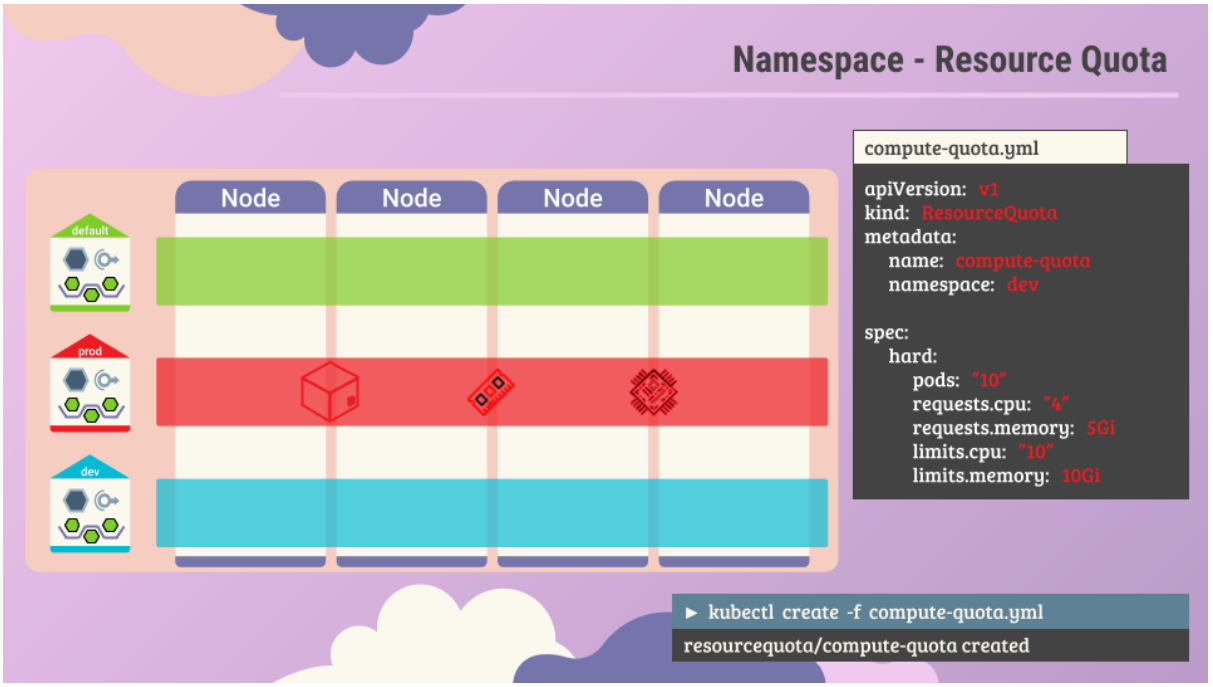
Теперь ты можешь просто запустить команду `kubectl get pods` без параметра namespace, чтобы вывести список PODs в среде dev. Обрати внимание, теперь тебе нужно будет указать параметр для других пространств имен, таких как default или prod. Аналогично можно перескочить и в пространство имен prod.

Наконец, чтобы просмотреть PODs во всех пространствах имен, используй в команде параметр `--all-namespaces` или короче `-A`.

В ней будут перечислены все PODs во всех пространствах имен.

Давай более подробно рассмотрим команду преключения контекста.
Эта команда сначала идентифицирует текущий контекст, а затем устанавливает пространство имен на желаемое для этого текущего контекста.

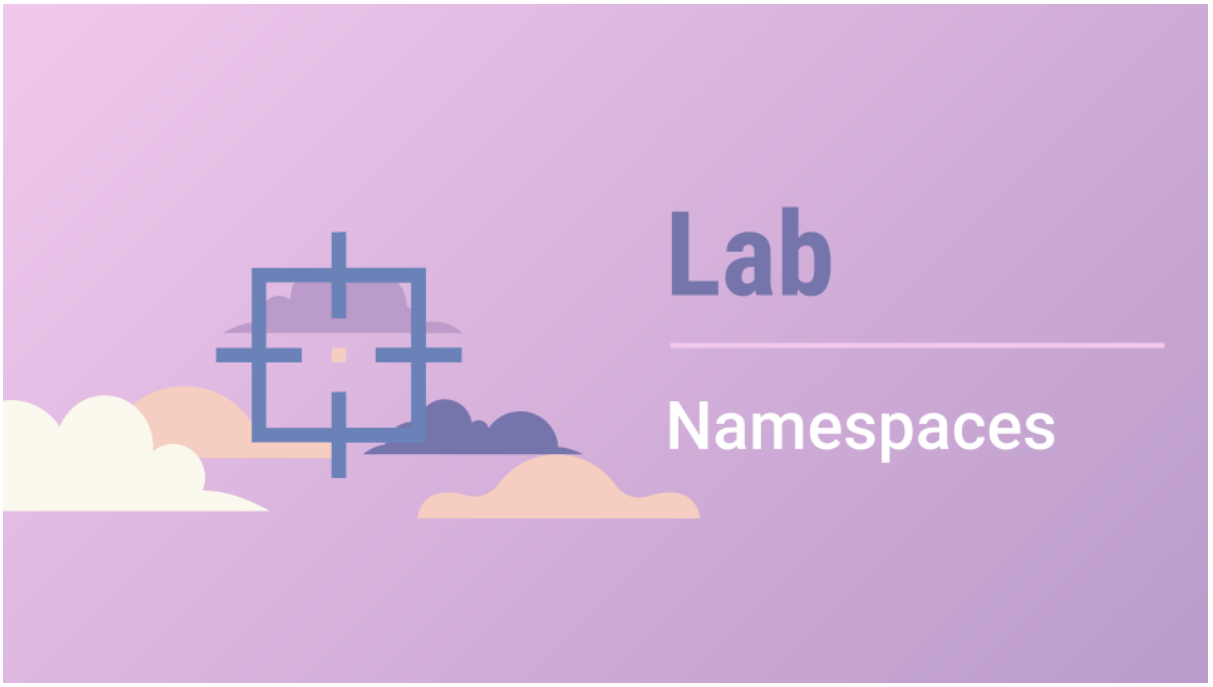
Эти контексты используются для управления несколькими кластерами в нескольких средах из одной системы управления. Это совершенно отдельная тема для обсуждения и требует отдельной лекции, поэтому мы обсудим контекст в другой лекции.



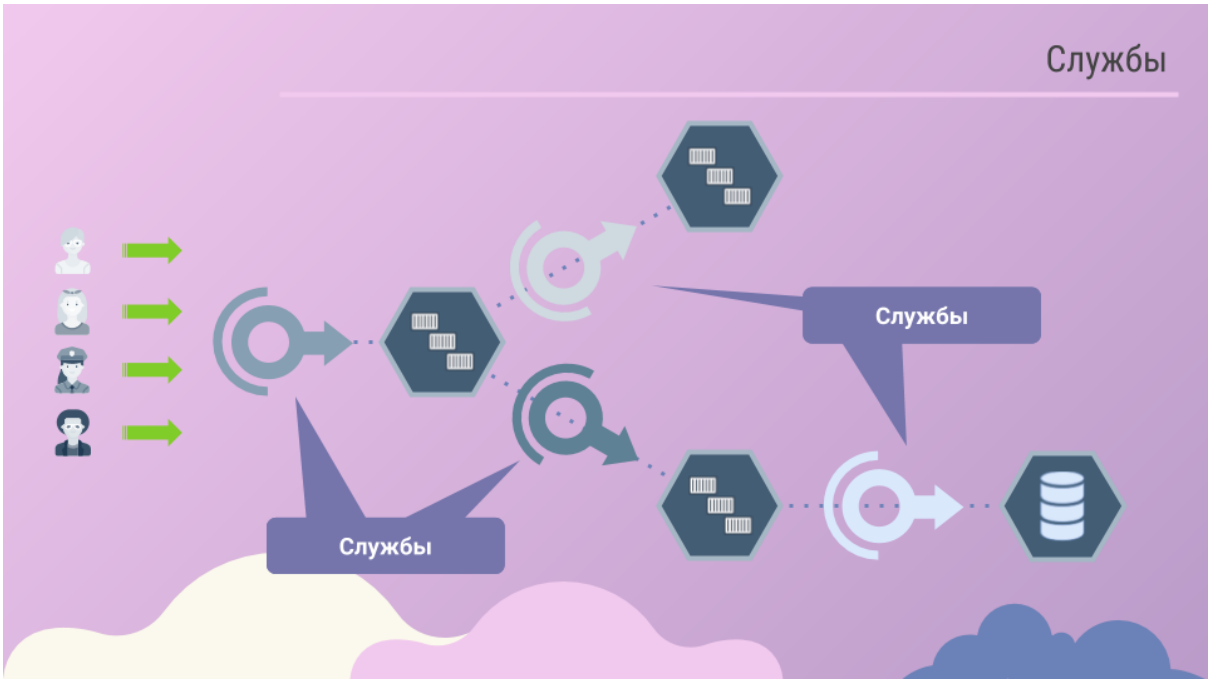
Чтобы ограничить ресурсы в пространстве имен создай квоту ресурсов (`ResourceQuota`).
Для этого начни с подобного этому файла определения. Укажи namespace для которого ты создаешь квоту, а в секции spec пропиши свои ограничения для этого пространства имен. Например здесь 10 PODs - сколько максимально PODs может быть в этом namespace, лимит на память в 10 Гб и 10 CPU.

Применим квоту такой командой, после чего наши ограничения начнут действовать.

Ну вот и все в этой лекции.
Поиграй с пространствами имен в упражнениях, а я жду тебя в следующей лекции.


Lab
Namespaces


Services

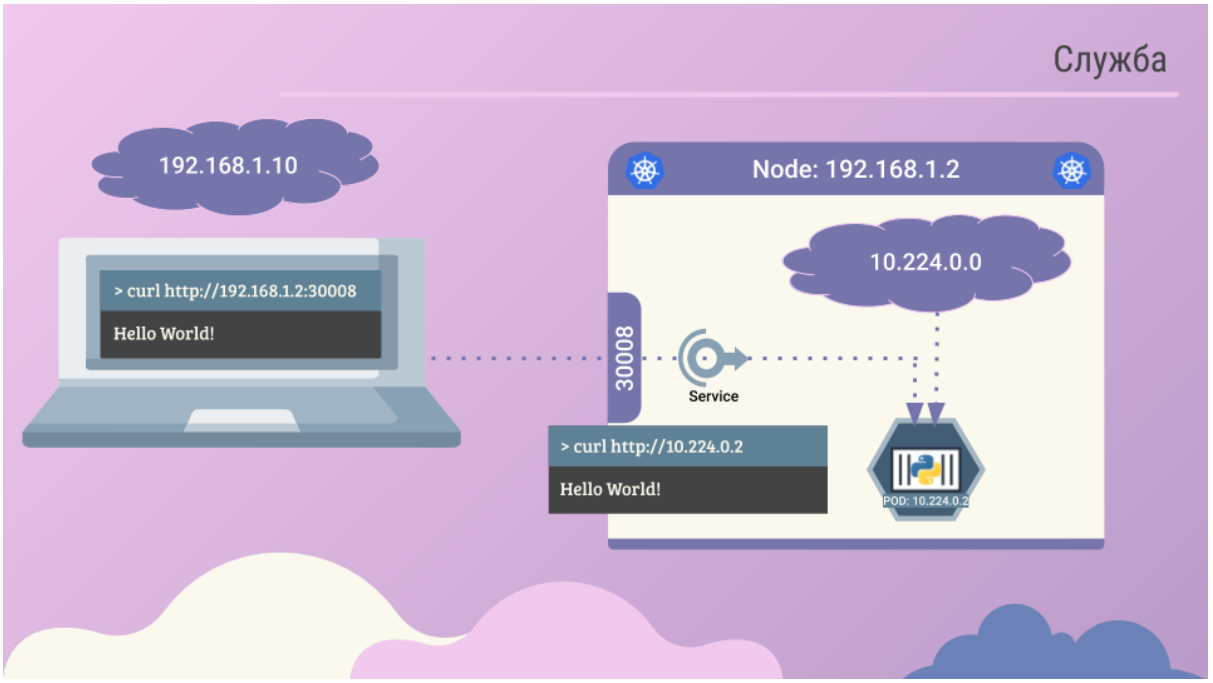


Службы Kubernetes обеспечивают связь между различными компонентами внутри и вне приложения. Они помогает нам соединять приложения как между собой так с конечными пользователями.

Например, в нашем приложении есть группы POD, работающих на различных ярусах приложения, такие как обработка функций фронтенда для обслуживания клиентской части, бекенд-уровне логики приложения и слой данных, ответственный за работу с хранилищем.

Services обеспечивают PODs возможность общаться между разными уровнями приложения.
Они позволяют группе PODs фронтенда быть доступными для пользователей, они дают возможность общаться фронтенду и бекенду, а также устанавливать соединение с внешней базой данных.
Службы уменьшают связанность микросервисов в нашем приложении.

Перед нами пример использования служб.



До сих пор мы говорили о том, что PODs общаются друг с другом через внутреннюю сеть. Давай посмотрим на некоторые другие аспекты нетворкинга в этой лекции. Начнем с внешнего общения.

Итак, мы развернули наш POD с запущенным на нем веб-приложением.
Как мы, как внешний пользователь, получаем доступ к веб-странице?

Прежде всего, давай посмотрим на текущую настройку. Узел Kubernetes имеет IP-адрес 192.168.1.2. Мой ноутбук также находится в той же сети, поэтому у него IP-адрес 192.168.1.10. Внутренняя сеть POD находится в диапазоне 10.224.0.0, а POD имеет IP 10.224.0.2. Ясно, что я не могу проверить связь или получить доступ к POD по адресу 10.224.0.2, так как он находится в отдельной сети.
Как же мне просмотреть веб-страницы?

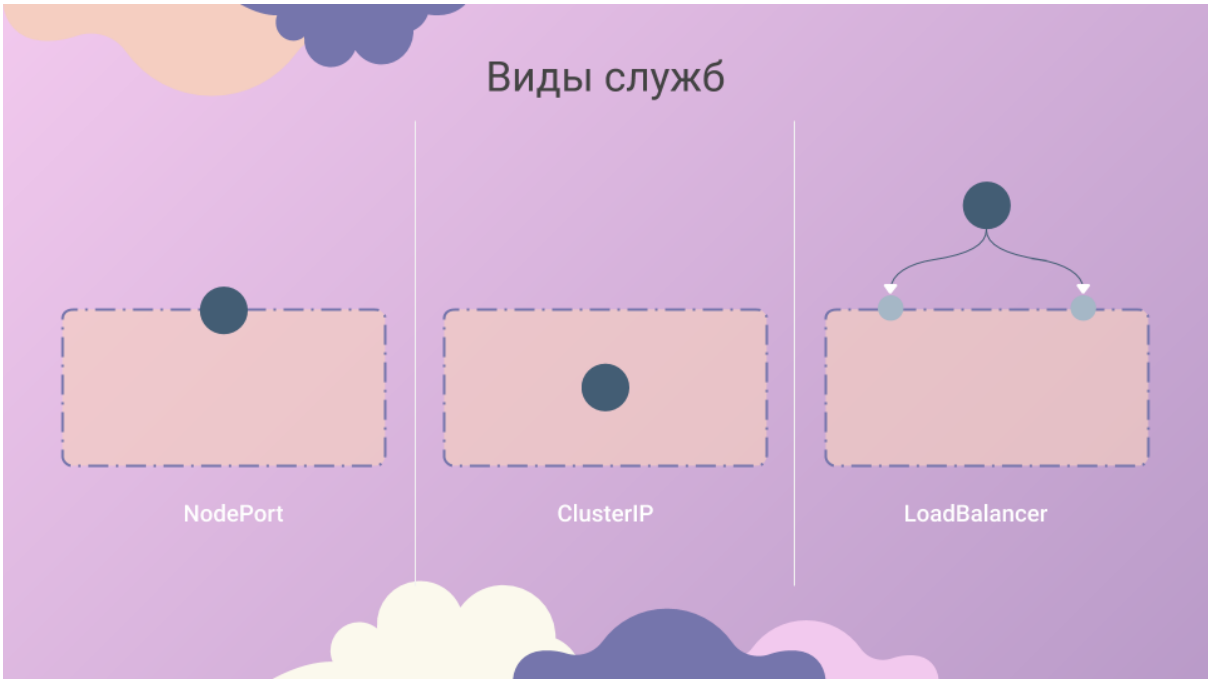
Во-первых, если бы мы подключились по SSH к ноде Kubernetes по адресу 192.168.1.2, то изнутри ноды мы смогли бы получить доступ к веб-странице POD, выполнив curl, или, если узел имеет графический интерфейс, мы могли бы запустить браузер и увидеть веб-страницу в браузере по адресу http://10.224.0.2.

Но это изнутри ноды Kubernetes, и это не то, что мне действительно нужно. Я хочу иметь доступ к веб-серверу со своего ноутбука, не подключаясь к узлу по SSH, а просто обращаясь к IP-адресу узла Kubernetes.

Итак, нам нужно что-то посередине, чтобы помочь нам сопоставить запросы к ноде с моего ноутбука к POD, на котором запущен веб-контейнер. Вот где в игру вступает объект Kubernetes service.
Service - это такой же объект, как POD, Replicaset или Deployment, с которыми мы работали раньше.

Один из вариантов использования службы - прослушивание порта на ноде и пересылка запросов с этого порта на порт в POD, на котором запущено веб-приложение.

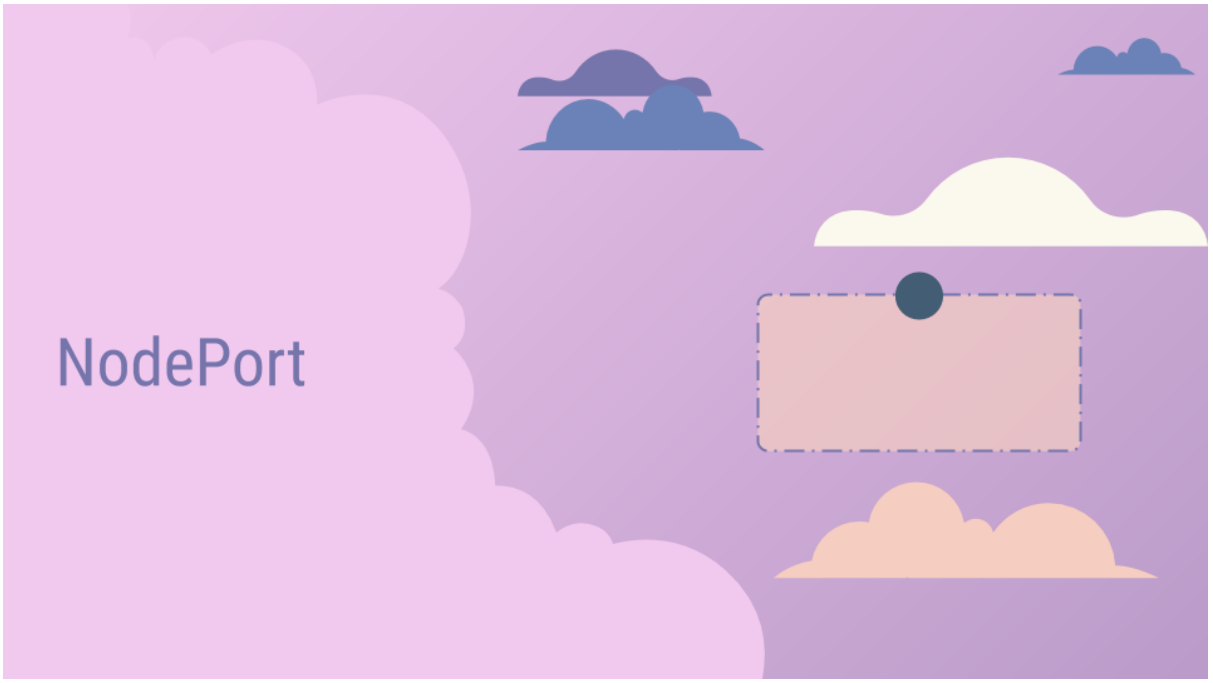
Этот тип service известен как служба NodePort, поскольку служба прослушивает порт на ноде и перенаправляет запросы на POD. Доступны и другие виды services, которые мы сейчас обсудим.

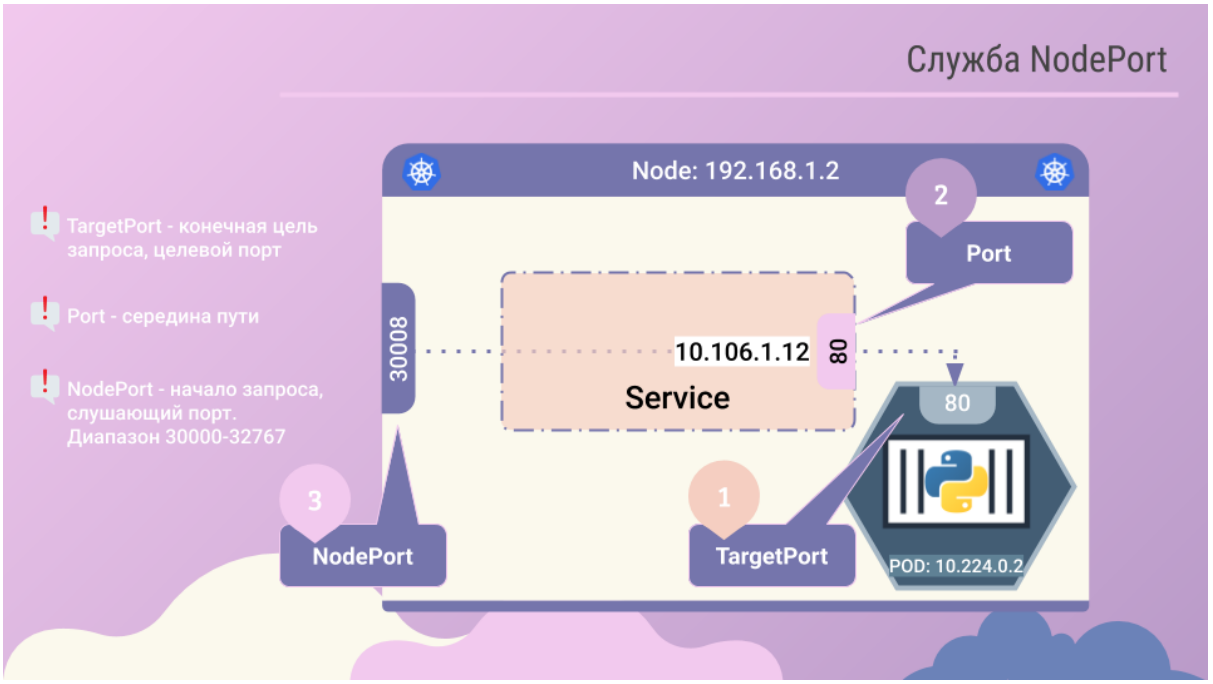


Первый - это то, что мы уже обсуждали NodePort, когда служба делает доступным внутренний POD через порт на узле.

Второй - ClusterIP, и в этом случае служба создает виртуальный IP-адрес внутри кластера, чтобы обеспечить связь между различными службами, такими как набор фронтендов и бэкендов.

Третий тип - это LoadBalancer, он предоставляет балансировщик нагрузки для нашего сервиса у поддерживаемых облачных провайдеров.
Хорошим примером этого может быть распределение нагрузки между разными веб-серверами в ярусе фронтенда.





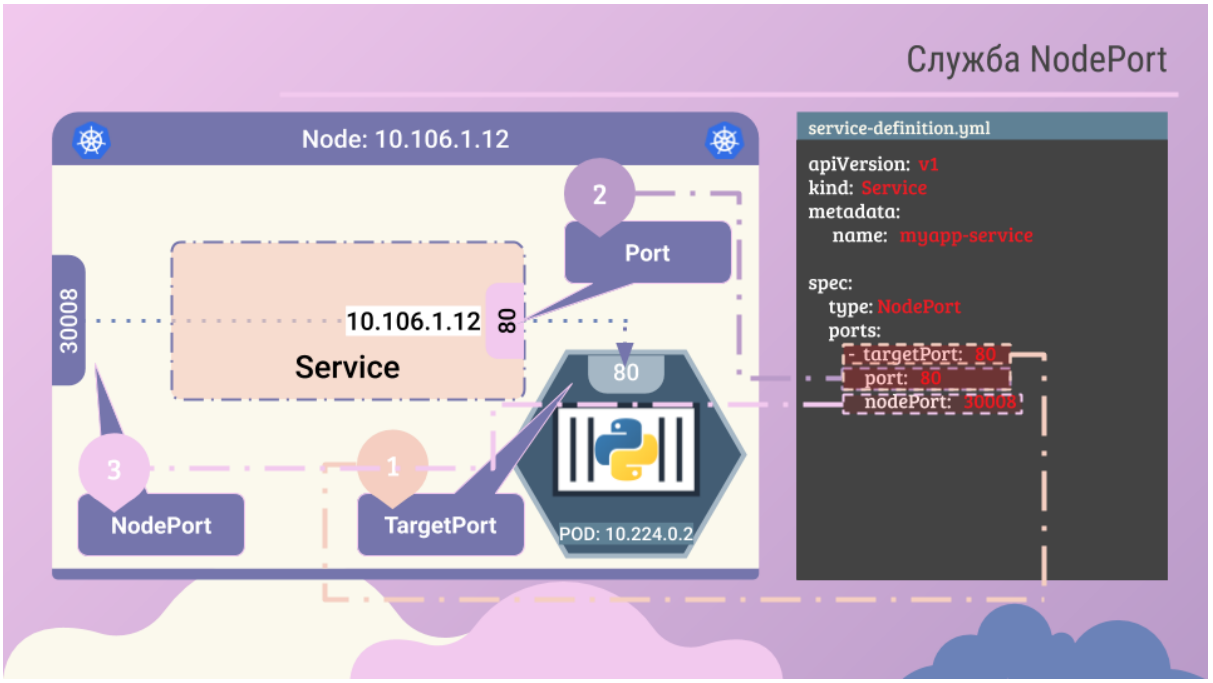
Несколько слайдов назад мы обсуждали внешний доступ к приложению. Мы сказали, что служба может помочь нам, сопоставив порт на ноде с портом в POD.
Давай подробнее рассмотрим службу. В ней задействовано 3 порта.

Порт в POD, на котором работает фактический веб-сервер, это порт 80. Он называется targetPort, потому что именно на него служба и перенаправляет запросы.

Второй порт - это порт самой службы. Его называют просто port. Поначалу все путаются в названиях. Помни, что эти определения относятся к объекту Service. А для простоты представь себе службу как виртуальный сервер внутри ноды.

И у этого сервера есть собственный IP-адрес внутри кластера. И называется он Cluster Ip службы.

И, наконец, у нас есть порт на самой ноде, который мы используем для внешнего доступа к веб-серверу. Его название NodePort. Как видишь, это 30008. Это потому, что NodePorts могут находиться только в допустимом диапазоне от 30000 до 32767.



Давай теперь посмотрим, как создать Service. Так же, как мы создали Deployment, ReplicaSet или Pod, мы будем использовать файл определения для создания службы. Базовая структура файла остается прежней. Как и раньше, у нас есть разделы apiVersion, kind, metadata и spec.

Версия apiVersion будет v1. Kind конечно Service. У метаданных будет имя, и это будет имя службы. У него могут быть labels, но пока они нам не нужны. Далее идет spec и, как всегда, это самая важная часть файла, поскольку здесь мы будем определять фактические службы, и это часть файла определения, которая сильно различается между разными типами объектов Kubernetes.

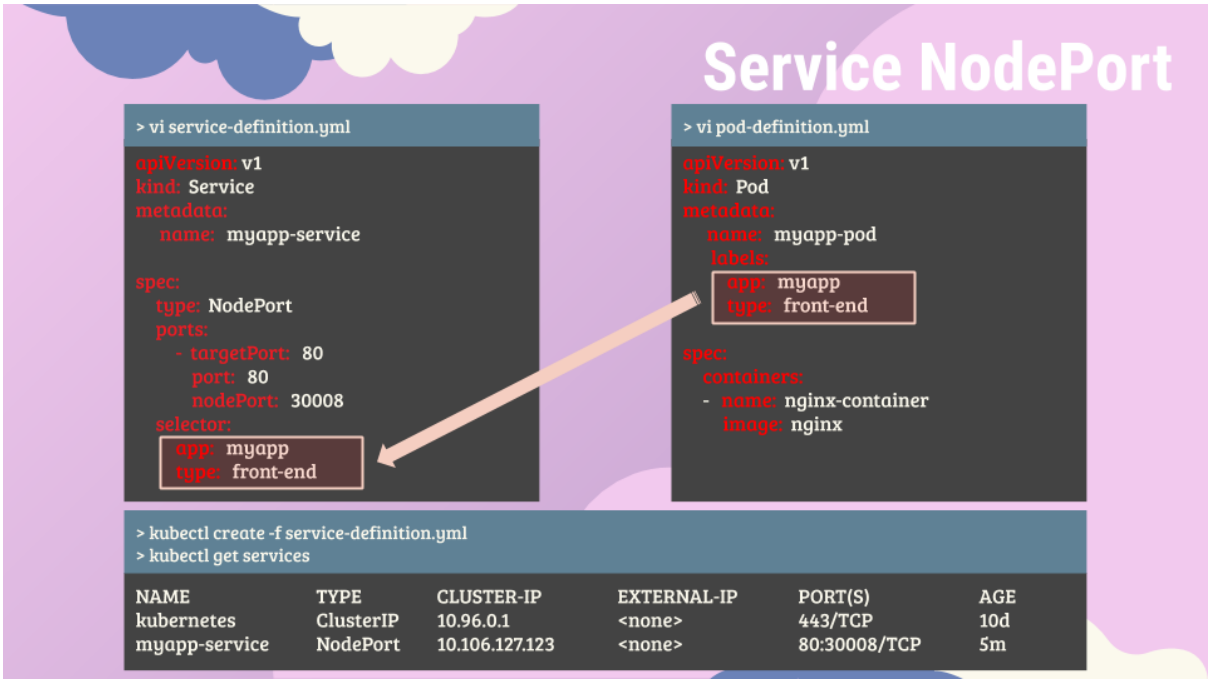
В разделе спецификации Service у нас есть тип и порты. Тип относится к типу создаваемой нами службы.
Как обсуждалось ранее, это может быть ClusterIP, NodePort или LoadBalancer.
В этом случае, поскольку мы создаем NodePort, мы установим его как NodePort.

Следующая часть спецификации - это ports. Здесь мы вводим информацию о том, что мы обсуждали, в левой части экрана.
Первый тип порта - это targetPort, для которого мы установим 80. Следующим будет просто port, который является портом самой службы, и мы также установим его на

80. Третий - это NodePort, который мы установим на 30008 или любое число в допустимом диапазоне.

Помни, что из них единственным обязательным полем является port. Если ты не укажешь targetPort, предполагается, что он совпадает с port, а если ты не предоставишь nodePort, то автоматически выделяется свободный порт в допустимом диапазоне от 30000 до 32767.

Также обрати внимание, что ports - это массив.
Видишь "минус" под разделом ports, он указывает, что это первый элемент в массиве. Ты можешь иметь несколько таких сопоставлений портов в одной службе.



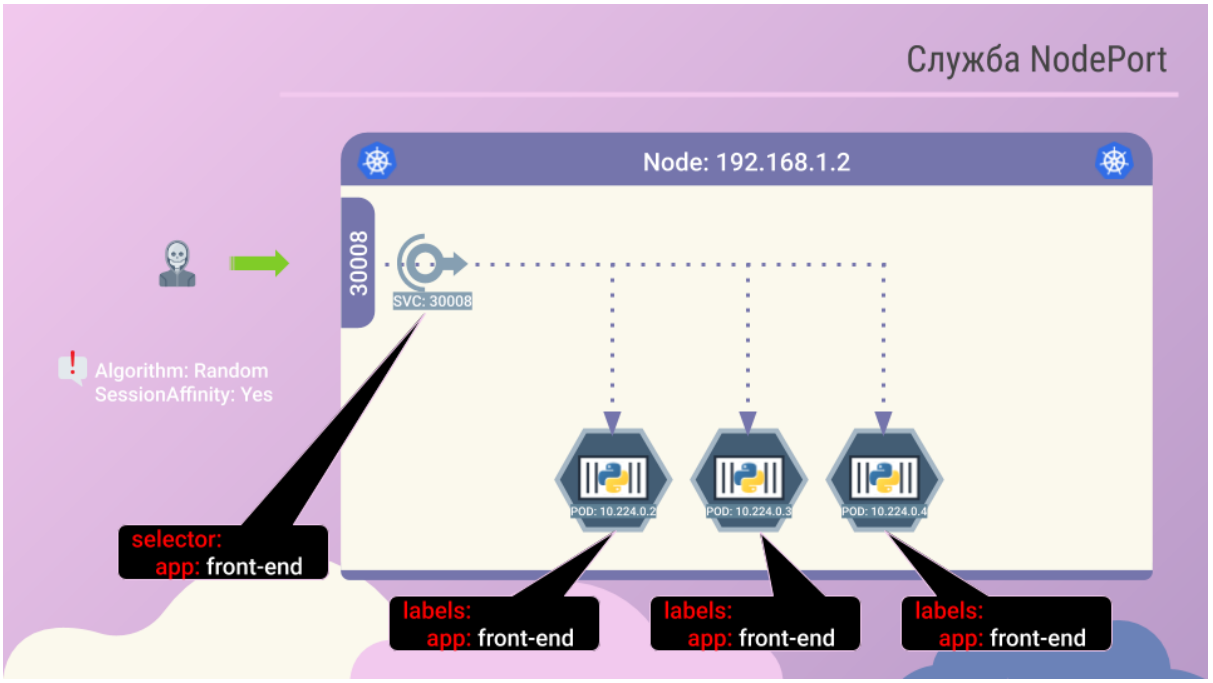
Итак, у нас есть вся информация, но чего-то действительно не хватает. В файле определения нет ничего, что связывало бы службу с POD. Мы просто указали targetPort, но не объяснили службе на каком POD искать этот порт. Могут быть сотни других PODs с веб-службами, работающими на порту 80. Итак, как нам это сделать?

Мы будем использовать метки и селекторы, чтобы связать их вместе. Это техника, которую ты очень часто будешь встречать в Kubernetes. Мы делали так ранее с ReplicaSets.

POD был создан с label. Нам нужно внести эту метку в файл определения Service. Итак, у нас есть новое свойство в разделе спецификации - selector. Под селектором укажи список меток для идентификации POD. Для этого обратись к файлу определения, который использовался для создания POD. Скопируй labels из манифеста POD и помести его в секцию selector. Это свяжет службу и POD. После этого запусти команду kubectl create указав файл определения службы. Эта команда создаст объект Service.

Чтобы увидеть созданную службу, запусти команду kubectl get services, ты увидишь все службы, их IP-адреса и сопоставленные порты.

Созданный нами тип - NodePort, а назначенный порт на узле - 30008.
Теперь мы можем использовать этот порт для доступа к веб-приложению с помощью curl или браузера.

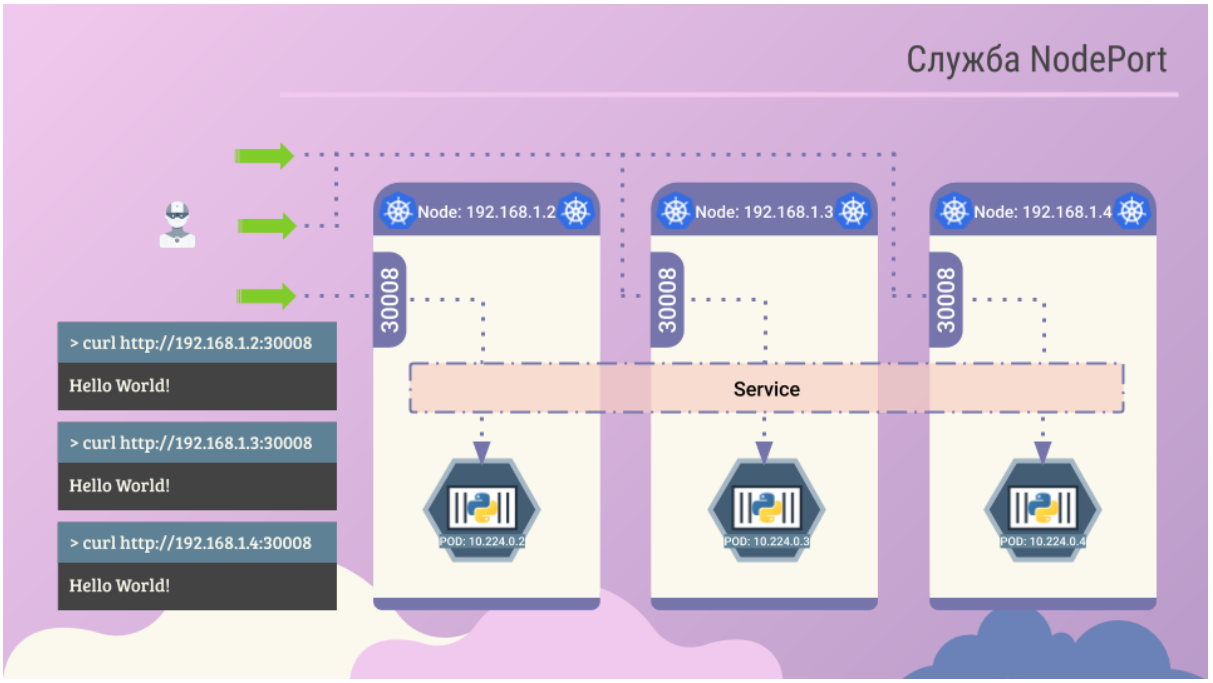


До сих пор мы говорили о службе, привязанной к одному POD. Но это не всегда так. Что нам делать, когда PODs несколько?

В производственной среде у тебя есть несколько экземпляров веб-приложения, работающих для обеспечения высокой доступности и балансировки нагрузки. В этом случае мы имеем несколько похожих POD, на которых работает наше приложение.

Все они имеют одинаковые метки с ключем app и значением front-end. Эта же метка используется в качестве селектора при создании службы. Таким образом, когда служба создается, она ищет соответствующие POD с метками и находит 3 из них. Затем служба автоматически выбирает все 3 PODs в качестве конечных точек для пересылки внешних запросов, поступающих от пользователя.
Для этого не нужно выполнять никаких дополнительных настроек.

Если тебе интересно, какой алгоритм служба использует для балансировки нагрузки - используется случайный алгоритм. Таким образом, Service действует как встроенный балансировщик нагрузки для распределения нагрузки между различными PODs.

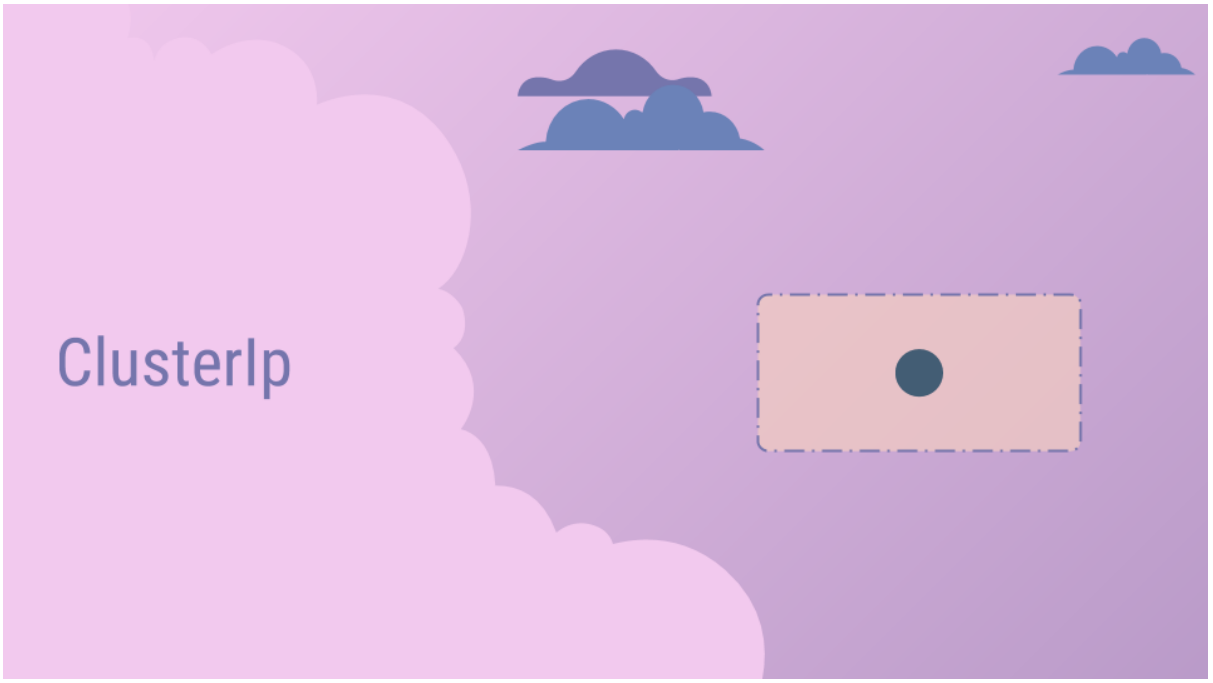


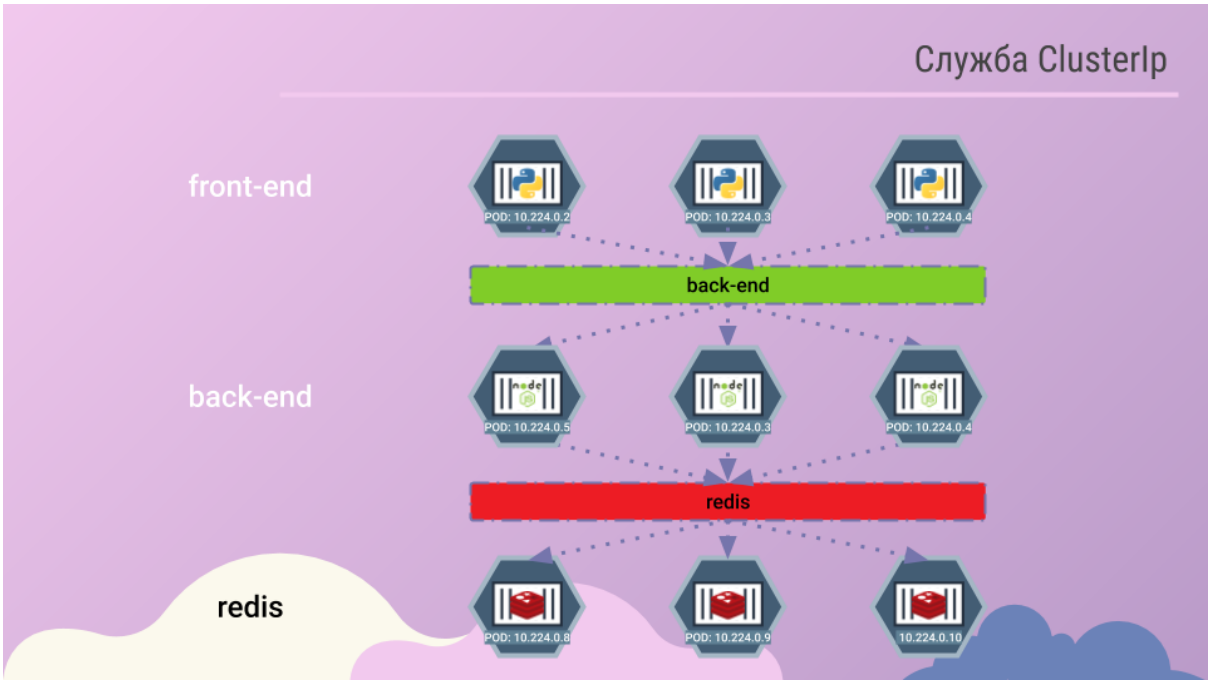
И, наконец, давай посмотрим, что происходит, когда PODs распределяются по нескольким нодам. В этом случае у нас есть веб-приложение на PODs в разных узлах кластера.

При создании службы, у нас нет необходимости делать ЛЮБУЮ дополнительную конфигурацию. Kubernetes создает службу, охватывающую все узлы в кластере, и ЖЕСТКО сопоставляет целевой порт с NodePort на всех узлах кластера. Таким образом, ты можешь получить доступ к своему приложению, используя IP-адрес любой ноды в кластере и используя тот же номер порта, который в данном случае равен 30008.

Подводя итог - в ЛЮБОМ случае, если это будет один POD на одной ноде, несколько PODs на одной ноде, несколько PODs на нескольких узлах, служба создается точно так же, без необходимости выполнять какие-либо дополнительные действия во время создания.

Когда PODs удаляются или добавляются, служба автоматически обновляется, что делает её очень гибкой и адаптивной. После создания тебе, как правило, не нужно вносить никаких дополнительных изменений в конфигурацию.





Фулстек веб-приложение обычно имеет различные типы PODs, на которых размещаются разные уровни функционала приложения. У нас может быть группа PODs, на которых работает фронтенд, другой набор PODs для бэкенда, плюс PODs, на котором запущена кеширующая база, например Redis, а также группа для постоянной базы данных, вроде MySQL и т.д. Фронтенд должен взаимодействовать с бэкендом, а тот в свою очередь со слоями работы с данными: кэширующим и слоем постоянного хранения. Как правильно установить связь между этими группами PODs?

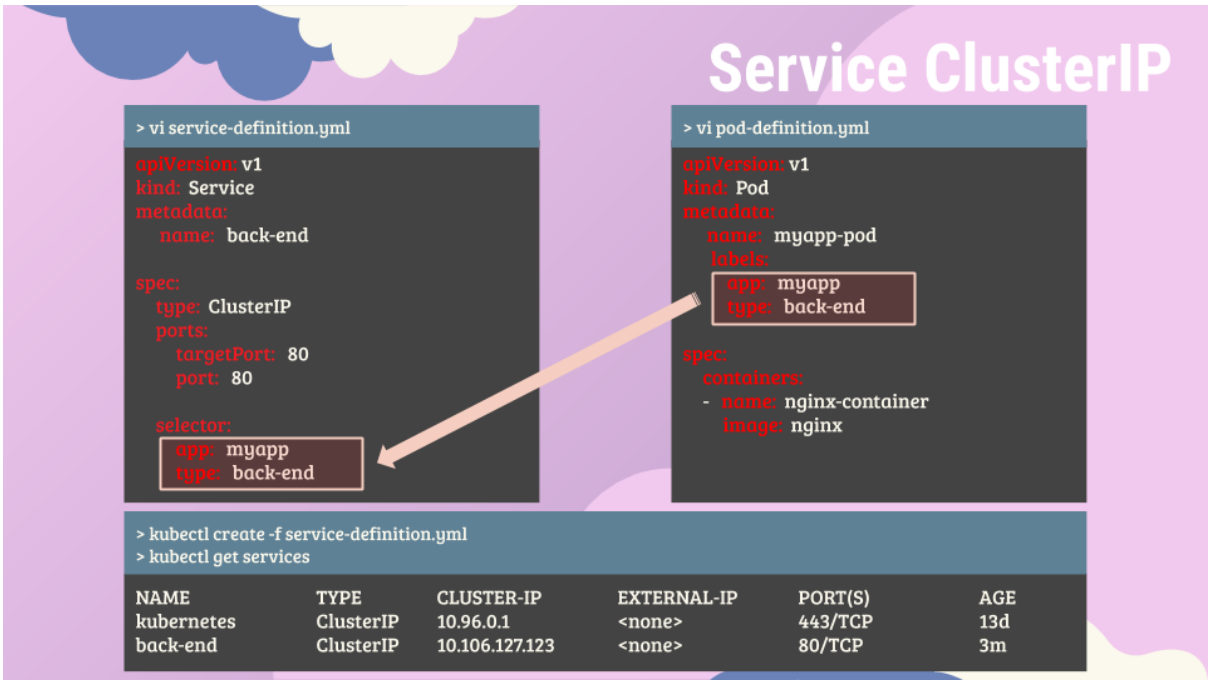
Всем PODs назначен IP-адрес, как мы видим на экране. Но эти IP-адреса, как мы знаем, не статичны, PODs могут отключиться в любое время, также постоянно

создаются новые PODs, поэтому мы НЕ МОЖЕМ полагаться на IP-адреса внутренней сети для связи в приложении. Допустим, первому POD фронтенда на 10.224.0.3 необходимо подключиться к бэкенду?
К какому из трех экземпляров бэкенда он обратится и как будет принято это решение?

Службы Kubernetes могут помочь нам сгруппировать эти PODы вместе и предоставить единый интерфейс для доступа к группе PODs. Например, служба, созданная для бэкенда, поможет сгруппировать все внутренние PODs вместе и предоставит единую точку другим PODs для доступа к ним. Запросы перенаправляются на один из POD в рамках службы случайным образом.

Точно так же создадим дополнительную службу для Redis и разрешим бэкенду обращаться к Redis через эту службу. Это позволяет нам легко и эффективно развертывать приложение на основе микросервисов в кластере Kubernetes.

Каждый уровень теперь можно масштабировать или перемещать по мере необходимости, не влияя на обмен данными между различными службами. Каждой службе назначаются IP-адрес и имя внутри кластера, и это имя должно использоваться другими PODs для доступа к службе. Такой тип службы называется ClusterIP.

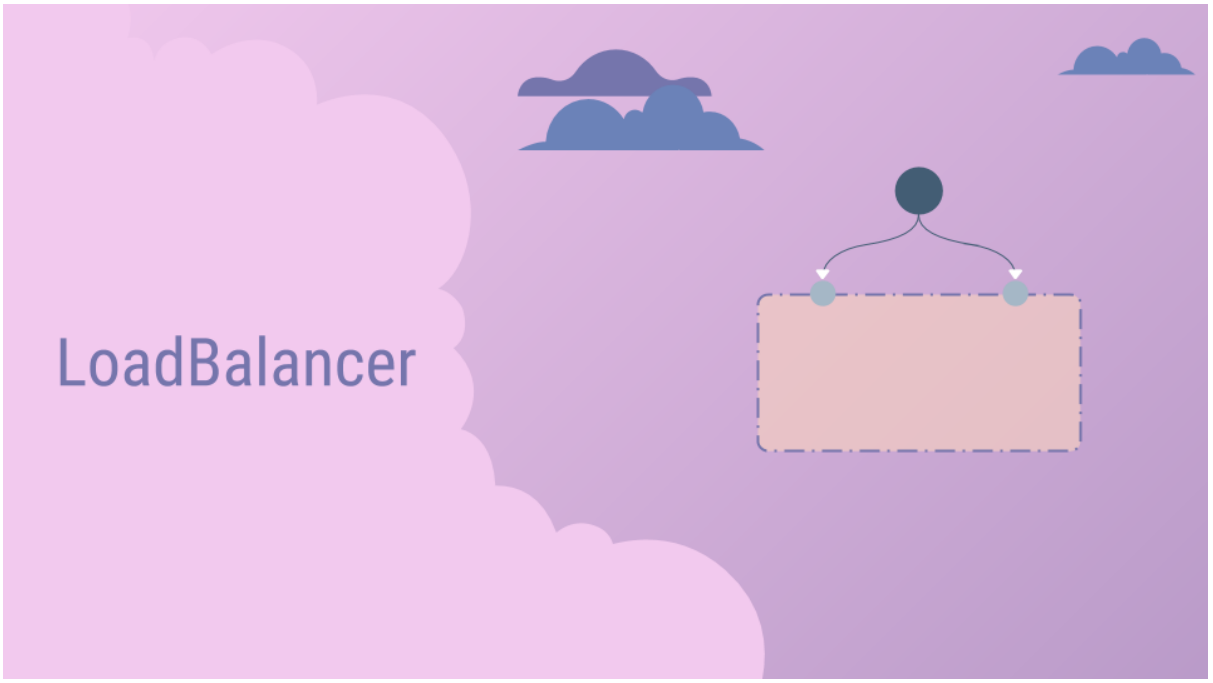


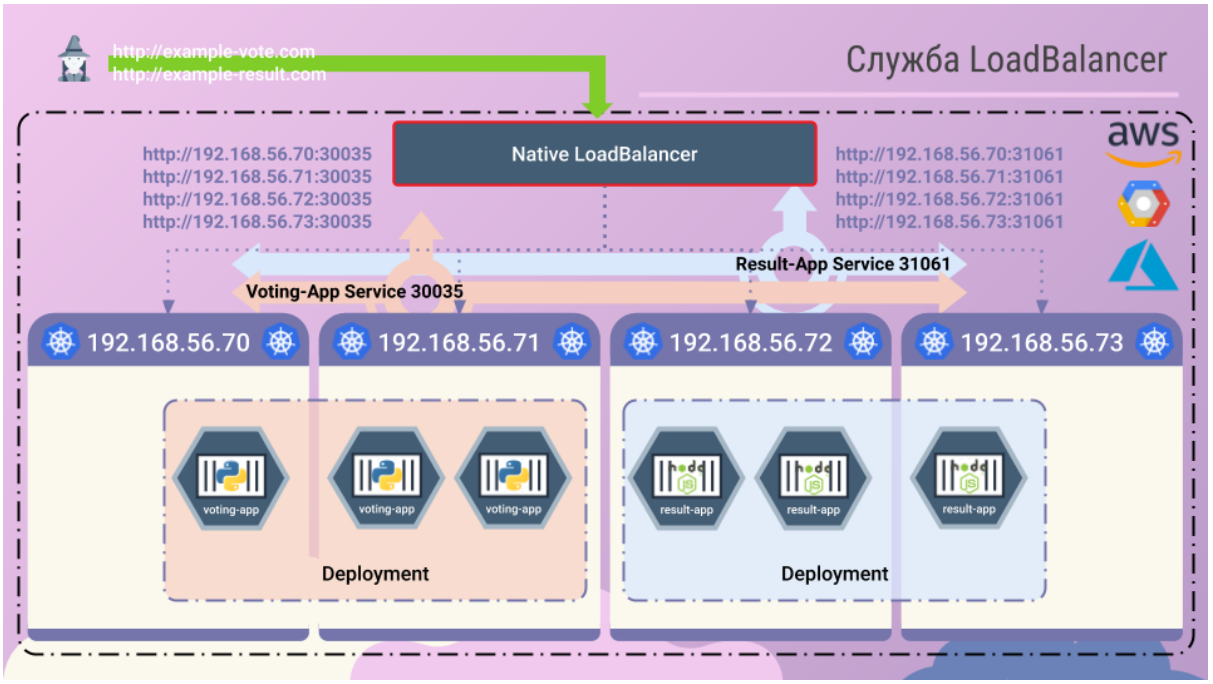
Для создания такой службы, как всегда, используем файл определения.
В нем сначала используем основные значения, а именно: apiVersion, kind, metadata и spec.
ApiVersion - v1, kind - Service. Дадим имя нашей службе, мы назовем ее back-end. В спецификации у нас есть тип (type) и порты(ports). Type будет ClusterIP. Фактически, ClusterIP является типом по умолчанию, поэтому, даже если ты не указал его,

Kubernetes автоматически будет считать что это ClusterIP. В секции ports будет targetPort и port. Целевой порт (targetPort) - это порт, на который открыт в бэкенд части приложения, в данном случае это 80. Порт (port) - порт самой службы. Он тоже слушает тоже 80. Чтобы связать службу с набором PODs, мы используем selector. Возьмем манифест POD, скопируем из него labels и положим их в секцию selector. Этого достаточно.

Теперь мы можем создать службу с помощью команды kubectl create, а затем проверить ее статус с помощью команды kubectl get services.

Служба и PODs за ней будут доступны другим PODs кластера используя имя службы или ее ClusterIP.





Мы видели службу NodePort, которая помогает нам сделать внешнее приложение доступным через порт на рабочих узлах. Давай сосредоточимся на наших тестовых фронтенд-приложениях, одно из которых собирает голоса пользователей, а другое публикует результаты.

Мы уже знаем, что порты приложений размещены на рабочих нодах кластера. Допустим, у нас есть кластер с четырьмя узлами, и чтобы сделать приложения доступными для внешних пользователей, мы создаем службы типа NodePort. Службы с типом NodePort помогают в получении трафика на порты нод и их перенаправлении в порты PODs.

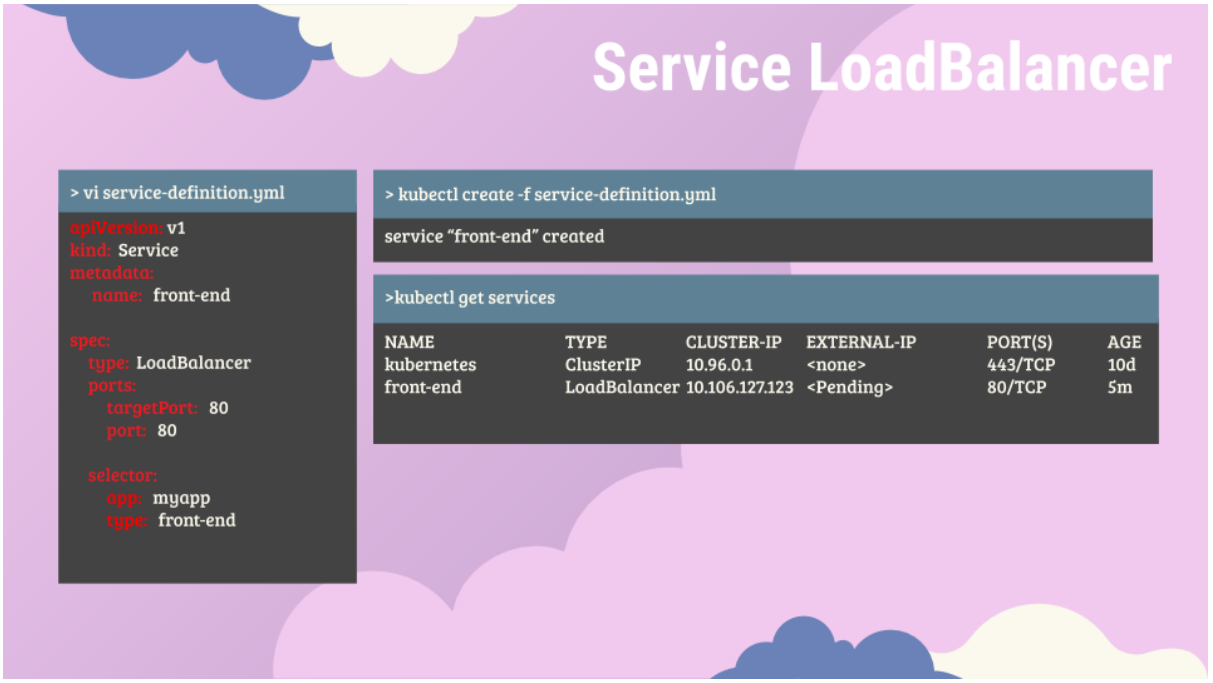
Какой URL-адрес мы дадим своим конечным пользователям для доступа к приложениям?

Пользователь может получить доступ к любому из этих двух приложений используя IP-адрес любого из узлов и верхне-диапазонный порт вроде 30000, на котором слушает служба. В этом случае мы имеем четыре комбинации IP-адресов и портов для приложения-голосования и четыре комбинации IP-адресов и портов для приложения-результата.

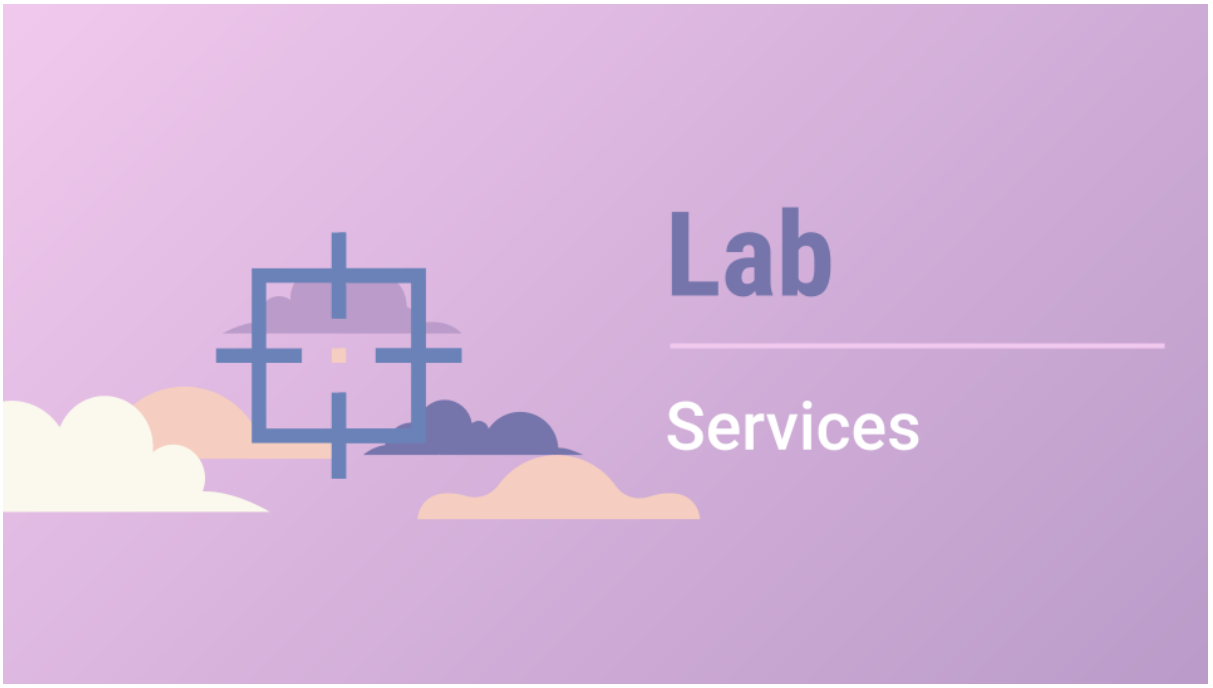
Имей в виду, что если экземпляры приложения выполняются не на всех нодах, служба будет слушать на всех узлах кластера. Скажем, PODs приложения-голосования развернуты только на узлах с IP 70 и 71, но они все равно будут доступны на портах всех нод в кластере. Так настраивается Service.

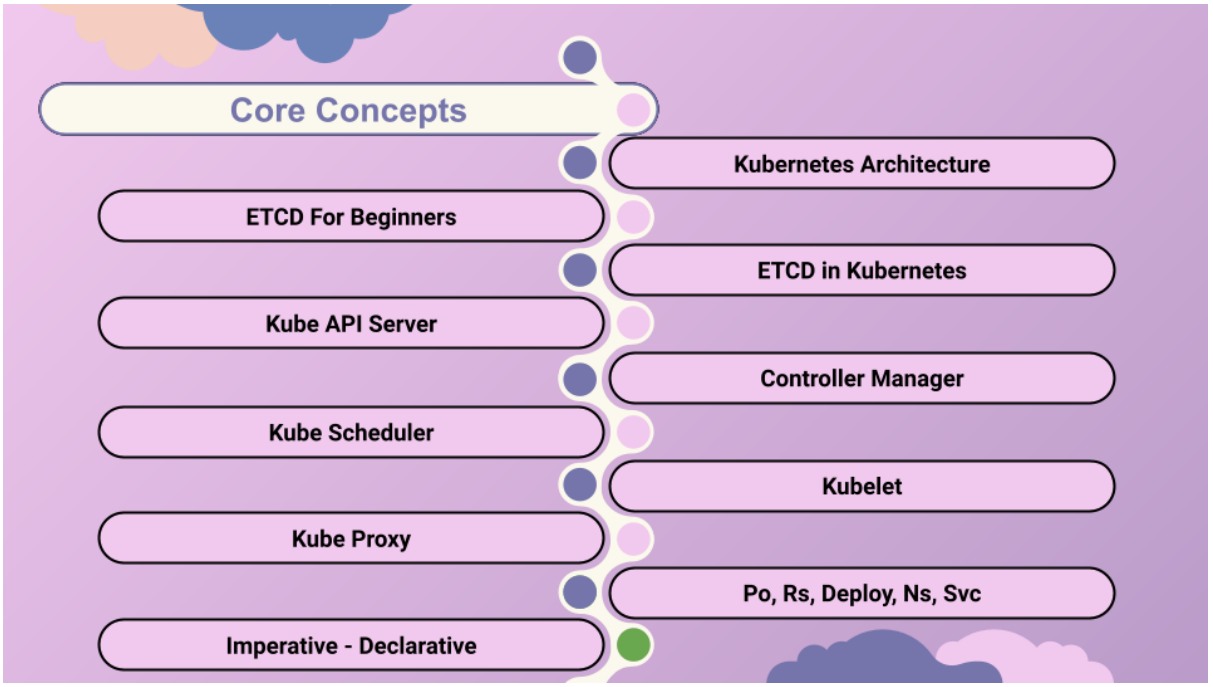
Ты можешь поделиться этими URL для доступа к приложению, но конечные пользователи вряд ли этому обрадуются. Для доступа к приложению им нужен единый URL, например, example-vote.com или example-result.com. Так как же этого добиться?

Один из способов достигнуть этого - создать новую виртуальную машину для балансировки нагрузки, а также установить и настроить на ней подходящий балансировщик нагрузки, например HAProxy, Nginx и так далее. Затем настроить балансировщик нагрузки для маршрутизации трафика на целевые узлы. Вобщем, настройка всей этой системы внешней балансировки, а также ее управление и обслуживание может оказаться утомительной задачей. Однако, если бы мы работали на клауд-платформе, такой как Google Cloud, AWS или Azure, то могли бы использовать их родной балансировщик нагрузки. Kubernetes поддерживает интеграцию с балансировщиками нагрузки определенных облачных провайдеров и умеет их настраивать для нас. Все, что нам нужно сделать, это установить тип службы для фронтенд служб на LoadBalancer вместо NodePort. Не забудь, что это работает с большинством, но не со всеми облачными платформами. Большие вендоры: AWS, GCP, Azure и Alibaba поддерживаются, в остальных проверь документацию.



Если ты установишь тип службы LoadBalancer в неподдерживаемой среде, такой как VirtualBox, или в любой другой не облачной, это будет иметь тот же эффект, что и установка его в NodePort, где службы доступны на верхне-диапазонном порту на нодах.



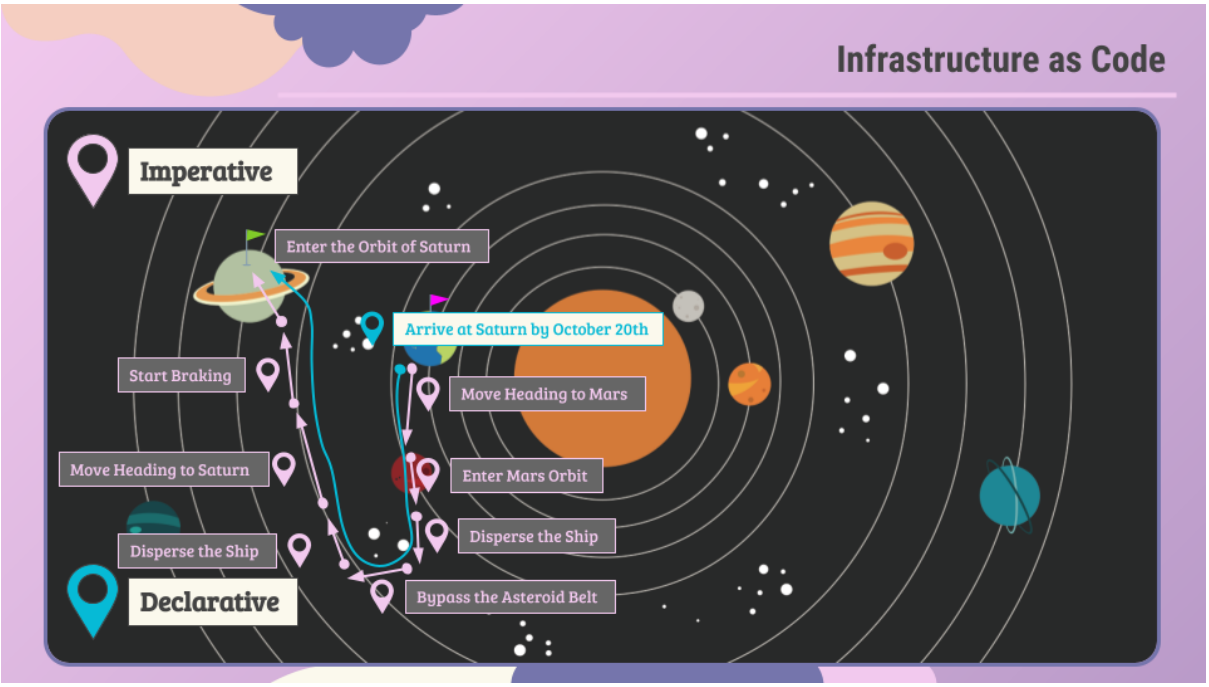


Привет и добро пожаловать. В этой лекции мы поговорим об отличиях императивного и декларативного подходов в Kubernetes, а в конце ты поймешь, как эти знания помогут тебе ускориться на экзамене.

До сих пор мы видели разные способы создания и управления объектами в Kubernetes.

Мы создавали объекты напрямую, выполняя команды, а также используя файлы определения конфигураций.

В настоящее время в мире инфраструктуры как кода (`infrastructure as code`) существуют разные подходы к управлению этой инфраструктурой. Они подразделяются императивный подход и декларативный подход.



Давай разберемся с этим на аналогии, допустим, нам нужно попасть на Сатурн к концу октября, чтобы посетить выставку редких минералов.

Как опытные космонавты мы бы поехали на космодром, сели в ракету и ввели в бортовой компьютер подробные пошаговые инструкции о том, как добраться до места назначения. Например:

- летим до Марса
- становимся на его орбиту
- ждем подходящего времени и используем марс для разгона
- проходим астероиды
- в определенное время разгоняем корабль
- в определенное время тормозим его
- и вот мы вышли на орбиту Сатурна

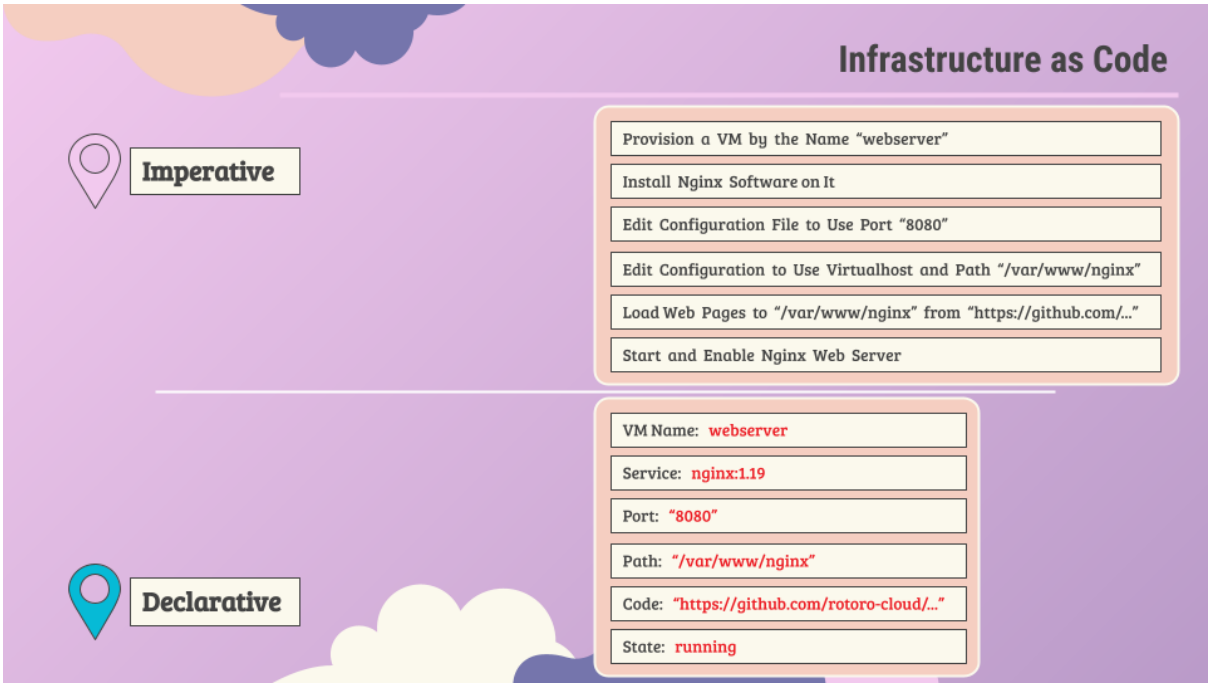
Т.е. мы полностью указываем, что и как делать. Это императивный подход.

Теперь посмотрим с другой стороны. Прежде чем ехать на космодром, мы указали на сайте время, когда нам нужно прибыть и место назначения. А в ответ получили время вылета.
Нам останется лишь не опоздать на космодром со своими чемоданами.

И это декларативный подход. В этом случае мы не даем пошаговых инструкций. Вместо этого мы просто указываем конечный пункт назначения и крайнюю дату прибытия.

Т.е. мы объявили требуемое конечное состояние, а система сама определяет правильный путь к месту назначения, указывая, что делать, а не как делать - вот это декларативный подход.

А какое отношение это имеет к тому, что мы изучаем?



В мире IaC примером императивного подхода к обеспечению инфраструктуры может быть набор инструкций, шаг за шагом. Например таких, как подготовка виртуальной машины с именем `webserver`, установка на нее программного обеспечения nginx, редактирование файла конфигурации для использования порта 8080, указания пути к документам сервера, скачивание исходного кода репозиториев GIT и, наконец, запуск непосредственно веб-сервера.

Здесь мы говорим, что требуется, а также как добиться результатов.

В декларативном подходе мы лишь декларируем свои требования.

Например, мы описываем целевое состояние так, что нам нужна виртуальная машина с именем `webserver` с программным обеспечением nginx, его рабочий порт должен быть 8080, он должен работать с приложением, которое будет лежать в `Path`, а взять его можно по указанному в `Code` адресу нашего GIT-репозитория.

И все, что необходимо сделать для создания этой инфраструктуры, будет выполнено самой системой или дополнительным программным обеспечением. Нам не нужно предоставлять пошаговые инструкции.

Инструменты управления, такие как Chef, Puppet, Ansible и Terraform придерживаются такого подхода.

А что будет, если с первого раза была выполнена только половина шагов?
Что произойдет, когда мы решим исправить ситуацию и снова предоставим тот же набор инструкций, чтобы выполнить оставшиеся шаги?

В этом случае, будет произведено множество дополнительных действий, вроде сбора информации и ее проверки, чтобы система понимала, существует ли уже что-то из того состояния, что заявлено.

И это понимание, что действительно нужно сделать, что не нужно делать, а что требуется переделать возникает на основе результатов этих тестов.

Тут много вопросов, например:

- Что произойдет при подготовке VM, если машина с именем `webserver` уже существует?
- Что делать с созданием базы данных случае сбоя? Следует проверить старую или сделать повторный импорт данных, ведь виртуальная машина уже существует.

Также все детали реализации не должны нас интересовать, например, если мы решим обновить версию программного обеспечения nginx, то это должно быть для нас просто.
Нам нужно лишь изменить версию в файле конфигурации nginx, а система должна позаботится обо всем остальном.

В идеале система должна быть достаточно интеллектуальной, чтобы знать, что уже было сделано, и применять необходимые в данный момент изменения.

И только декларативный способ делает подобные вещи.

Итак, императивным способом управления инфраструктурой в части создания объектов являются таких команды, как kubectl run для создания POD, команда kubectl create deployment для deployment, команда kubectl expose для создания службы для этого deployment.

Также мы можем изменить существующие объекты, например kubectl edit для редактирования ресурса, kubectl scale для масштабирования или kubectl set image для обновления образа в deployment.

Еще в практике этого подхода мы можем использовать файлы определения конфигурации объектов. Например команда kubectl create с параметром `-f` и указанием yml-файла.
В файле мы описываем объект, но в любом случае явно даем kubectl команду - create для создания, replace для замены или delete для удаления ресурса.

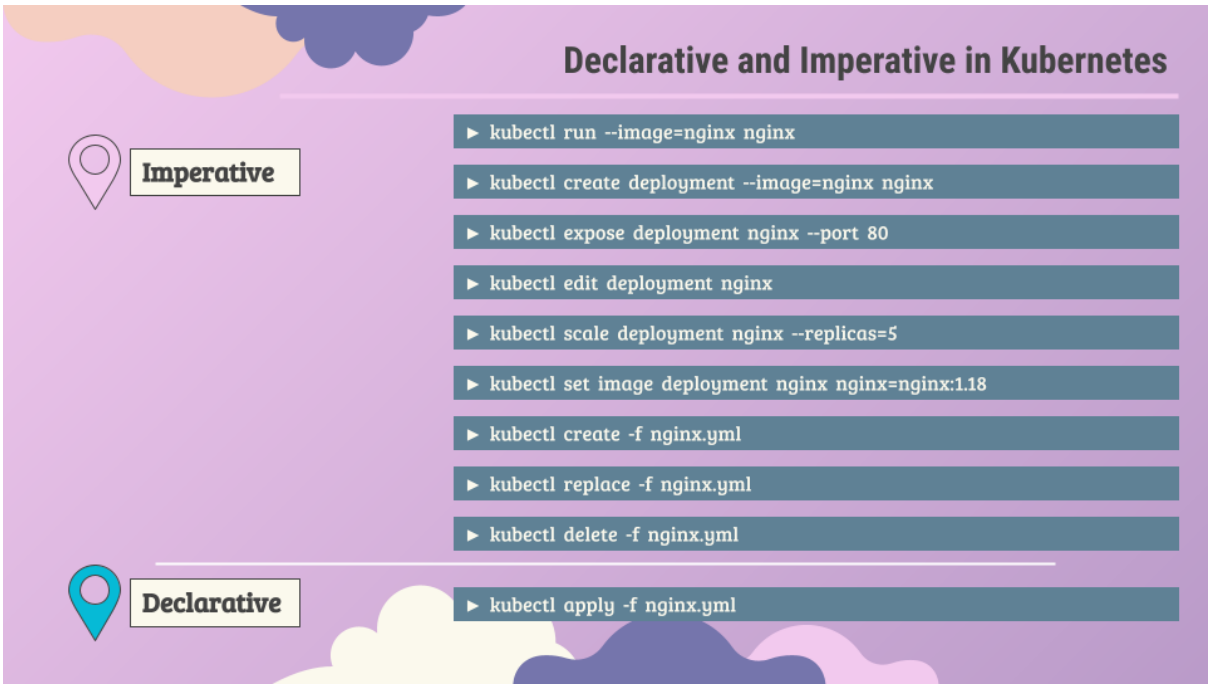
Все это императивный подход к управлению объектами и ресурсами.
Мы говорим, как именно привести инфраструктуру в соответствие с нашими потребностями путем создания, обновления или удаления объектов.

Декларативный подход заключается в создании набора файлов, который определяет ожидаемое состояние приложений и служб, так, чтобы процессы в кластере максимально не прерывались.

С помощью единой команды будет прочитан файл конфигурации, далее собрана информация о кластере, проанализировано его состояние и будет самостоятельно принято решенеие, что нужно сделать, чтобы привести инфраструктуру в ожидаемое состояние.

Итак, при декларативном подходе мы запустим команду kubectl apply для применения требуемого состояния системы. А система сама решит будут ли это команды для создания, обновления или удаления объекта.
Т.е. команда apply заставляет работать эту интелектуальную составляющую. Она просмотрит существующую конфигурацию и выяснит, какие изменения необходимо внести в систему.

Ок, давай теперь рассмотрим императивные команды более подробно.



Есть два пути.

Первый - это использование императивных команд, таких как команды run, create или expose, для создания новых объектов, их скалирования, а также упомянутые ранее команды для обновления существующих объектов.

Эти команды помогают быстро создавать или изменять объекты, поскольку нам не нужно иметь дело с файлами YAML, и они очень полезны во время сертификационных экзаменов.

Однако они ограничены в функциональности и потребуют формирования длинных и сложных цепочек команд для расширенных вариантов использования, таких, например, как создание многоконтейнерного POD или сложного deployment.

Еще один минус - эти команды запускаются один раз и забываются.
Они доступны только в истории оболочки пользователя, выполнившего эти команды.
Так что другому человеку сложно понять, как были созданы эти объекты.
Все эти изменения сложно отслеживать.

Поэтому с такими командами тяжело работать в больших или сложных средах, где изменение одного объекта может повлечь за собой неявное изменение других.

В таких случаях здесь нам поможет управление объектами с помощью файлов конфигурации объектов.

Создание файлов определения объекта, файлов конфигурации или файлов манифеста, это все название одного и того же, помогает нам точно зафиксировать, каким мы хотим видеть наши объекты. Описываем это в YAML-формате и используем kubectl create для создания объекта.

Теперь у нас всегда есть файл YAML, и его можно сохранить в репозитории нашей системы контроля версий.
После этого мы можем организовать процесс проверки и утверждения изменений для этих файлов, чтобы внесенное изменение было проверено и утверждено, прежде чем оно будет применено к производственной среде в будущем.

Когда необходимо внести изменения, например, изменить название образа на другую версию, есть разные способы сделать это.

Один из способов - использовать команду kubectl edit, указав имя объекта.
Таким образом, когда эта команда запускается, она открывает YAML-файл определения, аналогичный тому, который ты использовал для создания объекта, но с небольшими отличиями.

Здесь есть дополнительные поля, такие как поля статуса, которые ты здесь видишь. Они используются для хранения значений статуса POD. Т.е. это не тот файл, который мы использовали для создания объекта. Это аналогичный файл определения POD из памяти Kubernetes.

Ты можешь внести изменения в этот файл, сохранить и выйти, и эти изменения будут применены к `живому` объекту в кластере. Однако обрати внимание, что существует

разница между `живым` объектом и файлом определения, который у нас остался локально.

Изменение, которое мы сделали с помощью команды редактирования, на самом деле нигде не записывается. После применения изменения у нас останется только локальный файл определения, в котором фактически указан старый образ.

У нас появился конфигурационный дрифт - разница с тем, что записано в конфигурации и тем что происходит в реальной рабочей нагрузке.

В будущем, скажем, ты или твоя команда решите внести правки в этот объект, не зная, что изменение уже было внесено с помощью команды edit. И старое состояние будет потеряно, потому что когда новое изменение применяется, данные о предыдущей правке будут потеряны.

Итак, используй команду kubectl edit если ты вносишь изменения и уверен, что не это объект не связан с файлом определений, т.е. твоя команда не собирается полагаться на этот локальный файл конфигурации объекта в будущем.

Лучший подход к этому - сначала отредактировать локальную версию файла конфигурации объекта, внеся необходимые изменения, а именно обновив название образа, а затем запустить команду kubectl replace для обновления объекта.

Таким образом, ты изменил манифест, git это запомнил, и все заинтересованные лица могут отслеживать эти события в процессе мониторинга изменений.

Иногда нам требуется полностью удалить и воссоздать объекты, в таких случаях мы можем запустить ту же команду, но с опцией `--force`.

Хоть мы теперь и работаем с файлами, это по-прежнему императивный подход, потому что мы все еще инструктируем kubectl что именно делать и когда.

Т.е. сначала создать, а потом обновить или удалить эти объекты.
Сначала команда create, потом replace или delete.

А что, если запустить просто запустить команду create?

Если объект уже существует в данный момент, команда завершится с ошибкой, в которой говорится, что объект уже существует.

А при обновлении ты всегда должен убедиться, что объект существует, прежде чем запускать команду replace.
Если объект не существует, команда replace завершится сообщением об ошибке.

Таким образом, императивный подход очень утомителен для нас как администраторов, поскольку мы всегда должны быть в курсе текущих конфигураций

и выполнять проверки, чтобы убедиться, что все на месте, прежде чем вносить изменения.

Декларативный подход заключается в том, что мы используем те же файлы конфигурации объекта, с которыми мы работали ранее. Но вместо команд create, replace мы используем команду apply для управления состоянием объектов.

Команда kubectl apply достаточно умна, чтобы создать объект, если он еще не существует.

Если у тебя несколько файлов конфигурации объекта, ты можешь указать путь размещения другого файла. Также ты можешь указать несколько файлов и определить в файле несколько объектов, и они будут обработаны одной командой за раз.

Теперь, когда необходимо внести изменения, мы просто обновляем файл конфигурации и снова запускаем команду apply. На этот раз она знает, что объект существует, поэтому только обновит объект новыми изменениями из файла.

Таким образом, команда никогда не выдает ошибку, которая говорит, что объект уже существует или обновления не могут быть применены. И всегда определяет правильный подход к обновлению объекта.

В дальнейшем для любых изменений, которые мы захотим внести в приложение, будь то обновление образа или других полей существующих файлов конфигурации, добавление новых файлов конфигурации для новых объектов, все, что нам

потребуется - лишь обновить наш локальный каталог с изменениями, а затем сделать kubectl apply, и она позаботится обо всем остальном.

Мы подробнее обсудим, как именно эта команда работает в следующей лекции.
А пока позволь дать несколько советов в рамках экзамена.

На экзамене тебе стоит использовать императивный подход, чтобы максимально сэкономить время. Например, если вопрос состоит в том, чтобы просто создать POD или deployment с заданным образом, то одна из этих быстрых команд может помочь тебе добиться этого.

Поэтому важно практиковать императивные команды, знать их возможности и ограничения.

Если тебе потребуется отредактировать свойство существующего объекта, то использование команды edit, может быть самым быстрым способом.

Но если у тебя есть сложное задание, например, оно требует нескольких переменных среды контейнера, команд в контейнерах и т. д., то использование файла определения конфигурации для создания объекта будет предпочтительнее.

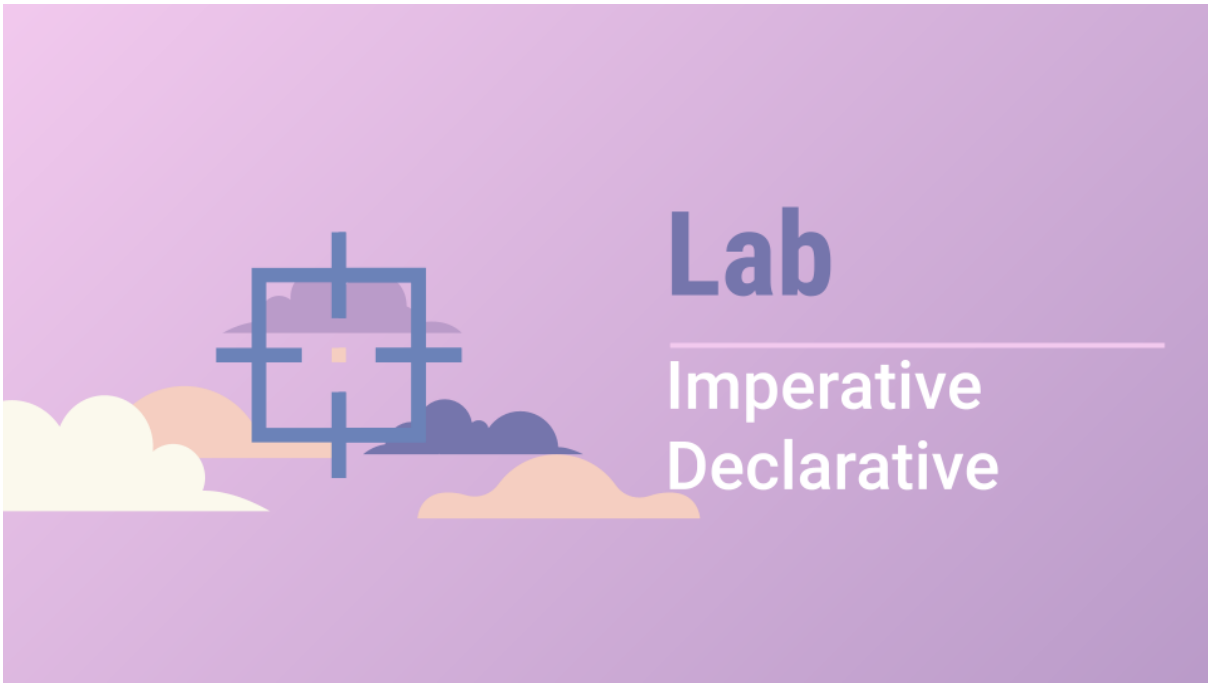
Опять же, если ты видишь, что допустил ошибку, то сможешь легко поправить файл и применить его снова. И использование в этом случае kubectl apply будет лучшим вариантом.

Т.е. для простых задач экзамена, где мала вероятность ошибиться хорош императивный стиль, для тех, где легко ошибиться лучше декларативный.

Я дам еще несколько подсказок по императивным командам в презентации курса.

Для получения более подробной информации о различных подходах к управлению кластером ознакомся со страницами документации. Еще у нас будет лабораторная для практики, где ты сможешь попрактиковать использование императивного подхода при решении своих вопросов.

Что ж, увидимся на следующей лекции.


Lab
Imperative
Declarative