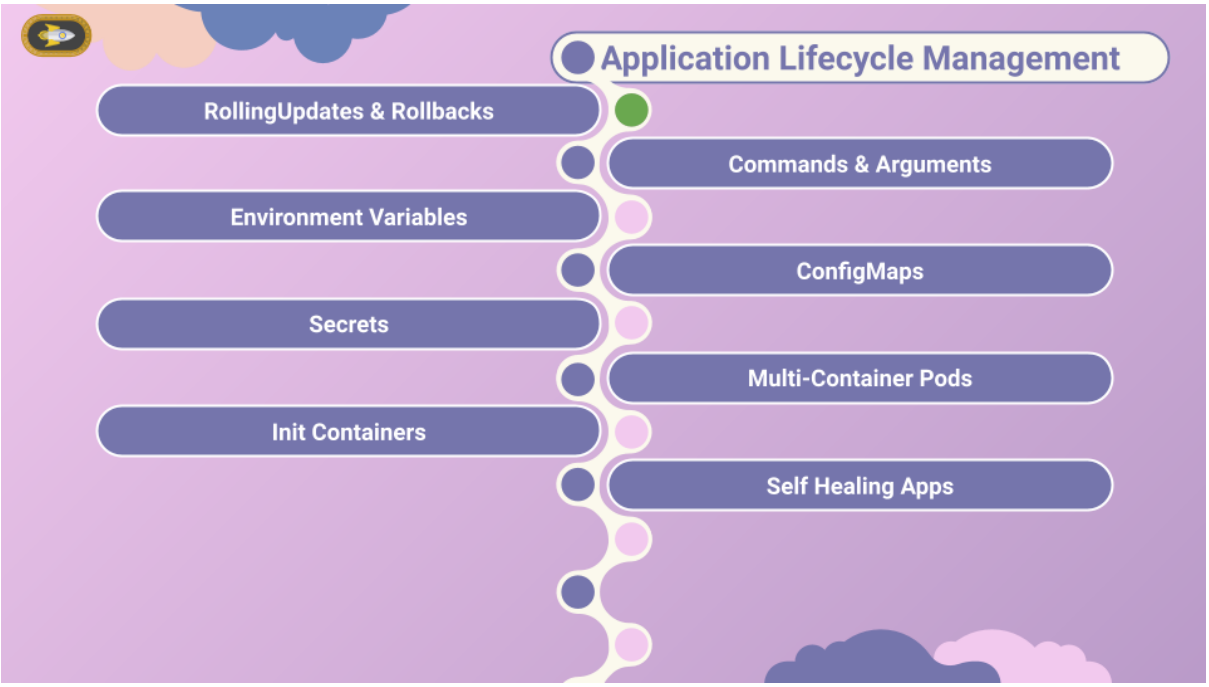
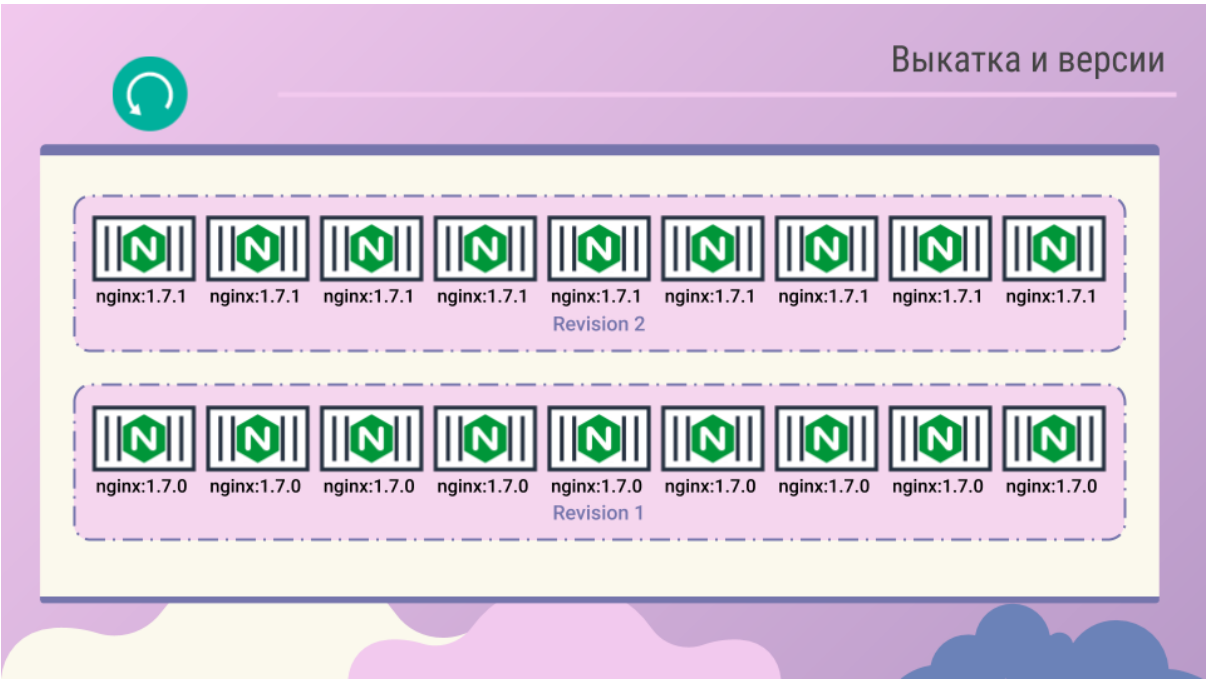
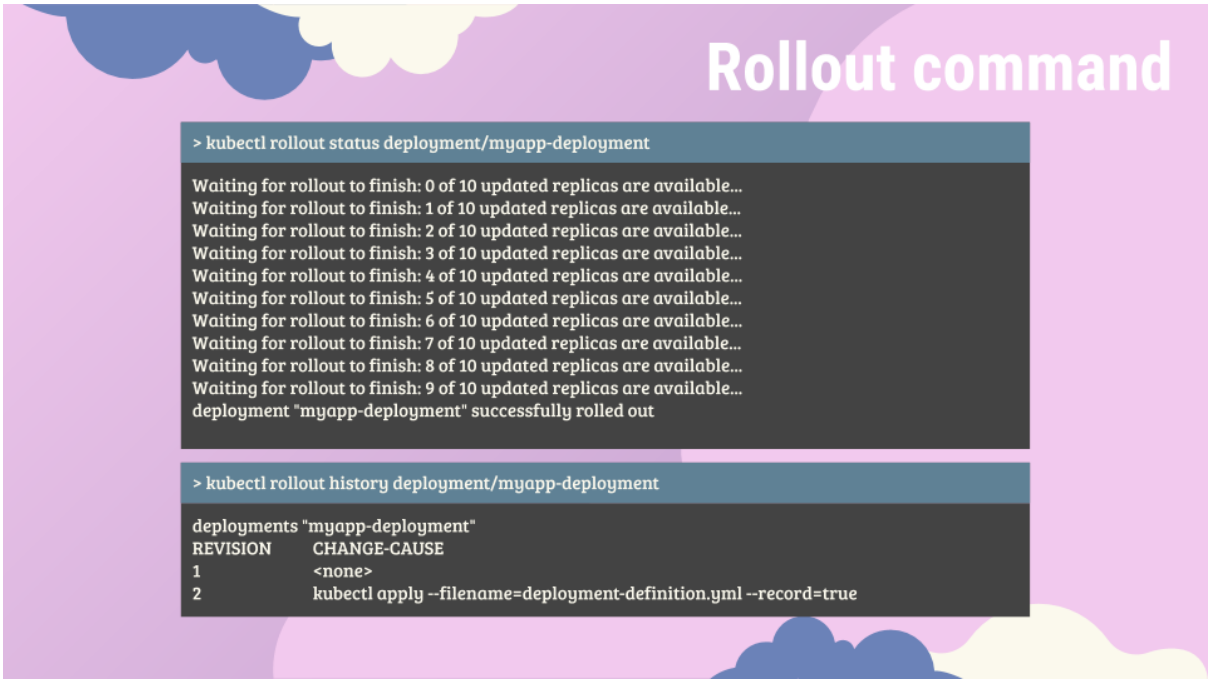


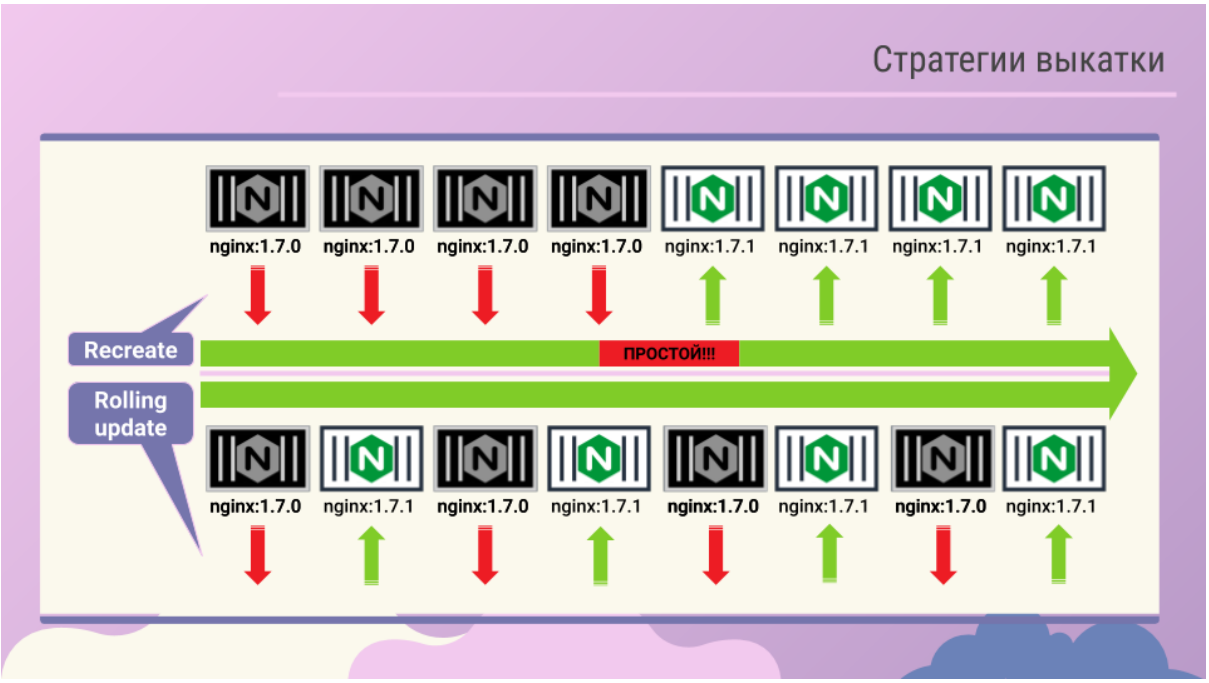


Прежде чем мы рассмотрим, как обновлять наше приложение, давай попробуем разобраться в версионировании и откатах в Deployments.
Каждый раз при создании нового Deployment или обновлении образа в существующем, запускается Rollout. Rollout - это процесс постепенного развертывания или обновления контейнеров приложений. Как я сказал, при создании Deployment запускается Rollout. Этот Rollout создает ревизию развертывания, его отпечаток, снепшот. Назовем его Revision 1.

В будущем, когда приложение будет обновлено, то есть когда версия контейнера будет меняться на новую, запустится новый Rollout, который создаст новую ревизию с именем Revision 2.
Эта функция помогает нам отслеживать изменения, внесенные в deployment и позволяет при необходимости вернуться к предыдущей развернутой версии.



Используй команду kubectl rollout status, за которой следует имя deployment, чтобы увидеть состояние выкатки.
Для просмотра списка ревизий и истории изменений выполни команду kubectl rollout history и имя deployment.



В Kubernetes есть два типа стратегий развертывания.
Например, у нас развернуто 4 реплики веб-приложения. Один из способов обновить их до нужной версии - уничтожить все инстансы, а затем создать экземпляры новой версии. Это означает, что сначала нужно уничтожить 4 запущенных экземпляра, а затем развернуть 4 новых экземпляра требуемой версии приложения.
Проблема заключается в том, что в течение периода между тем, как старые версии уже не работают, а новые еще не поднялись, приложение не запущено и недоступно для пользователей.

Эта стратегия известна как Recreate strategy, и, к счастью, это НЕ стратегия deployment по умолчанию.

Вторая стратегия заключается в том, что мы не уничтожаем все инстансы сразу. Вместо этого мы останавливаем экземпляр старой версии и запускаем новый. И так один за другим. Таким образом, приложение никогда не прекращает работу, а обновление происходит плавно.

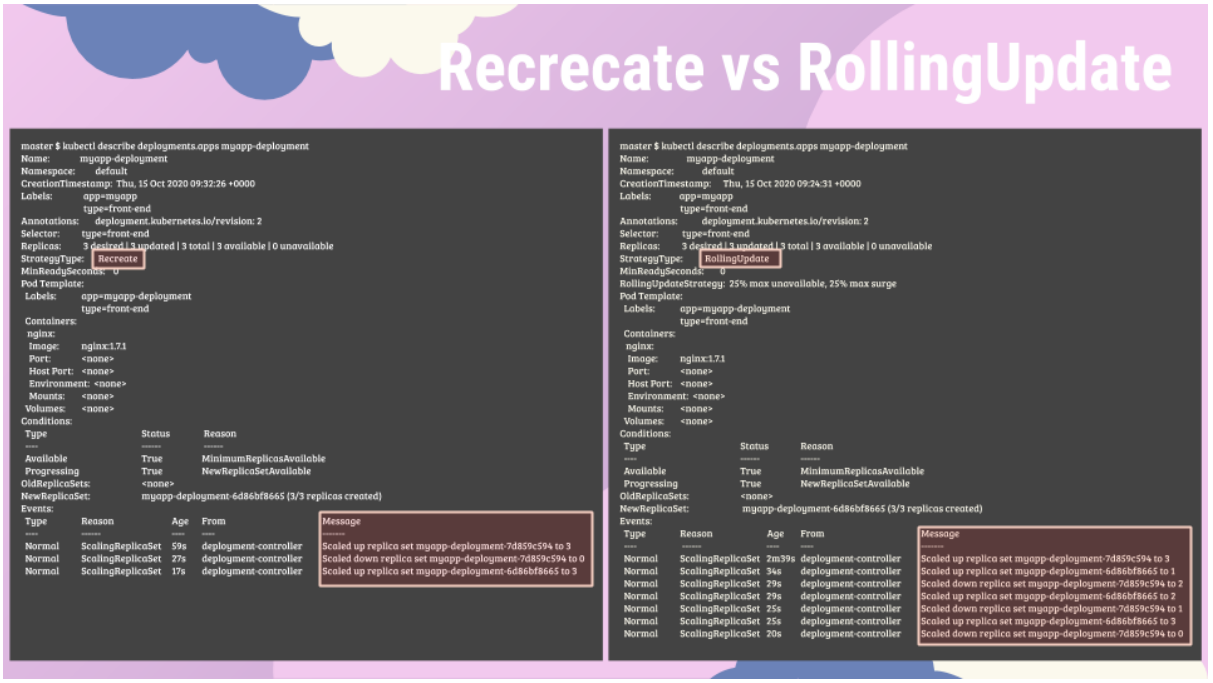
Имей в виду, если ты не укажешь стратегию при создании deployment, Kubernetes будет считать, что это RollingUpdate. Другими словами, RollingUpdate - это стратегия развертывания по умолчанию.

Ок, мы поговорили об обновлениях. Но как именно нам это делать, обновлять deployment?
Когда я говорю об обновлении, нужно понимать, что это могут быть разные вещи: мы можем обновить версию Docker-образа нашего приложения и через это изменить версию приложения, или мы можем обновить метки PODs, или изменить количество реплик и т. д.

Поскольку у нас есть файл определения deployment, нам легко внести изменения. После необходимых исправлений мы запускаем команду kubectl apply, чтобы применить изменения. Запускается новый rollout и создается новая версия deployment.

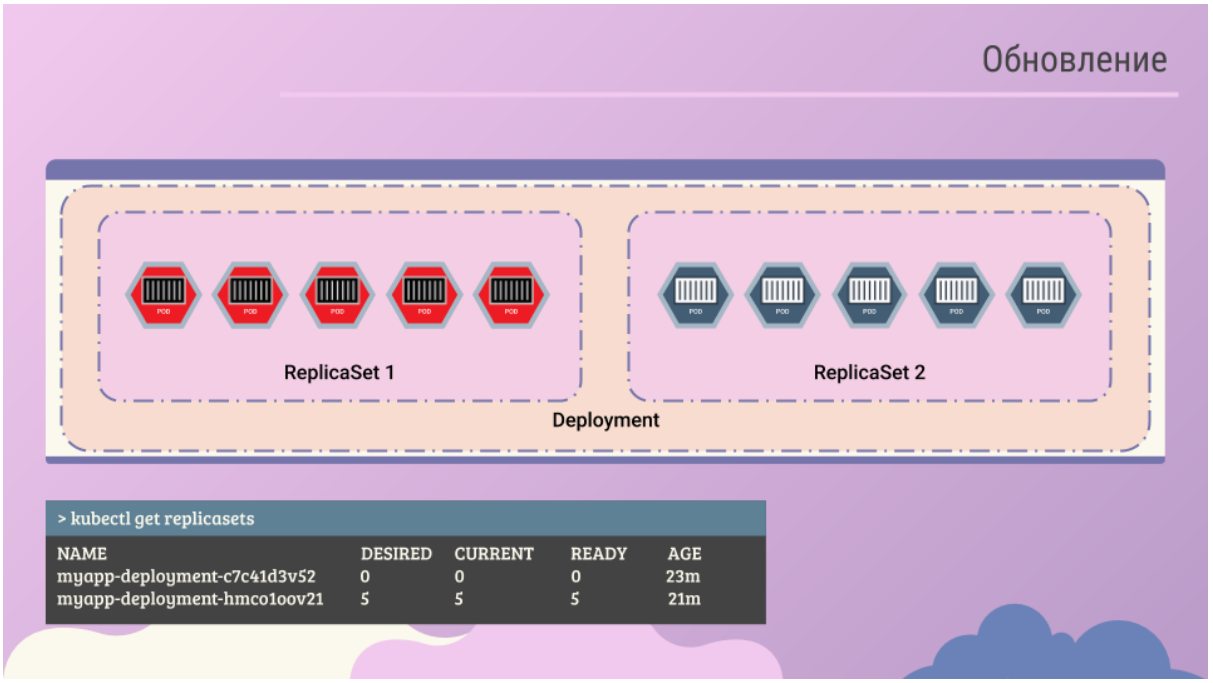
Но есть ДРУГОЙ способ сделать то же самое. Ты можешь использовать команду kubectl set image для обновления образа приложения. Но помни, что в манифесте останется старая версия образа и она будет отличаться от конфигурации в кластере.

Это называется дрифт конфигураций. Поэтому, если ты практикуешь ручное конфигурирование, нужно быть аккуратным при использовании не обновленных файлов определения.



Разницу между стратегиями Recreate и RollingUpdate также можно увидеть при подробном рассмотрении deployment.
Запусти команду kubectl describe deployment, чтобы просмотреть подробную информацию о развертывании. Как видишь, при использовании стратегии Recreate события (events) показывают, что старый ReplicaSet сначала уменьшен до 0, а новый ReplicaSet сразу увеличен до 3.

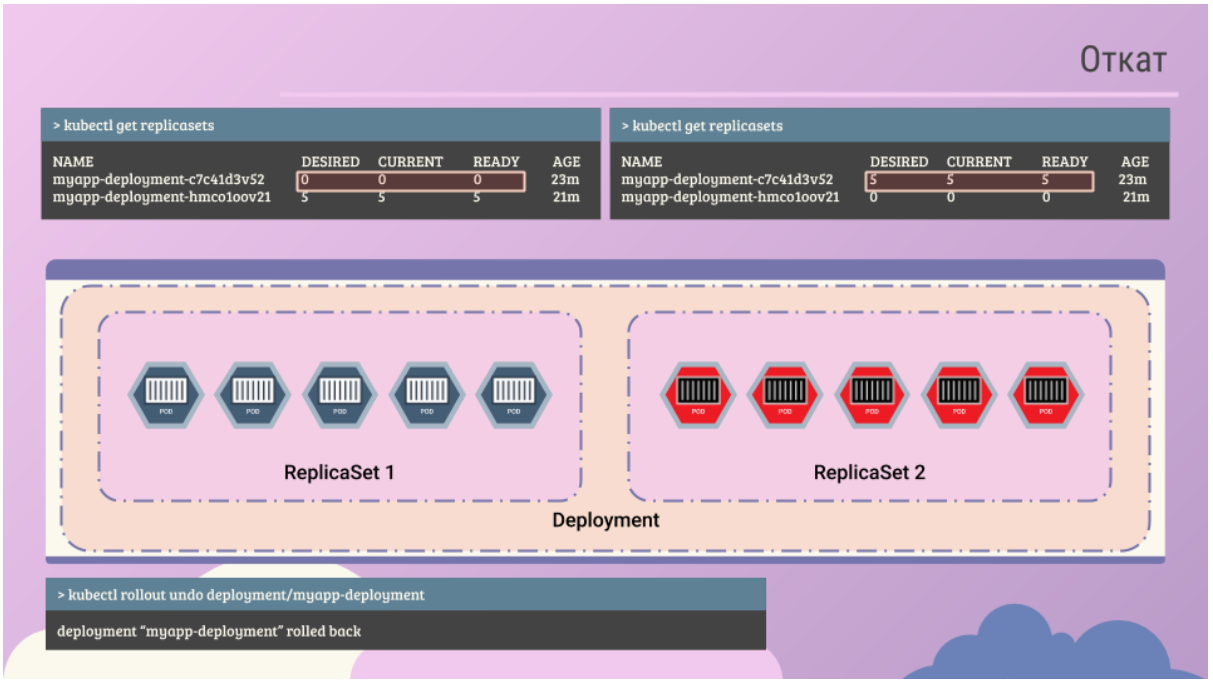
Однако, когда использовалась стратегия RollingUpdate, старый ReplicaSet уменьшался по одному, и в соответствии с этим также увеличивалось количество реплик нового ReplicaSet.



Заглянем deployment под "капот". Когда создается новый deployment, скажем, для развертывания 5 реплик, он сначала автоматически создает ReplicaSet, который, в свою очередь, создает количество POD, необходимое для соответствия количеству реплик.

Когда ты обновляешь свое приложение, как мы видели на предыдущем слайде, объект deployment создает НОВЫЙ ReplicaSet и начинает развертывание контейнеров в нем.

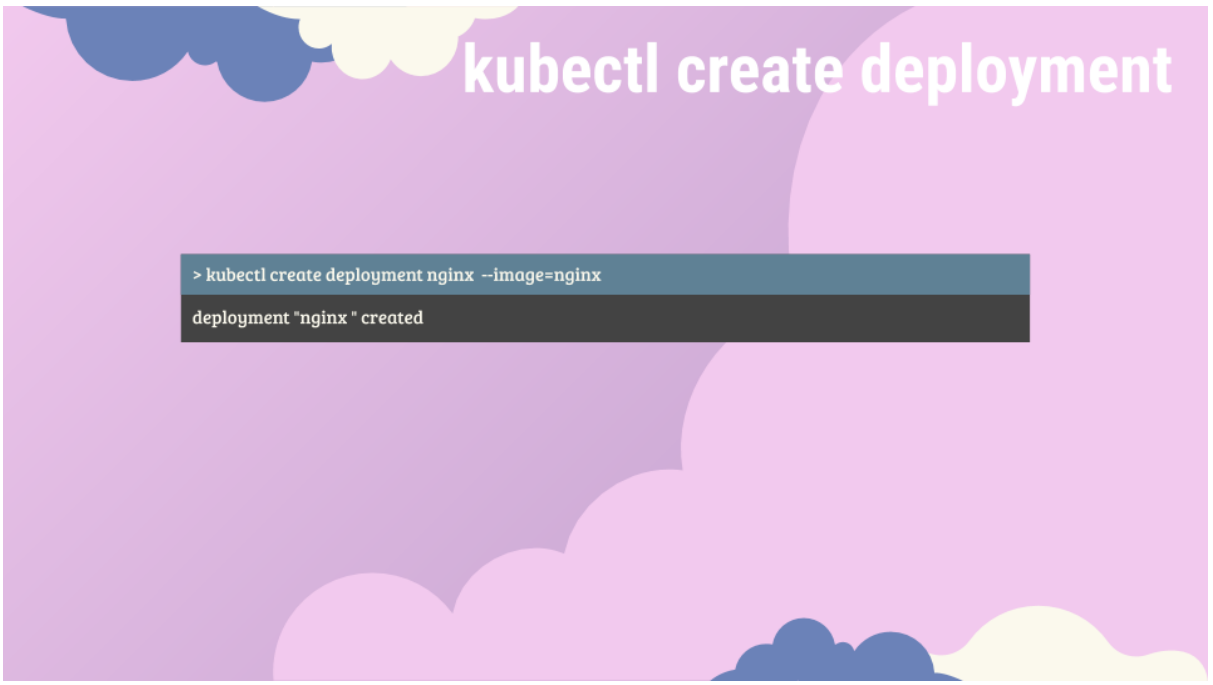
В то же время удаление POD в старом ReplicaSet идет в соответствии со стратегией RollingUpdate.
Это можно увидеть с помощью команды kubectl get replicasets.
Здесь мы видим старый набор реплик с 0 POD и новый набор реплик с 5 POD.



Допустим, мы обновили свое приложение, и вдруг поняли, что что-то пошло не так. Что-то неправильно с новой версией сборки, которую мы использовали для обновления. Итак, нам надо откатить обновление.

Deployments Kubernetes позволяют вернуться к предыдущей версии. Чтобы отменить изменение, выполни команду kubectl rollout undo, после которой укажи имя deployment. После этого deployment уничтожит PODs в новом ReplicaSet и восстановит старые PODs в старом ReplicaSet.

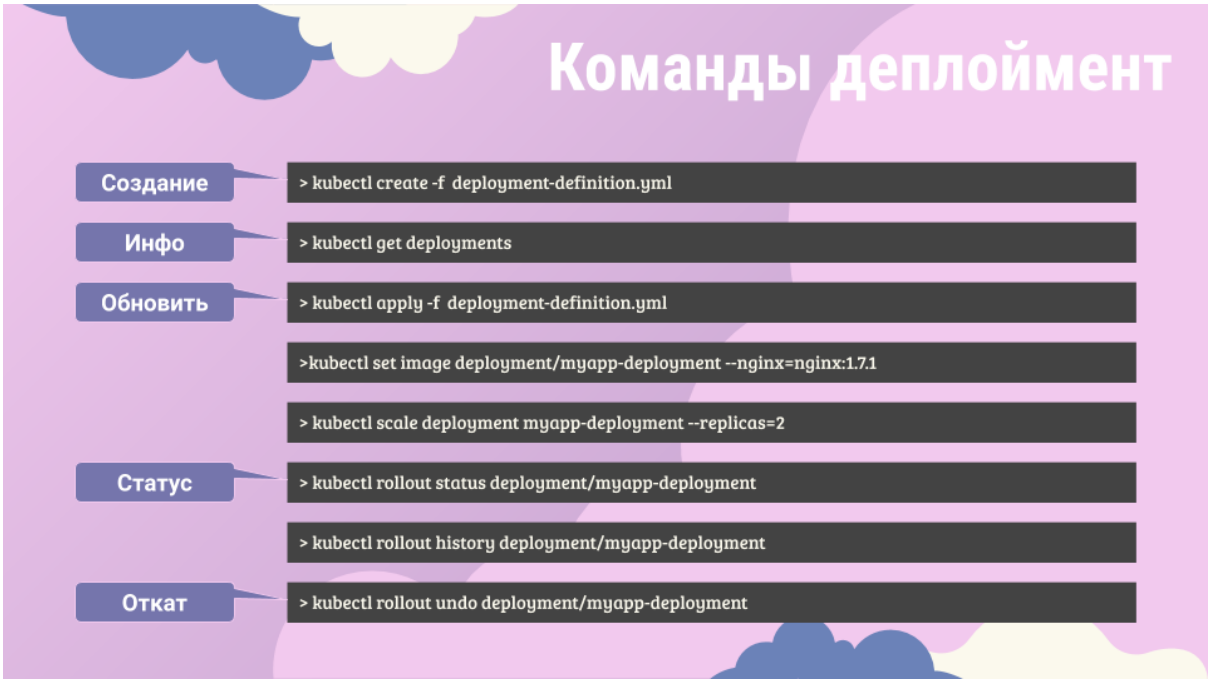
Как видишь, наше приложение вернулось к своему старому виду. Сравним вывод команды kubectl get replicasets до и после отката. Перед откатом в первом наборе реплик было 0 POD, а в новом наборе реплик было 5 POD, после завершения отката все стало наоборот.



Как и в случае для POD, deployment можно создать быстрой командой kubectl create deployment с указанием образа. Очень похоже на команду kubectl run для создания POD.

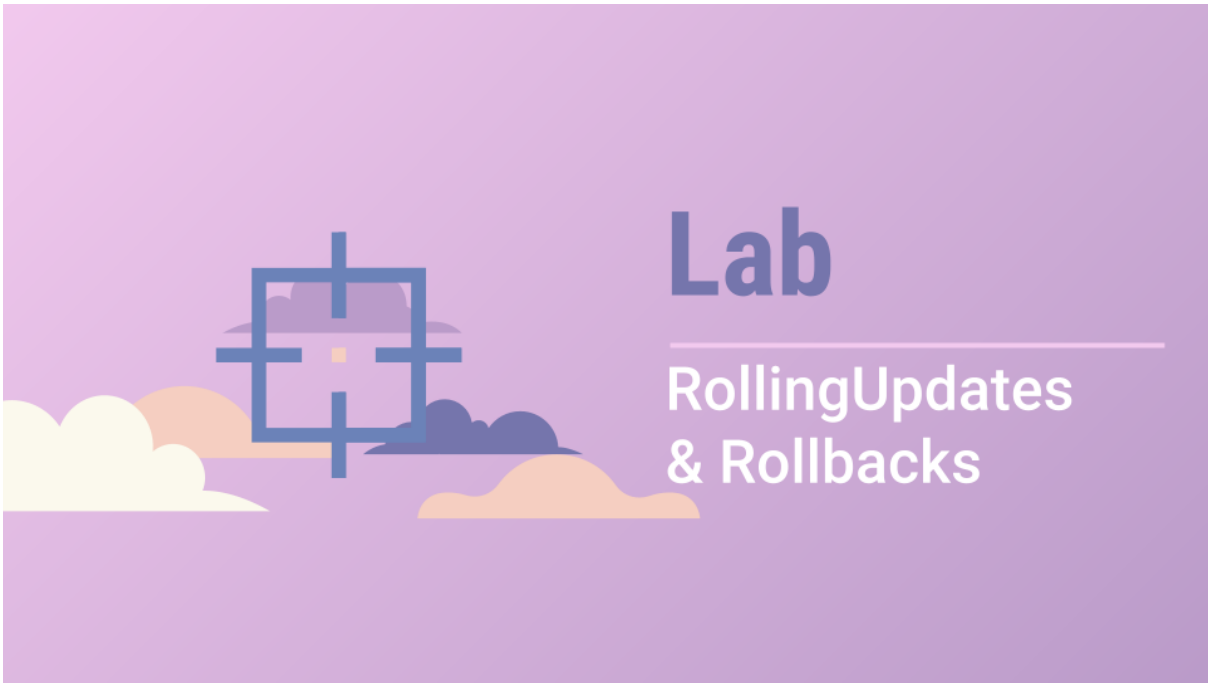
Это один из способов создания развертываний, указав только имя образа и не используя файл определения.

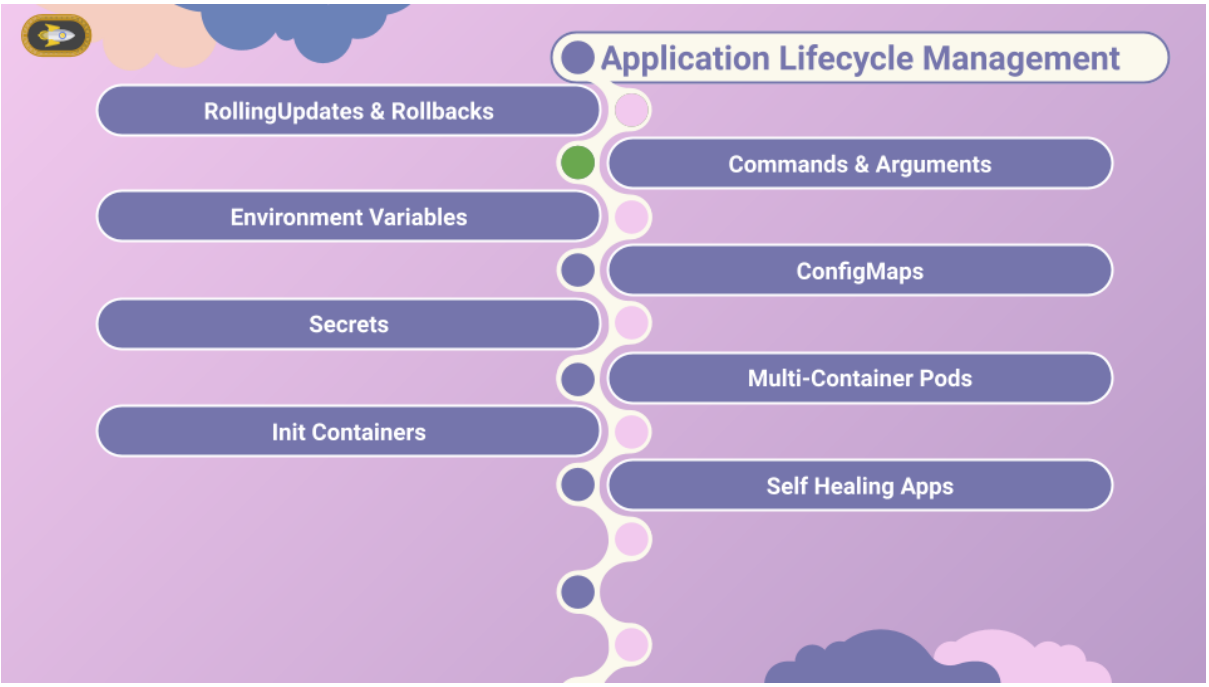
В свою очередь ReplicaSet и PODs создаются автоматически.
Тем не менее, рекомендуется использовать манифест, поскольку появляется возможность сохранить файл в системе контроля версий, и при необходимости изменить его позже.
Это является хорошей практикой в свете принципа "Инфраструктура-как-код".



Давай быстро пробежимся по командам, для резюмации.

Используй команду kubectl create для создания deployment.
Команду get deployments для получения списка развертываний,
Команды apply и set image для обновления deployments, а команду scale для масштабирования.
Команду rollout status, чтобы увидеть состояние выкатки, а rollout history для истории ревизий.
Команду rollout undo для отката на другую ревизию.

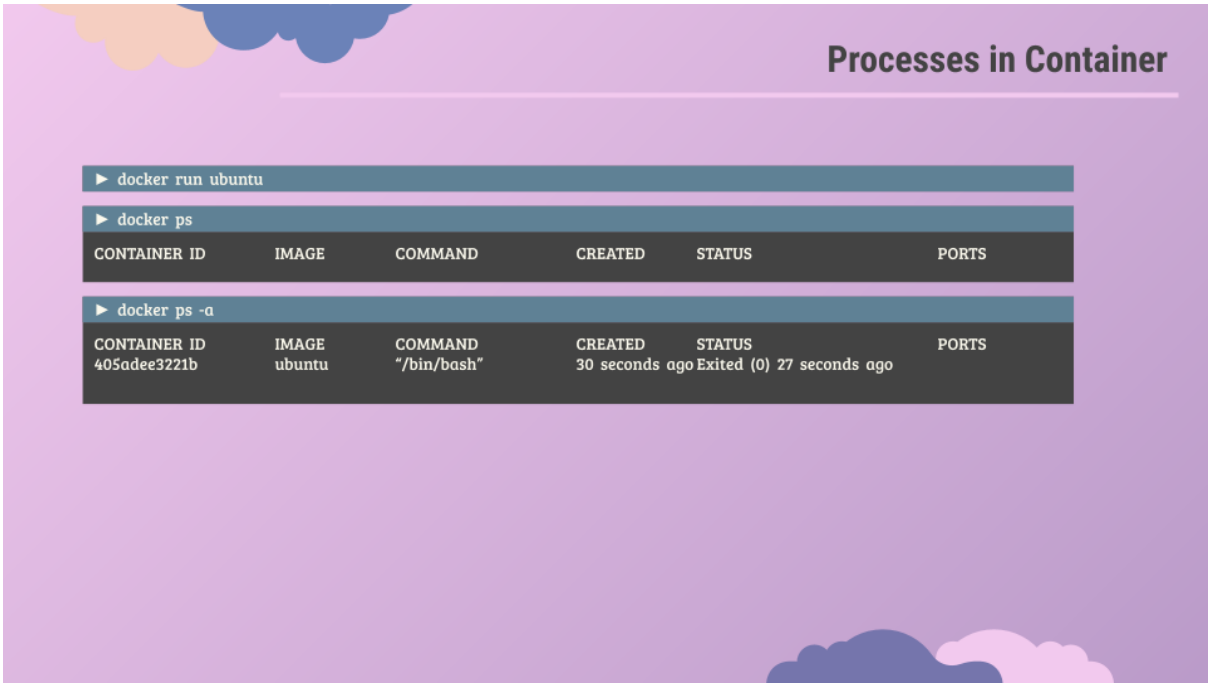




Привет и добро пожаловать на эту лекцию.
Эта и следующая лекции ведут нас к пониманию команд и аргументов в контейнерных средах.

Эта тема не указана в качестве обязательной в учебной программе сертификации, но я думаю, что ее важно объяснить, поскольку этой теме обычно не уделяют внимания, что ведет к путанице. Давай сначала освежим нашу память о командах в докер-контейнерах.

В следующей лекции мы будем смотреть с перспектив Kubernetes, а сейчас поговорим о аргументах команд и точках входа в Docker.



Начнем с простого сценария.
Предположим, что мы запустили контейнер с образом ubuntu.

Когда мы ввели команду docker run ubuntu, Docker запустил экземпляр образа ubuntu и немедленно сделал выход из контейнера.

Если сейчас посмотреть список работающих контейнеров, то мы не увидим их.

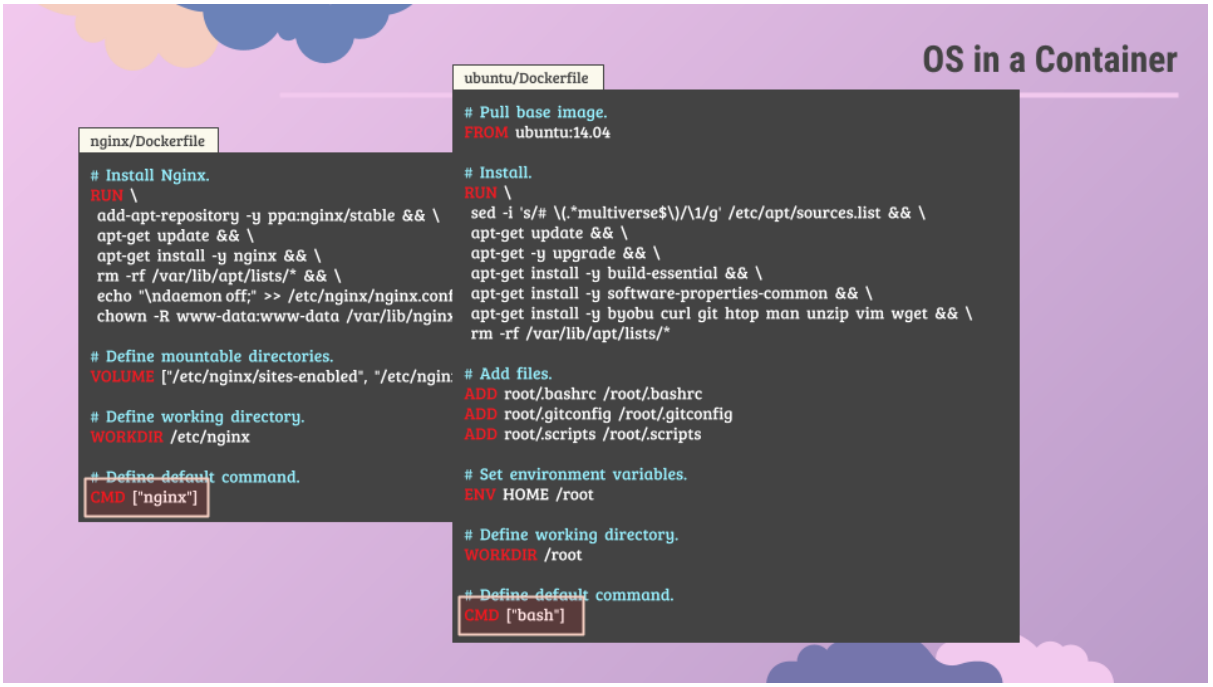
Если же запустить docker ps -a, тем самым запросив список всех контейнеров на хосте, мы увидим наш новый контейнер, он будет остановлен со статусом exited.

Почему это не работает как виртуальная машина?

Как мы уже говорили, контейнеры не предназначены для размещения операционных систем. Они подходят для некоторых процессов вроде запуска веб-сервера, базы данных или аналитической задачи. Когда задача завершена, контейнер останавливается. Жизнь в нем поддерживает работающий внутри процесс.

Т.е. если веб-сервер, или база данных, или другой запущенный процесс внутри контейнера остановится или скрашится, то контейнер немедленно выйдет.

Итак, что определяет, какой процесс будет запущен в контейнере?



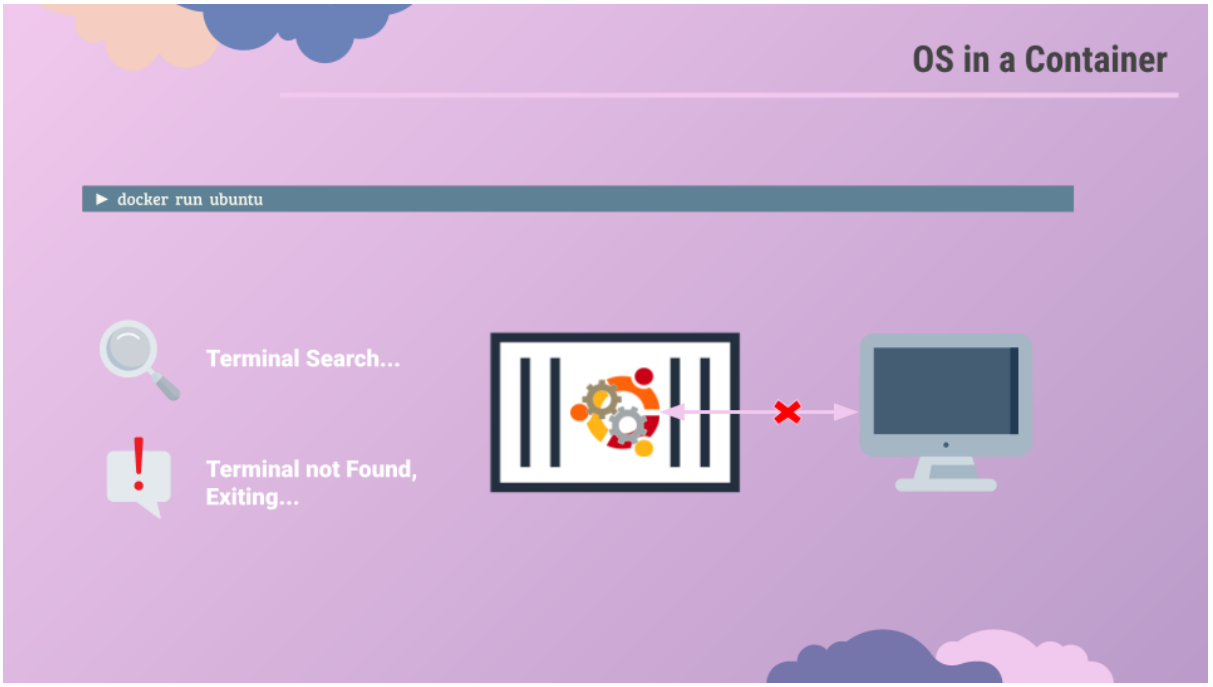
Если посмотреть на Dockerfile такого популярного приложения как nginx ты увидишь инструкцию CMD. Эта инструкция определяет команду, которая будет запускаться в контейнере при запуске его из образа. Это команда nginx.

Для официального образа MySQL в Dockerfile будет команда mysqld.

Ранее мы пытались запустить контейнер с пустой ОС Ubuntu.
Посмотрим на Dockerfile этого образа и мы увидим, что команда по умолчанию там bash. Но как мы знаем bash это не совсем такой процесс, вроде веб-сервера или базы данных.

Это оболочка, которая ждет ввода с терминала и, если если она не может найти терминал, она завершается.

Когда мы запускали контейнер с Ubuntu ранее, Docker создавал контейнер из образа Ubuntu и запускал программу bash по умолчанию. Терминал к контейнеру при запуске Docker не подключал.



Таким образом, программа bash не находила терминал и завершала работу.
А поскольку процесс, который был запущен при создании контейнера, завершался, то и контейнер также останавливался.

Как нам указать другую команду при старте контейнера?

Один из вариантов - добавить нужную команду к docker run и таким образом переопределить команду по умолчанию, указанную в образе.

В этом случае мы выполним docker run Ubuntu с командой sleep 5 в качестве дополнительного параметра.

Теперь, когда контейнер стартует, будет запущена команда sleep. Она будет выполняться 5 секунд, после чего удачно завершится, что соответственно и завершит выполнение контейнера.

Как сделать эти изменения постоянными? Скажем, тебе нужно всегда запускать команду sleep.

Создай свой собственный образ из базового образа ubuntu и укажи новую команду с помощью инструкции CMD.
Есть несколько разных возможностей указать команду:

- как ты ее пишешь в шелле
- в виде JSON-массива

Не забудь, если ты укажешь это в JSON, первым элементом массива должен быть исполняемый файл.

В этом случае инструкция CMD написана неверно. "sleep 5" команда и параметр указаны вместе. Это не сработает, команда и параметры должны быть разнесены в отдельные элементы массива.

Ок, с помощью Dockerfile который ты видишь, я создам свой новый образ ubuntu-sleeper с помощью команды:

docker build ubuntu-sleeper

Теперь можно просто запустить его контейнер используя:

docker run ubuntu-sleeper

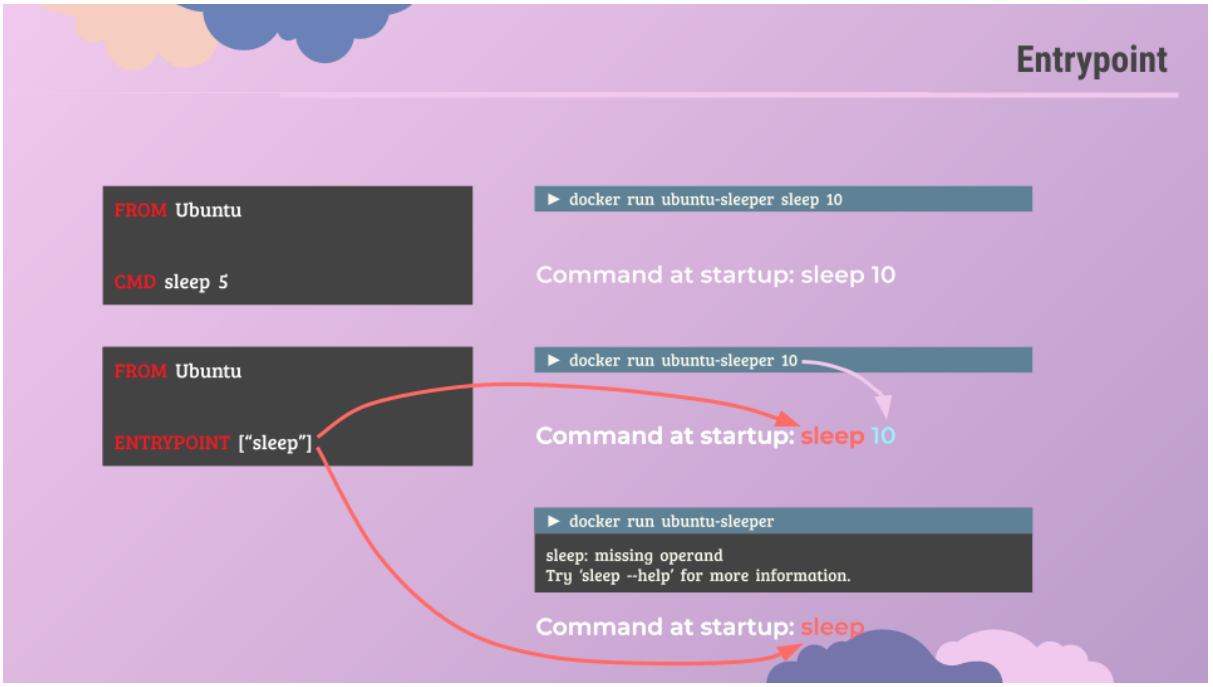
Я получил предполагаемый результат - контейнер запускается, ждет 5 секунд и выходит. Это количество секунд захардкожено в моей инструкции CMD.

Как мне изменить этот промежуток, не меняя образ?

Один из путей запустить контейнер с добавлением новой команды
docker run ubuntu-sleeper sleep 10

В этом случае команда sleep 10 выполнится при старте и успешно отработает положенные ей 10 секунд. Но это
решение так себе.

Само название ubuntu-sleeper подразумевает, что контейнер будет спать, не хотелось бы дополнительно указывать инструкцию вроде "проспи 10 секунд".



Что-то более изящное, например не указывать команду sleep, а указать только количество нужных секунд, что-то вроде:

docker run ubuntu-sleeper 10

Я хочу, чтобы контейнер меня при этом понял и автоматически вызвал команду sleep с моим параметром.

Вот здесь появляется инструкция ENTRYPOINT.

Инструкция ENTRYPOINT похожа на инструкцию CMD, в ней ты также указваешь программу или команду, которая будет

запущена в контейнере, а также все дополнительные аргументы, которые ты бы указал в командной строке.

В этом случае 10 будет добавлено в значение ENTRYPOINT и будет собрана единая команда.
Таким образом контейнер запустится с командой sleep 10.

В этом отличия этих двух инструкций.
В случае инструкции CMD переданные параметры командной строки будут полностью заменены, тогда как в случае

ENTRYPOINT эти параметры будут добавлены.

Что случится, если я запущу docker run ubuntu-sleeper без добавления количества секунд?

Здесь к команде sleep добавится пустой параметр, т.е. образуется просто команда sleep, и я получу сообщение об отсутствии операнда. Значит, хорошо бы иметь значение по умолчанию на такой случай.

Этого можно добится используя обе инструкции сразу: ENTRYPOINT и CMD.
В ENTRYPOINT пропишу исполняемую команду, а в CMD значение аргумента по умолчанию.

Теперь команда для старта в контейнере соберется из ENTRYPOINT и значения из CMD, т.к. мы не передали аргументов в командную строку и тем самым не переопределили CMD.

Таким образом получим команду sleep 5. Если же мы укажем параметр, то инструкция CMD будет переопределена, как в случае с sleep 10.

Запомни, чтобы это сработало всегда нужно указывать ENTRYPOINT и CMD в формате JSON.

И наконец, что, если нам потребуется изменить ENTRYPOINT во время выполнения? Использовать какую-то другую команду, например заменить sleep на super-sleep.
Это возможно используя параметр --entrypoint.

docker run --entrypoint super-sleep ubuntu-sleeper 10

В этом случае контейнер будет запущен с командой super-sleep 10 внутри.

Это все в этой лекции, давай погрузимся в практику с нашими лабораторными.

Привет и добро пожаловать в лекцию, где мы рассмотрим команды и аргументы в PODs Kubernetes.

В предыдущей лекции мы создали простой докер-образ, который спит определенное количество секунд.

Мы назвали его ubuntu-sleeper и запустили его с помощью команды `docker run ubuntu-sleeper`.

По умолчанию он спит в течение пяти секунд, но ты можешь переопределить его, передав аргумент командной строки.

Теперь мы создадим POD, используя этот образ.

Мы начинаем с пустого шаблона определения POD, вводим его название, указываем имя образа.

Когда POD создается, он создает контейнер из указанного образа, и контейнер находится в спящем режиме в течение пяти секунд, прежде чем выйти.

Теперь, нам нужно, чтобы контейнер спал 10 секунд, как во второй команде. Как указать дополнительный аргумент в файле определения POD?

Все, что дополнительно нужно добавить к команде `docker run`, будет помещено в свойство `args` файла определения POD в виде массива, как ты видишь здесь.

Давай попробуем связать это с Dockerfile из предыдущей лекции.

В Dockerfile есть ENTRYPOINT, а также указана инструкция CMD. ENTRYPOINT - это команда, которая выполняется при запуске, а CMD - это параметр по умолчанию, передаваемый команде.

С помощью параметра args в манифесте POD мы переопределяем инструкцию CMD из Dockerfile.

Но что, если нам нужно переопределить ENTRYPOINT? Скажем заменить ее как раньше мы делали с Docker, когда меняли с `sleep` на `super-sleep`?

В мире Docker мы бы запустили команду docker run с параметром --entrypoint, в котором указали бы новую команду. В мире Kubernetes в файле определения POD есть соответствующее этому параметру поле, оно называется `command`.

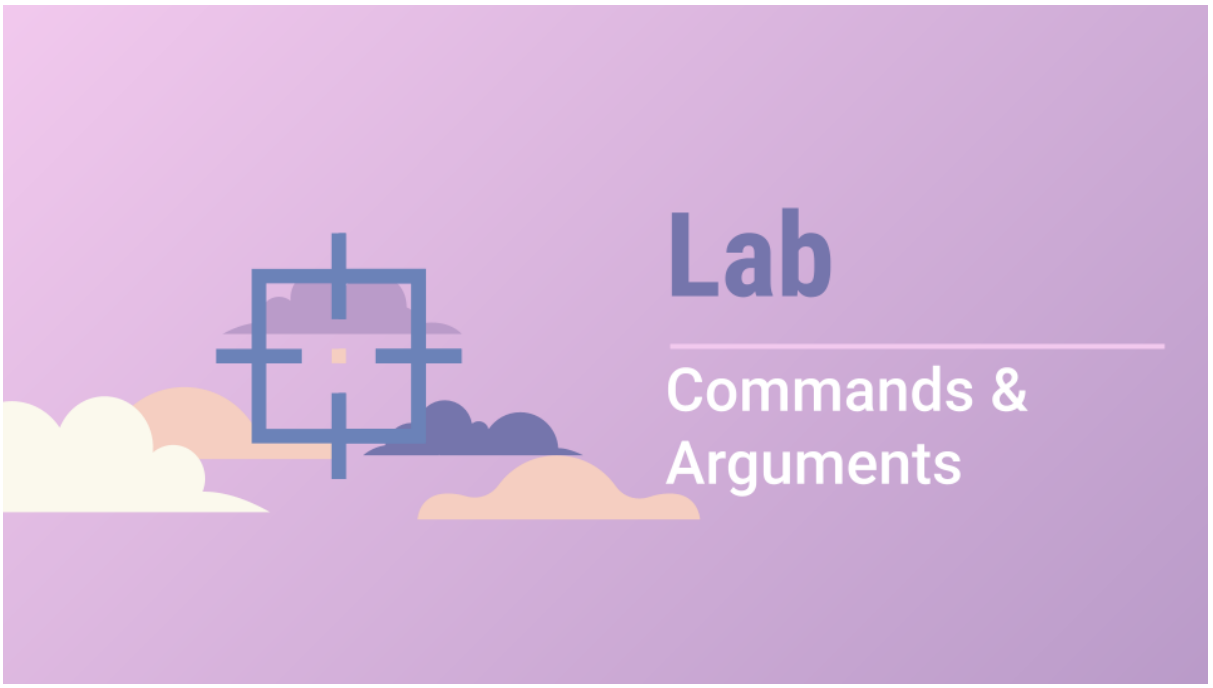
Это поле, если оно задано, будет использовано Kubernetes чтобы запускать контейнер с измененной инструкцией --entrypoint.

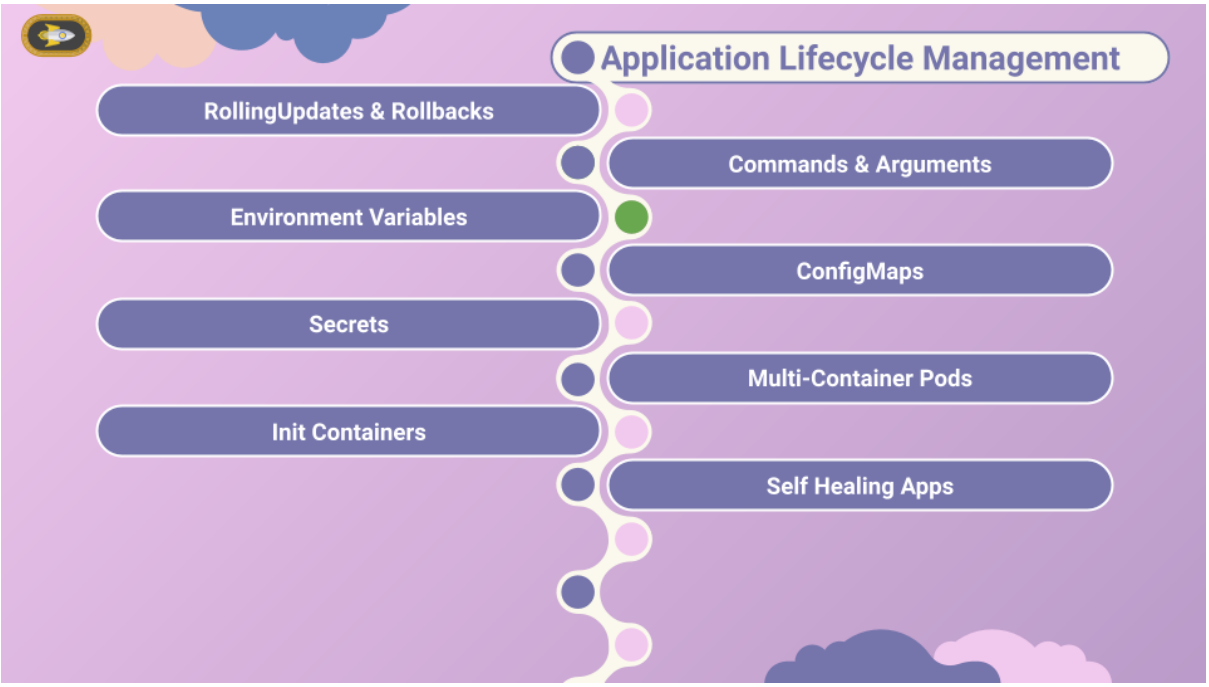
Чтобы подвести итог: есть два поля которые соответствуют двум инструкциям в Dockerfile.

Поле command переопределяет оригинальную инструкцию ENTRYPOINT из Dockerfile, поле args переопределяет инструкцию CMD.

Часто новичков это сбивает. Помни, что command в Kubernetes это тоже, что ENTRYPOINT в Docker. Аналог CMD - это args.

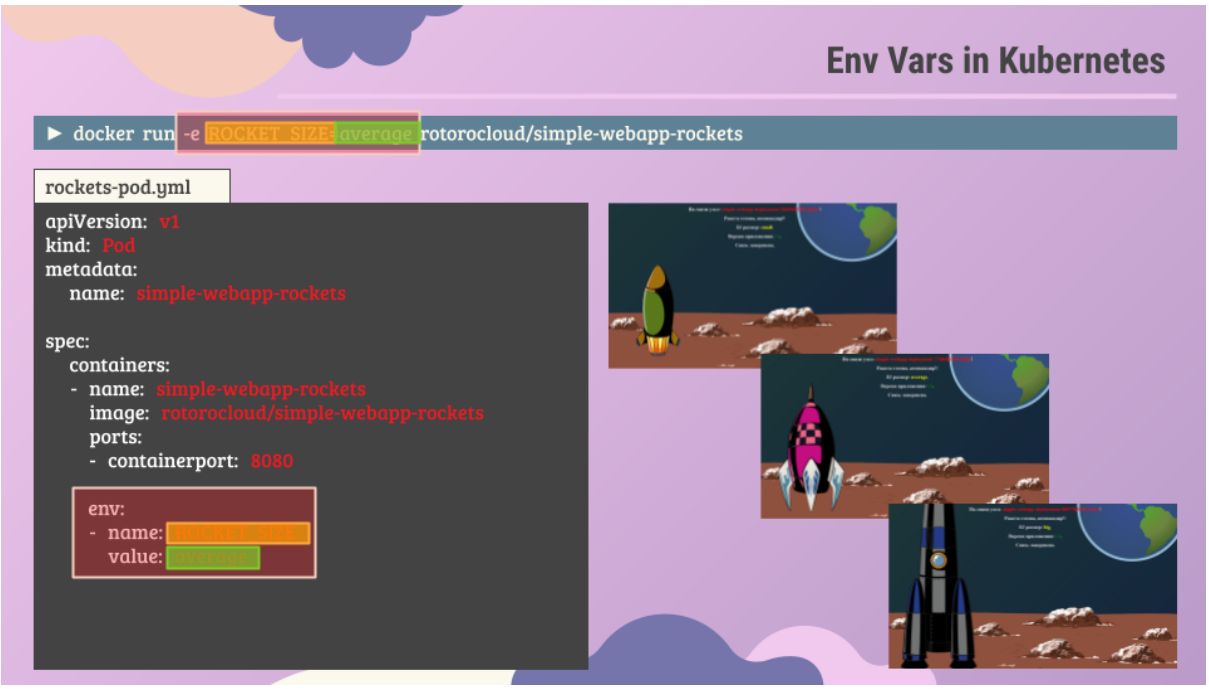
Ну вот и все о командах и аргументах в Kubernetes, посети раздел упражнений и попрактикуй разбор конфигураций контейнеров в части команд и аргументов.





Привет и добро пожаловать. В этой лекции мы увидим, как установить переменные среды в Kubernetes.

В мире Docker мы используем ключ -e для того, чтобы прокинуть какие-то данные в контейнер в качестве переменных окружения. Этот же подход реализован в Kubernetes, но в нем он более разнообразный.

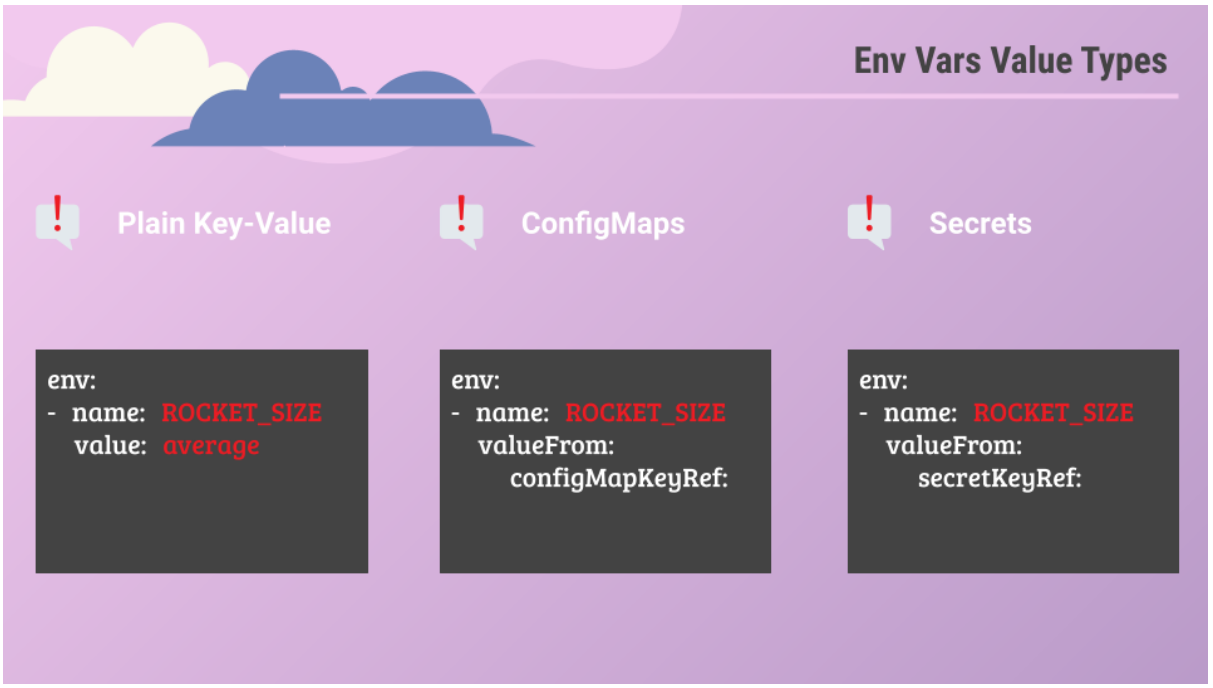


Данный файл определения POD использует тот же образ, что и команда docker выше.

Итак в Docker для установки переменной окружения мы запускаем контейнер с опцией -e, а в манифесте у нас свойство `env`, расположенное в секции контейнера.

ENV - это массив. Таким образом, каждый элемент в свойстве env начинается с тире, обозначающего элемент в массиве. У каждого элемента есть name, это название переменной и свойство value, соответственно ее значение.

Определив таким образом переменные среды, ты сможешь обращаться к ним из контейнера после запуска.



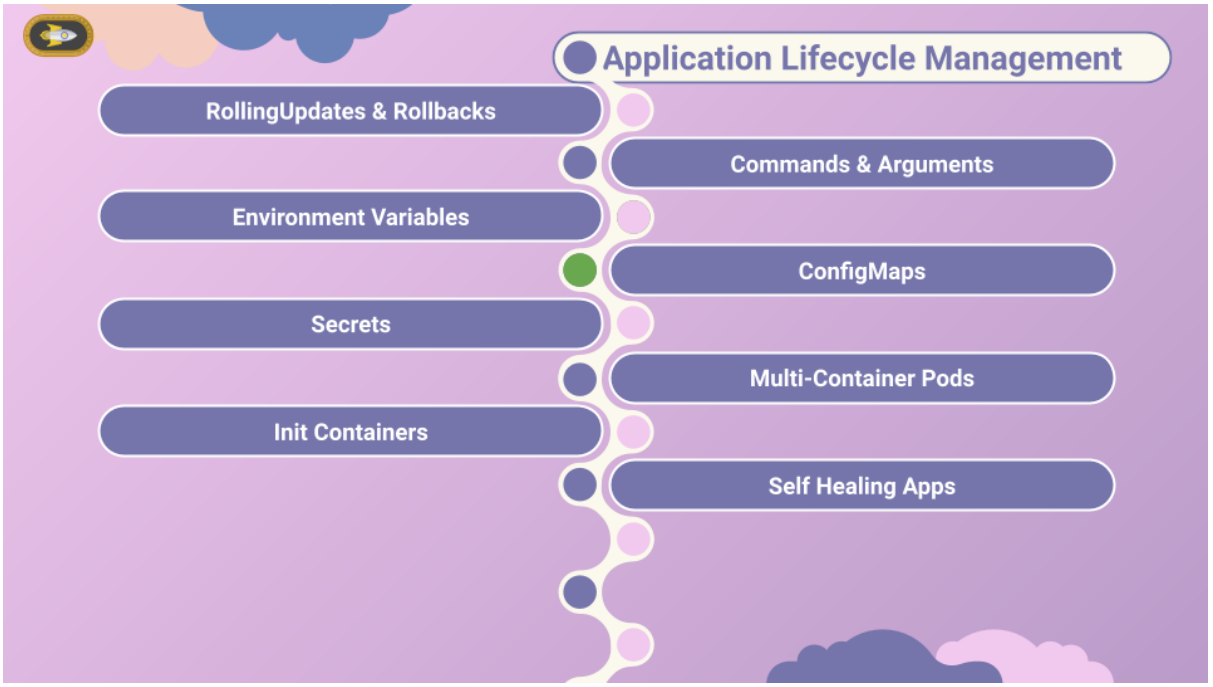
То, что мы только что увидели, было прямым способом указания переменных среды с использованием простого формата пары `ключ-значение`.

Однако есть и другие способы установки переменных среды, такие как использование configmaps и secrets.

Разница с этим способом задания заключается в том, что вместо прямого указания значений переменных с помощью свойства value, мы говорим откуда брать эти значения свойством `valueFrom`.
А далее ссылку на объект, где лежат данные.

Мы обсудим configmaps и secrets в деталях в следующих лекциях.

На этом лекция закончилась, увидимся на следующей.

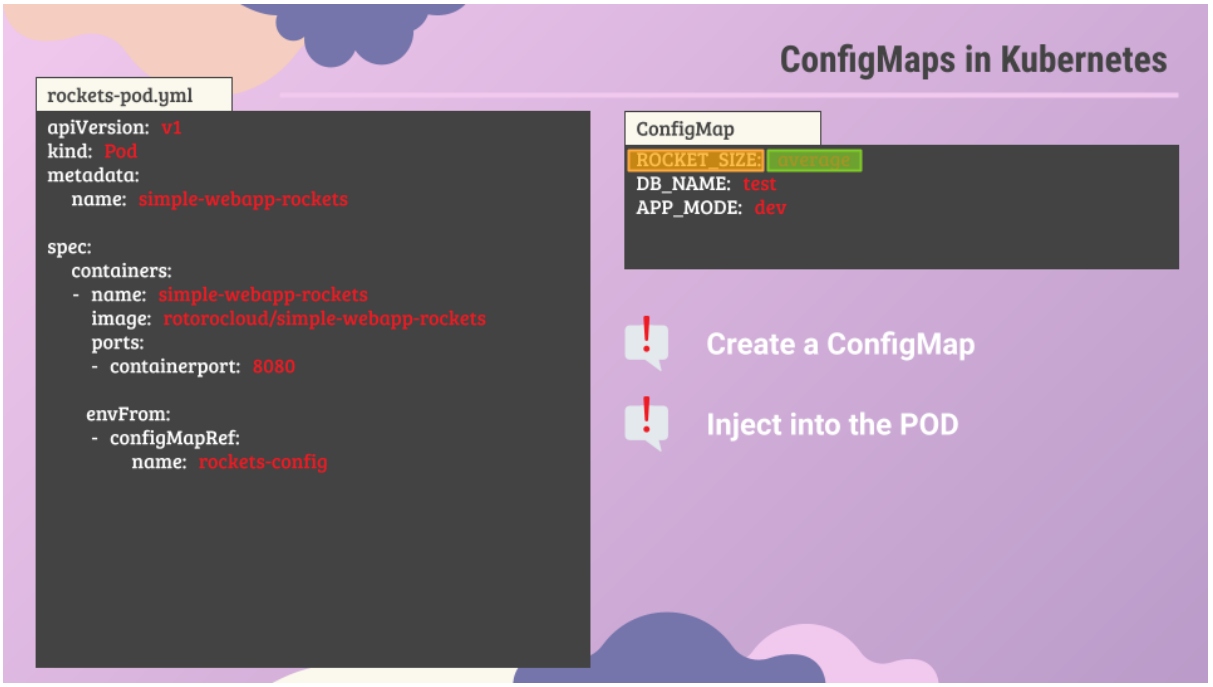


Привет и добро пожаловать на лекцию.
В этой лекции мы обсудим, как работать с данными конфигураций в Kubernetes.

В предыдущей лекции мы увидели, как определять переменные среды в файле определения POD.

Когда у тебя много разных манифестов, становится сложно управлять переменными сред, хранящимися в этих файлах определений.

Мы можем извлечь эту информацию из файла и централизованно управлять ею с помощью карт конфигураций - configuration maps. Или короче - ConfigMaps. Они используются в Kubernetes для передачи данных конфигураций в виде пар ключ-значение.

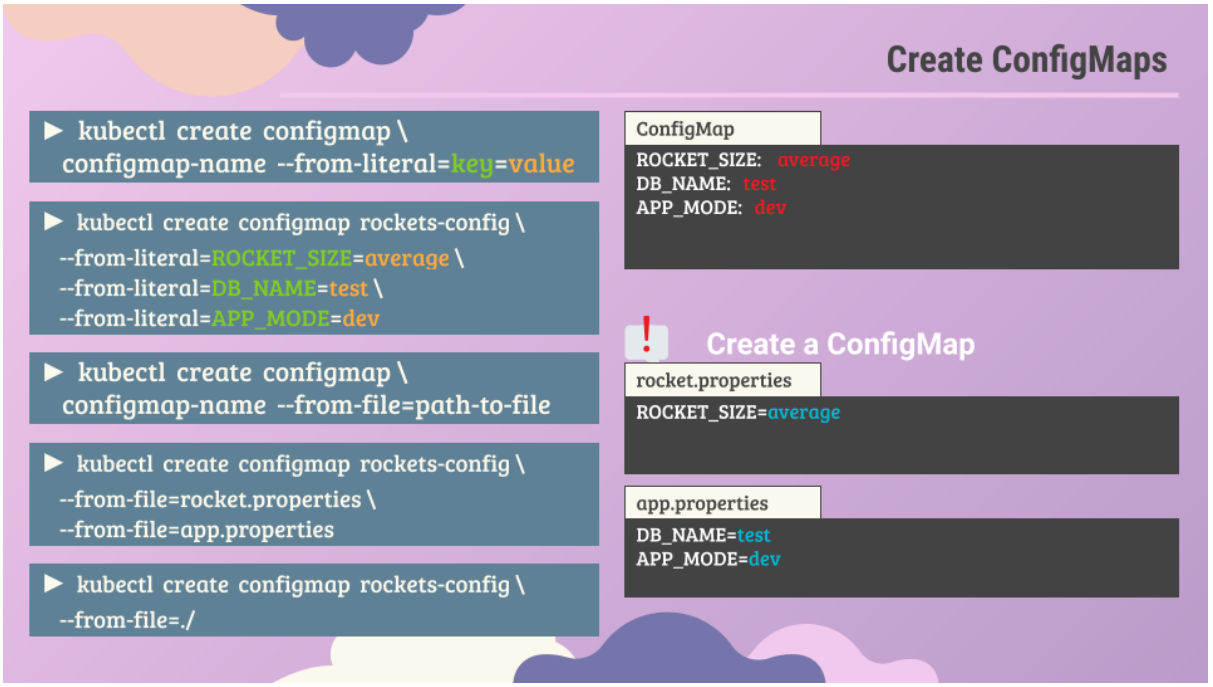


Когда POD создан, мы инжектируем в него ConfigMap. Таким образом, пары ключ-значение станут доступны в качестве переменных среды для приложений, размещенных внутри контейнера в POD.

Настройка ConfigMaps состоит из двух этапов:

- сначала создай ConfigMaps
- затем введи их в POD.

Как и у любого другого объекта Kubernetes есть два способа создания. Императивный метод - без использования файла определения ConfigMap и декларативный метод с использованием файла определения.



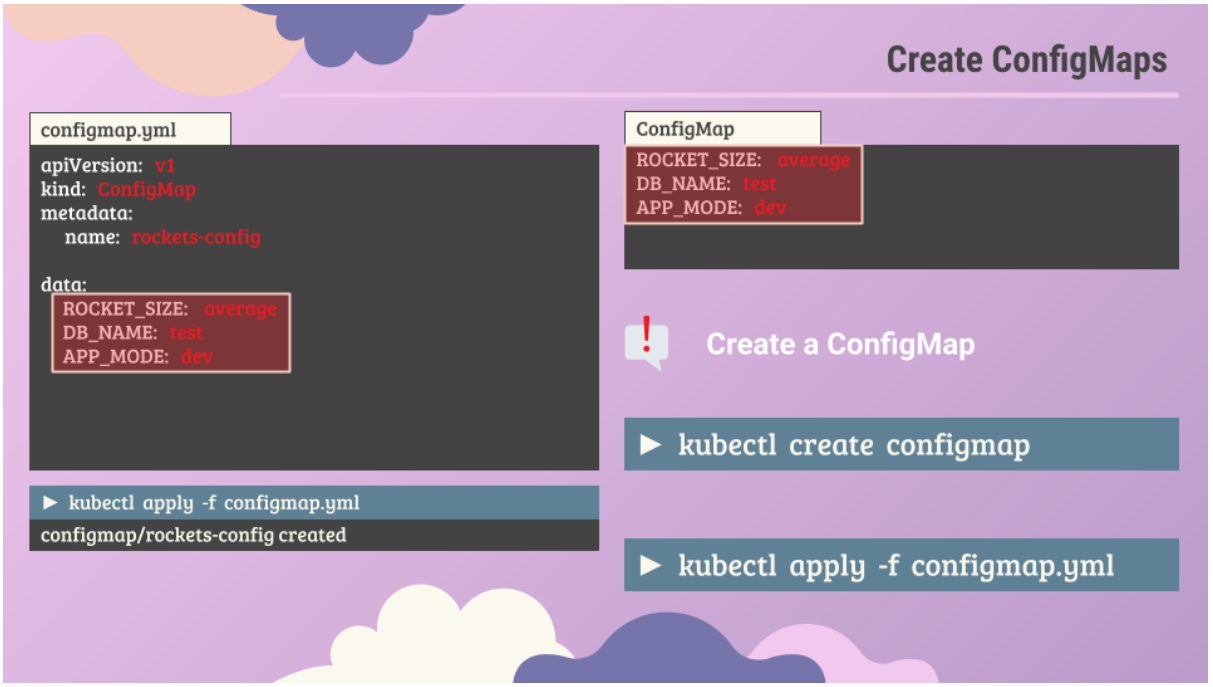
Пройдемся по первому методу, тут тоже можно действовать по-разному.
Если тебе не хочется создавать файлы-манифесты ConfigMaps, ты можешь просто использовать команду kubectl create configmap и указать необходимые аргументы. Ты можешь напрямую указать пары ключ-значение в командной строке. Чтобы создать configMap с заданными значениями, запусти команду kubectl create configmap, за которой имя configmap и параметр `-–from-literal`.

Этот параметр --from-literal используется для указания пар ключ-значение в самой команде. В этом примере мы создаем configmap с именем rockets-config с парой значений ключа ROCKET_SIZE=average. Можно добавить дополнительные пары ключ-значение, используя эту опцию несколько раз.

Однако это будет сложно, если у тебя будет слишком много элементов конфигурации. Для этого есть более удобный способ - ввод данных конфигурации через properties-файл.
Используй параметр `--from-file`, чтобы указать путь к файлу, содержащему необходимые данные.

Данные из этого файла будут прочитаны и сохранятся в созданном ConfigMap. При необходимости в ConfigMap можно залить несколько файлов или целый каталог.

Теперь давай рассмотрим декларативный подход.
Для этого мы создаем файл определения точно так же, как мы делали это для POD.



У файла есть apiVersion, kind, metadata, а вместо `spec` у нас есть `data`.
ApiVersion - v1, kind - ConfigMap. Под метаданными укажем название ConfigMap.

Мы назовем его rockets-config. Под данными укажем переменные конфигурации в формате ключ-значение.
Запустим команду kubectl apply и укажем имя файла конфигурации. Таким образом создается ConfigMap rockets-config с указанными нами значениями.

Мы можем создать столько ConfigMap, сколько нам нужно, это будет одинаково для всех.

Иногда тебе требуется программная генерация ConfigMaps по какому-то шаблону. Например, при использовании в конвейере cicd. Для разных сред у тебя может быть один шаблон и варианты кастомизации. Для этих целей в Kubernetes механизм configMapGenerator. Создай файл-шаблона kustomization.yml, сделай

kubectl apply -k

Эта команда возьмет данные из файла rocket.properties и создаст на основе этого файл-шаблона готовый configMap.
У команды kubectl kustomize широкие возможности, почитай о ней в документации.

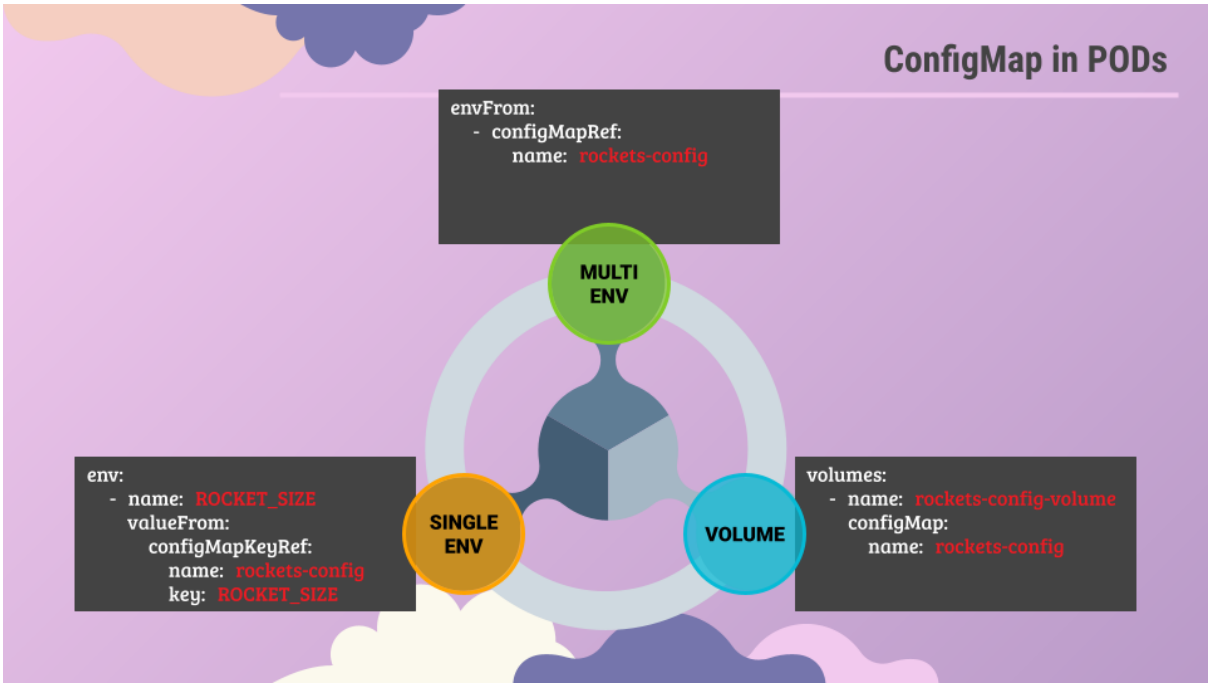
Ок, с примером ConfigMap для моего приложения ты уже знаком, здесь другой для mysql и еще один для redis. Поэтому важно правильно называть ConfigMap, поскольку ты будешь использовать эти имена позже.

Мы создали ConfigMap с именем rockets-config, для небольшого сетапа это допустимо, но для более сложного лучше использовать название с смыслом в несколько слов.

Команда kubectl describe configmaps перечислит ConfigMaps и позволит заглянуть в раздел данных.

Итак, мы создали ConfigMap, теперь давай перейдем к шагу 2, и вставим ее в POD с приложением. У меня есть простой файл определения POD, который запускает мое простое веб-приложение демонстрирующее ракеты. Для ввода в контейнер переменной среды добавим новое свойство в секцию container названием `envFrom`. Свойство envFrom представляет собой list, поэтому мы можем передать столько переменных среды, сколько требуется. Каждый элемент в этом list соответствует элементу ConfigMap. Укажем имя созданной ранее ConfigMap.

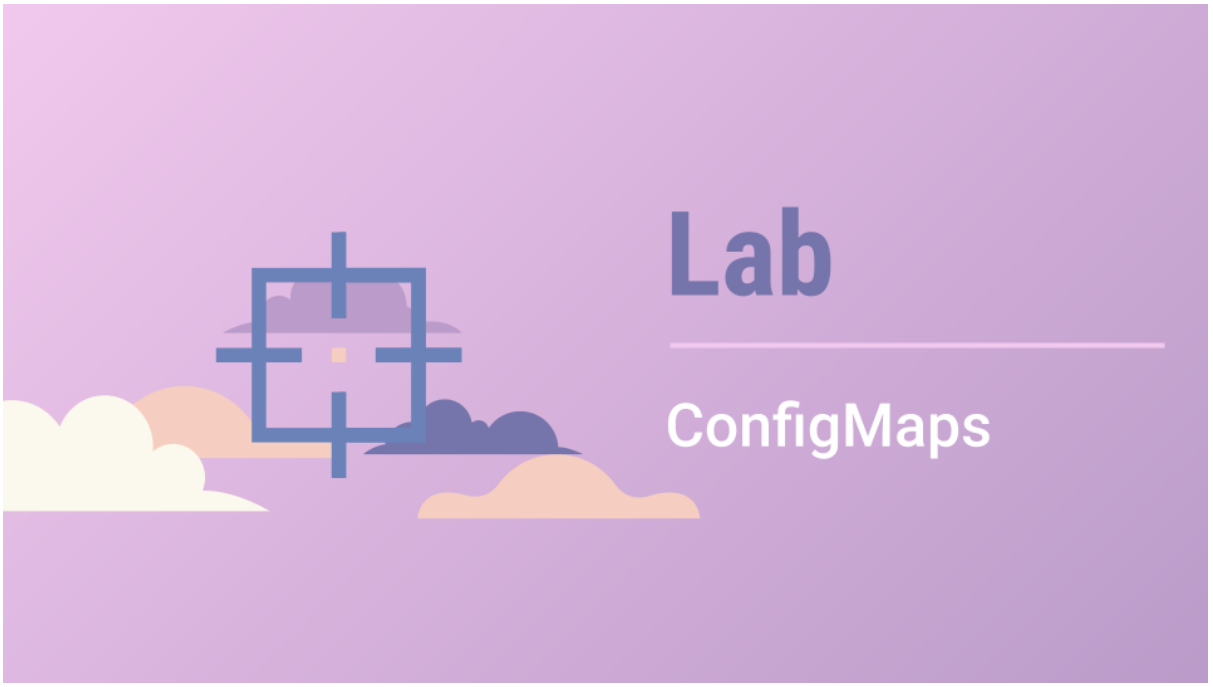
Вот так мы инжектируем эту конкретную ConfigMap. При создании файла определения POD теперь создается веб-приложение с средним размером ракеты, но если мы изменим значение в ConfigMap и пересоздадим POD, то вывод приложения изменится.

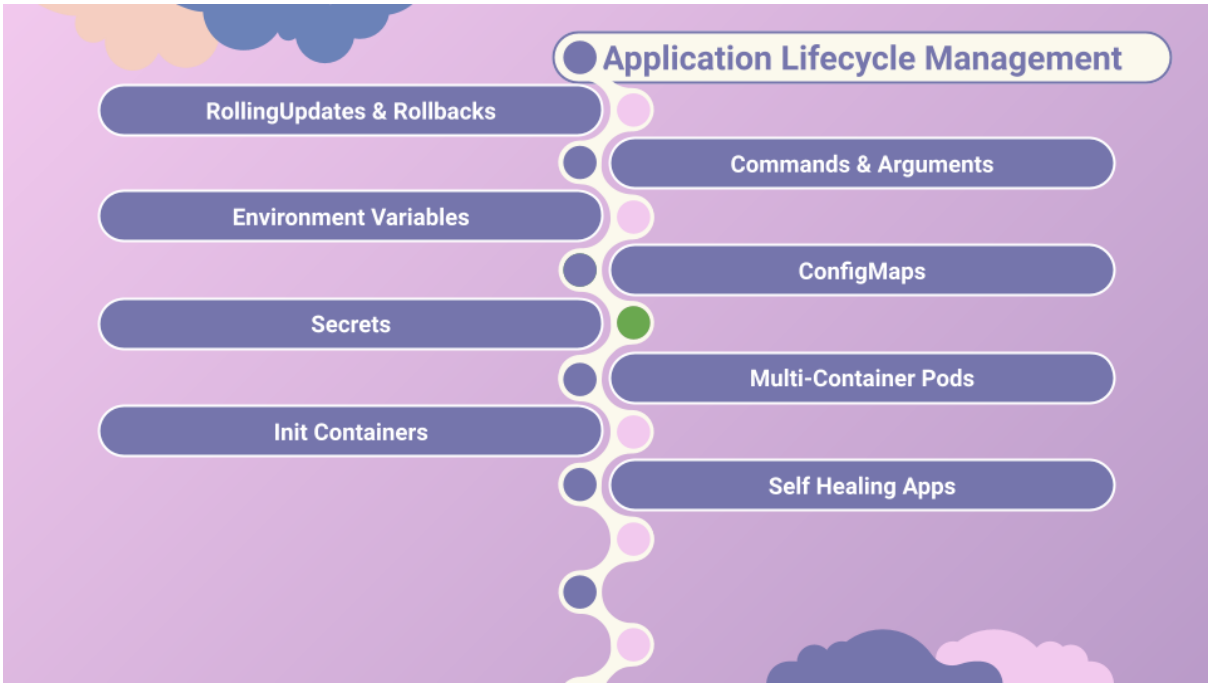


Мы только что увидели использование ConfigMaps для ввода переменных окружения. Существуют и другие способы внедрения данных конфигурации в PODs, например как отдельную переменную среды или можем подключить много данных в виде файлов с помощью механизма томов - volumes. Считается что использование volumes более секьюрно и удобно, т.к. при изменении ConfigMap данные в томе тоже обновятся сами.

Мы рассмотрим некоторые из этих вариантов в лабораторной, сопровождающую эту лекцию. Так что переходи туда и тренируй просмотр и траблешутинг переменных среды в Kubernetes.

А я жду тебя на следующей лекции.


Lab
ConfigMaps



Привет и добро пожаловать на лекцию о Secrets в Kubernetes.

Здесь у нас представлено простое веб-приложение на python, которое подключается к базе данных mysql. В случае успеха приложение отображает успешное сообщение.

Если мы внимательно посмотрим на код, то увидим, что имя пользователя, пароль и имя хоста жестко прописаны в коде.

Конечно, хардкодить это плохая идея.

Как мы узнали из предыдущей лекции, одним из вариантов было бы переместить эти значения в ConfigMap. ConfigMap хранит данные конфигурации в простом текстовом формате, в `plain text`.

Так что приемлемо было бы переместить имя хоста и имя пользователя в configMap. Но это определенно не подходящее место для хранения пароля.

Вот это место, где в игру вступают secrets. Эти объекты Kubernetes используются для хранения конфиденциальной информации, такой как пароли, токены, ключи и другие чувствительные данные.

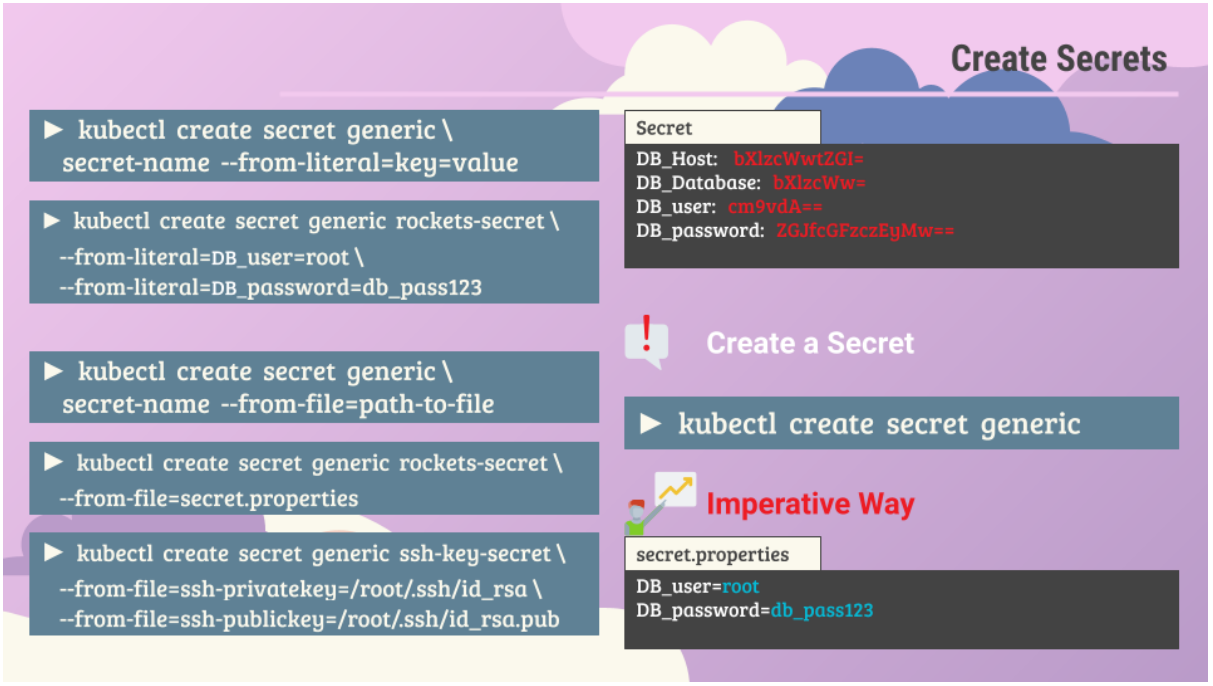
Они похожи на configMap, за исключением того, что хранятся в закодированном или хешированном формате. Но из контейнера все же эти значения выглядят уже как обычно, без кодировки.

Как и в случае с ConfigMaps, работа с secrets состоит из двух этапов:
- создания secret в начале
- введение его в нужный контейнер

Есть два способа создать secret.

Императивный подход - без использования файла определения secret и декларативный способ с использованием файла определения.

Императивной командой мы можем напрямую указать пары ключ-значение, прописав их в командной строке.



Чтобы создать secret из заданных напрямую значений, запусти команду

kubectl create secret generic

Далее в команде следует имя секрета и опция `—from-literal`.
Этот параметр --from-literal используется для указания пар ключ-значение в самой команде.

В этом примере мы создаем secret с именем rockets-secret и двумя парами ключ-значение: DB_user=root и DB_password=db_pass123. Как видишь, это похоже на ConfigMaps, параметр —from-literal можно указать несколько раз. Однако если у тебя много секретной информации, это станет очень громозкой и сложной конструкцией. Здесь лучше воспользоваться другим способом - секретные данные через файл.

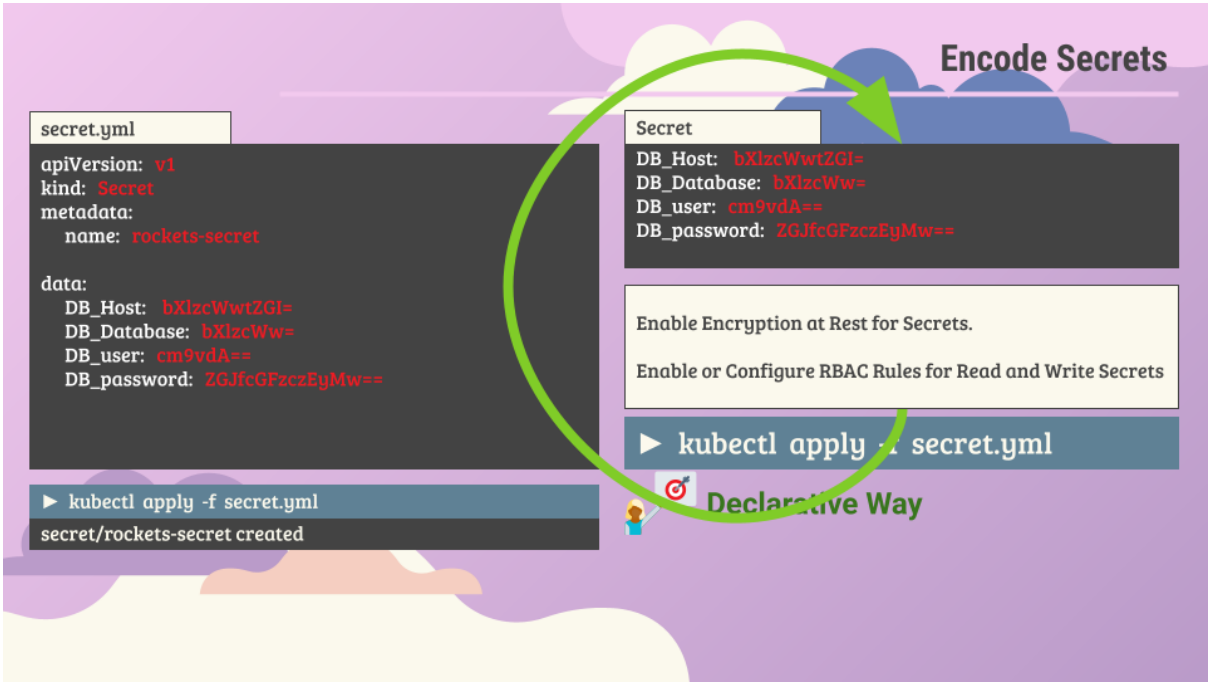
Используй параметр `—from-file`, чтобы указать путь к файлу, содержащему необходимые данные в формате properties. Данные из этого файла читаются и сохраняются под именем secret. Можно указать несколько файлов или задать полю secret значение из файла, как ты видишь. Это, например, удобно для хранения ssh-ключей.

Теперь давай рассмотрим декларативный подход.

Для этого мы создаем файл определения, точно так же, как мы делали это для ConfigMap. Файл имеет apiVersion, kind, metadata и data.

ApiVersion: v1, kind: Secret. Под метаданными укажем название secret.

Назовем его rockets-secret. В поле данные добавим секретные данные в формате ключ-значение.



Я уже упоминал одну особенность secrets - они хранят чувствительные данные в закодированном формате.

Здесь мы указали данные в виде обычного текста, что не очень безопасно. Поэтому при создании secret с декларативным подходом мы должны указывать секретные значения в хешированном формате.

Тут небольшое отступление. Кодировка - это не шифрование. Данные могут быть приведены в читаемому виду любым, кто получил к ним доступ. Поэтому имей в виду, что информацию в secrets можно дополнительно зашифровать и ограничить доступ к secret с помощью правил RBAC.

Теперь мы можем создать свой secret, применив манифест.

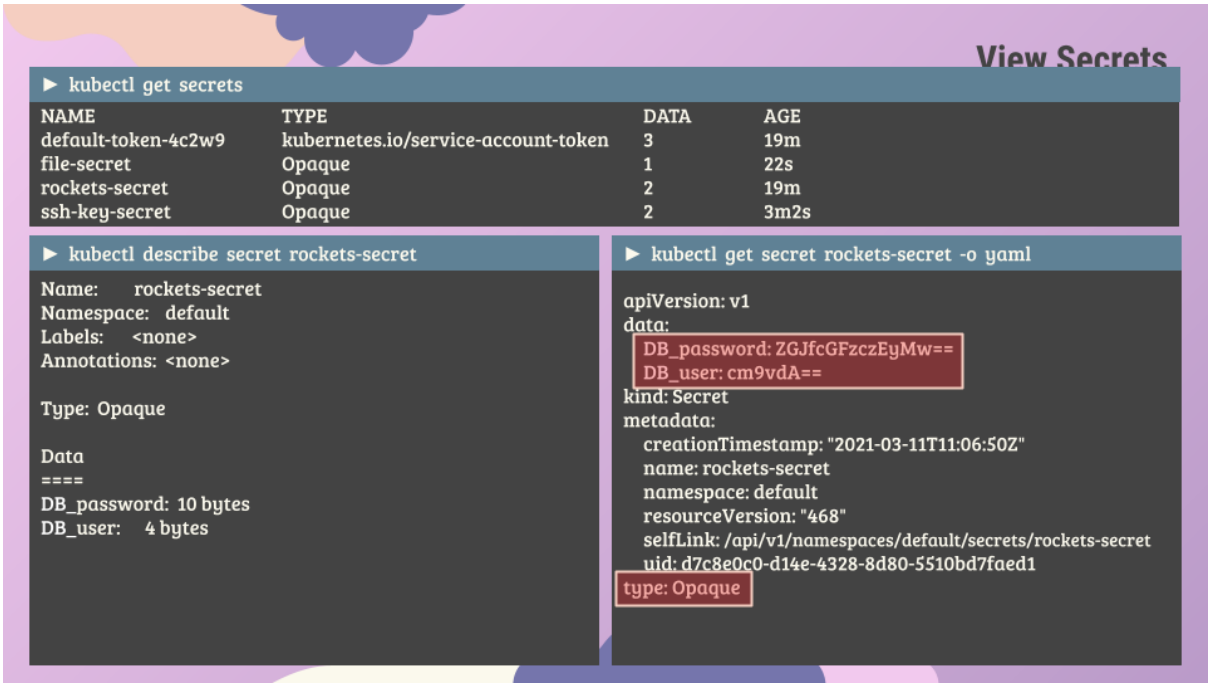
Но прежде тебе требуется сконвертировать данные в такую закодированную форму. Как преобразовать данные из обычного текста в такой закодированный формат?

На машине с Linux можно воспользоваться командой echo –n, за которой следует текст, который требуется преобразовать. В данном случае `root`. Далее передать его по pipe утилите base64.

Чтобы просмотреть секреты, запусти команду kubectl get secrets.

В выводе перечислены все созданные secrets. Как видишь помимо наших здесь есть другие, которые Kubernetes создал для внутренних целей.

Чтобы просмотреть дополнительную информацию о созданном secret запустите команду kubectl describe secret.



Она покажет атрибуты секрета, но скроет сами значения.
Для просмотра значений запусти команду kubectl get secret с выводом в формате YAML, используя параметр –o, как ты видишь здесь.

Теперь ты можешь увидеть эти хешированные значения, осталось их только декодировать по 64-основанию. Используй ту же команду base64, которая была ранее для кодирования, но на этот раз с опцией `--decode`.

Ок, мы создали секрет. Теперь перейдем к шагу 2 - встроить secret в POD.

Здесь у меня есть простой файл определения для POD, который запускает мое приложение.

Чтобы внедрить переменную среды, добавь в контейнер новое свойство с именем envFrom.

Свойство envFrom представляет собой список, поэтому мы можем передать столько переменных среды, сколько требуется. Каждый элемент в этом списке соответствует секретному элементу. Укажи название secret, который мы создали ранее.

Создание ссылки на secret в файле определения POD делает эти конфиденциальные данные доступными в качестве переменных среды для приложения в контейнере.

Мы только что увидели внедрение секретов в качестве переменных среды в POD.
Есть и другие способы ввести secret в POD.

Ты можешь также частично вводить secret, как отдельные переменные среды с помощью конструкции `secretKeyRef`, или ввести secret полностью в виде файлов из смонтированного тома.

Для подключения секретной информации выбирай способ, который тебя наиболее устраивает.

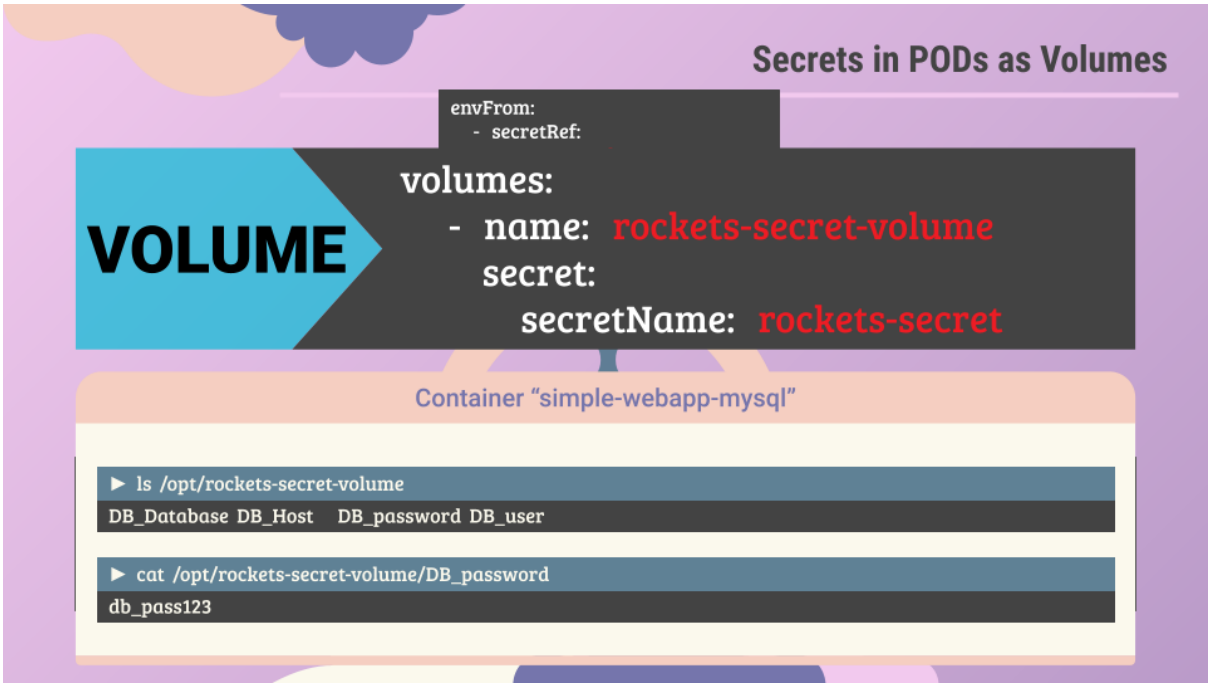
В случае тома, каждый атрибут секрета будет представлен в отдельном файле, название файла будет именем атрибута в secret, а содержимое - значением этого секретного поля.

В этом случае, поскольку у нас есть четыре атрибута в нашем secret, то в смонтированной внутри контейнера папке создадутся четыре файла с названиями атрибутов.

Если мы посмотрим на содержимое `DB_password`, то мы увидим в нем пароль в открытом виде. Механизм здесь такой же, как в Configmap, и если мы изменим secret, эти файлы также автоматически обновятся.

Напоследок несколько мыслей о хранении секретных данных.

Помни, что secrets кодируют данные в формате base64. Любой, у кого есть secret в кодировке base64, может легко его расшифровать. Таким образом, секреты нужно считать небезопасными.



Концепция безопасности secrets в Kubernetes немного сбивает с толку. Сообщество Kubernetes исходит из того, что secrets `более безопасный вариант` для хранения конфиденциальных данных. Они безопаснее, чем хранение в виде обычного текста, поскольку снижают риск случайного раскрытия паролей и других чувствительных данных.

И здесь небезопасен не сам секрет, а методы, связанные с ним.

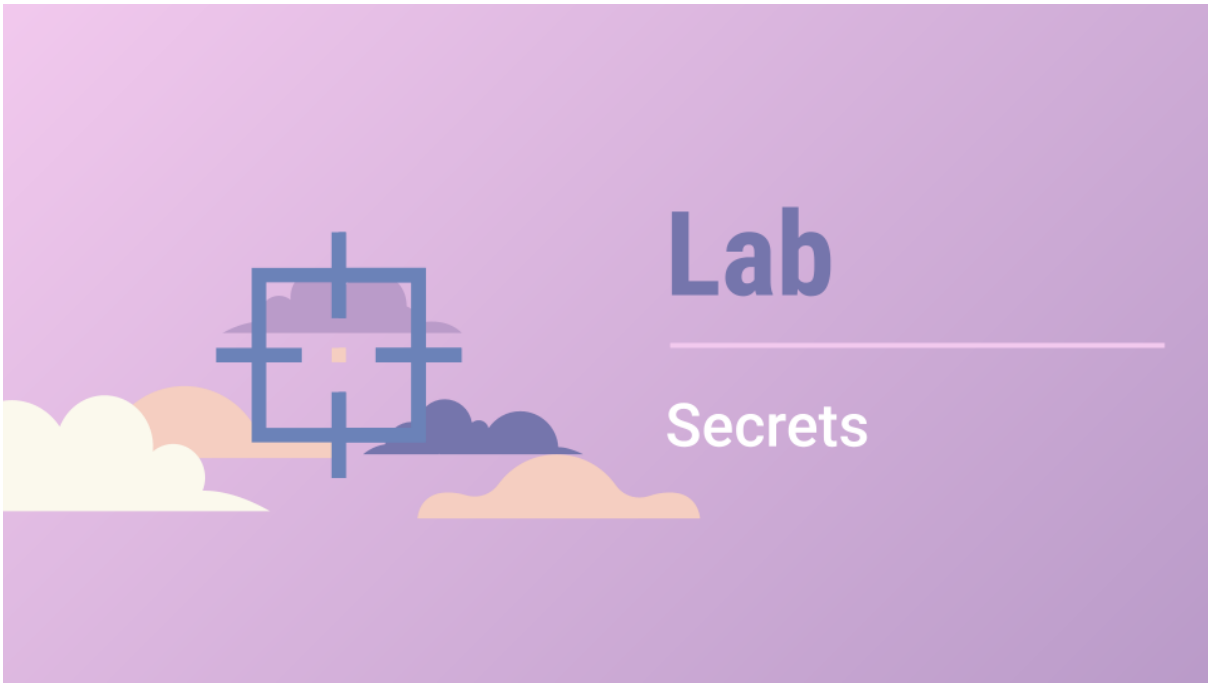
Секреты не зашифрованы, поэтому в этом смысле это не безопаснее. Однако некоторые продвинутые методы использования secrets делают его более безопасным. Мы поговорим об этом в разделе security далее в курсе, но давай кратко пробежим best practices:

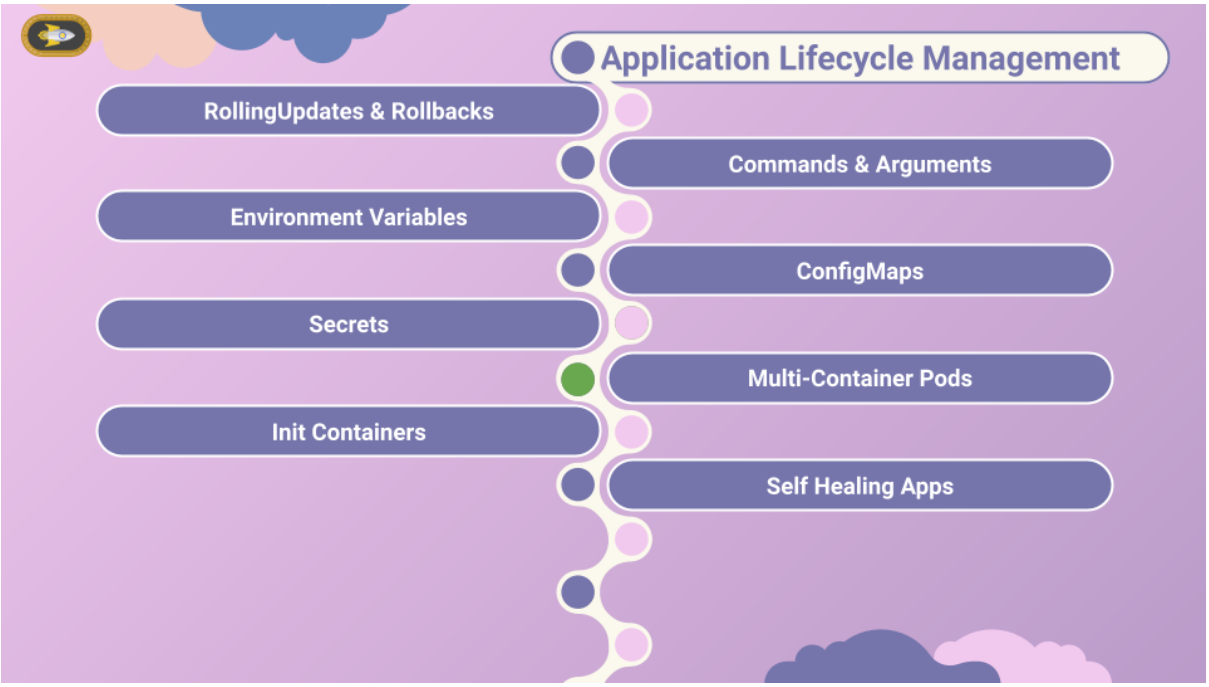
- перестань отслеживать secrets в SVC-репозиториях, организуй процесс контроля изменений для secrets иначе
- используй шифрование при хранении, чтобы они хранились в зашифрованном виде в ETCD.
- отмени доступ любому пользователю для secrets-API для действий `watch` и `list`
- пусть только PODs, которым необходимы эти конкретные данные получают к ним доступ
- kubelet хранит секрет в tmpfs, чтобы секрет не записывался на диск. Твои приложения также не должны сохранять их в постоянной памяти и не показывать в логах
- использование SSL/TLS
- аккуратнее с root привилегиями в кластере

Есть также специальные решения, это AWS Secrets Manager, Google Cloud Platform KMS, Azure Key Vault для клауд-вендоров и опенсорсные вещи, вроде Helm Secrets, HashiCorp Vault, Conjur, которые помогут тебе в хранении. Я надеюсь, что в скором

времени мы поговорим об этом подробно в курсе сертифицированного специалиста по безопасности Kubernetes от rotoro.cloud.

И это все в лекции про secrets. Впереди упражнения. Потренируйся работать с secrets.





Привет и добро пожаловать в эту лекцию, посвященную много-контейнерным PODs.

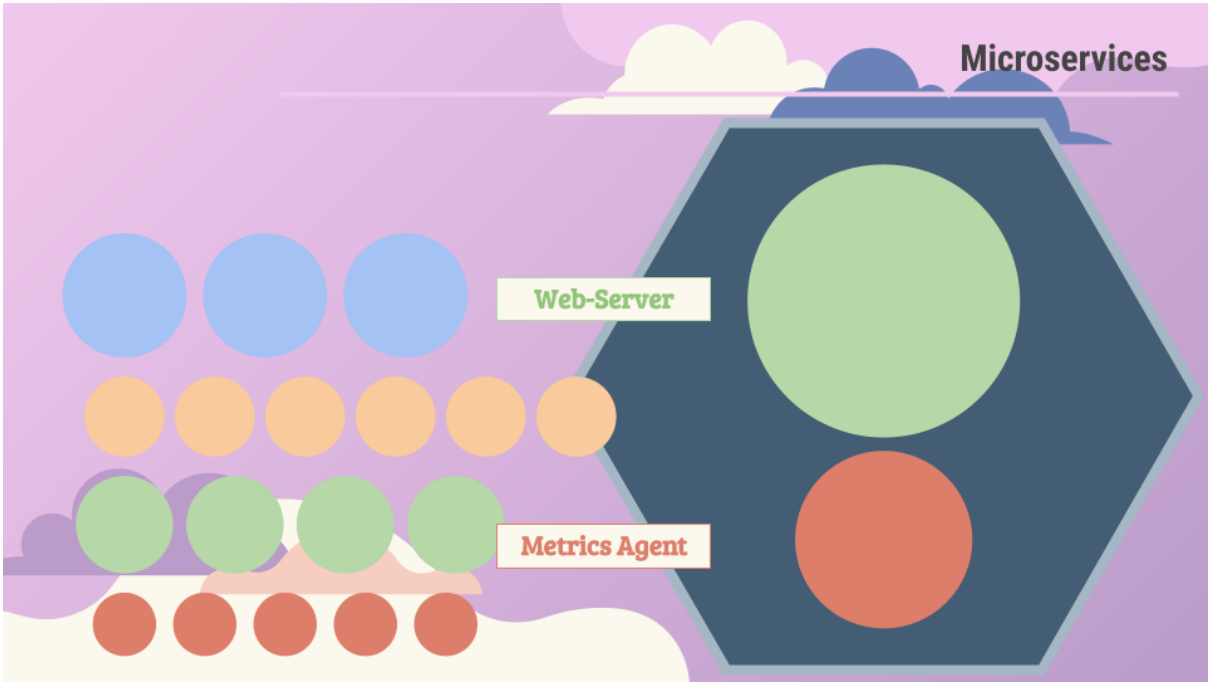
Идея разделения большого монолитного приложения на суб-компоненты, известные как микросервисы, позволяет нам разрабатывать и развертывать наборы небольших, независимых друг от друга компонентов системы.

Эта архитектура может помочь нам масштабироваться вверх и вниз, а также изменять каждый кусочек этого пазла по мере необходимости, не изменяя сразу все приложение. За счет этого мы можем двигаться быстрее в разработке, отладке и проверке новых гипотез.

Часто в наших задачах нам могут понадобиться две службы для совместной работы, такие как веб-сервер и служба регистрации метрик контейнера. Нам нужен один экземпляр агента на каждый экземпляр веб-сервера, и они должны быть соединены друг с другом.

Мы не хотим объединять код этих двух служб, поскольку каждая из них нацелена на разные функции и может использоваться где-то еще.
Также мы знаем, что нам наиболее просто и эффективно будет поддерживать эти службы, если они останутся как есть, т.е. разделенными. А при объединении же кода вырастет сложность сопровождения и упадет скорость внедрения новых версий. Т.о. нам нужен только функционал этих двух приложений только для их совместной работы в текущем варианте развертывания.

Итак, нужен один агент для каждого экземпляра веб-сервера, и эти пары должны легко масштабироваться вместе. Значит, эти контейнеры должны иметь один и тот же жизненный цикл, что означает, что они создаются вместе и уничтожаются вместе.

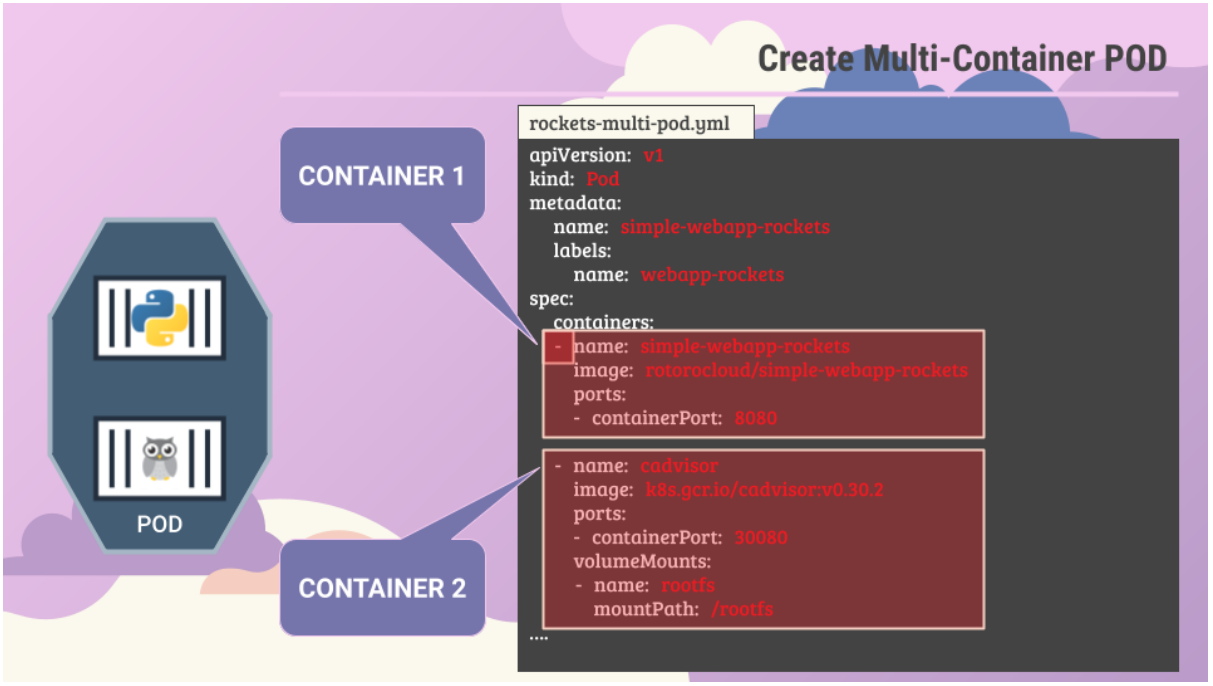


Также они должны использовать одно и то же сетевое пространство, что означает, что они могут найти друг друга на общем localhost. И еще было бы очень удобно, если бы они имели доступ к одним и тем же томам хранилища.

Это все есть в POD с несколькими контейнерами.

Таким образом, нам не нужно устанавливать совместное использование томов или настраивать services, чтобы обеспечить связь между контейнерами, составляющих много-контейнерный POD.

Как это делается? Добавим новый раздел в секцию containers в файл определения POD.



Как ты помнишь, что раздел контейнеров в спецификации POD является массивом.

И причина, по которой это массив, состоит в том, чтобы разрешить нескольким контейнерам работать в одном POD.

В этом случае мы добавляем новый контейнер с именем cadvisor к нашему основному контейнеру с приложением.

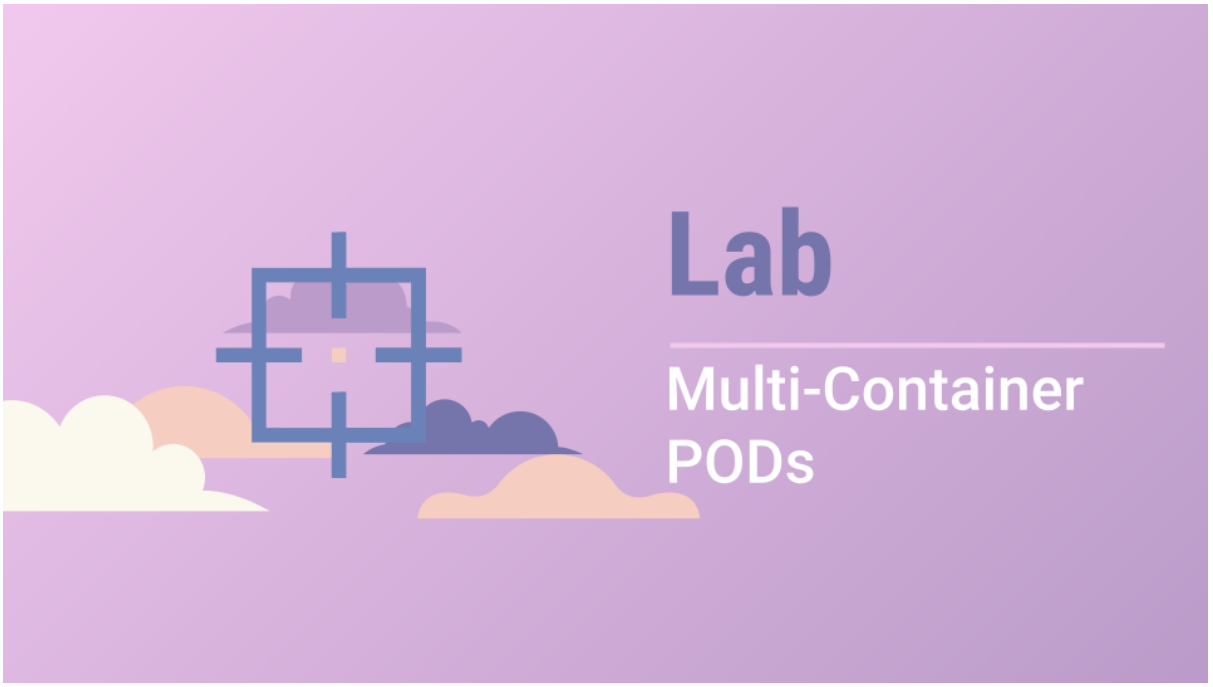
Теперь у нас здесь описание двух контейнеров.

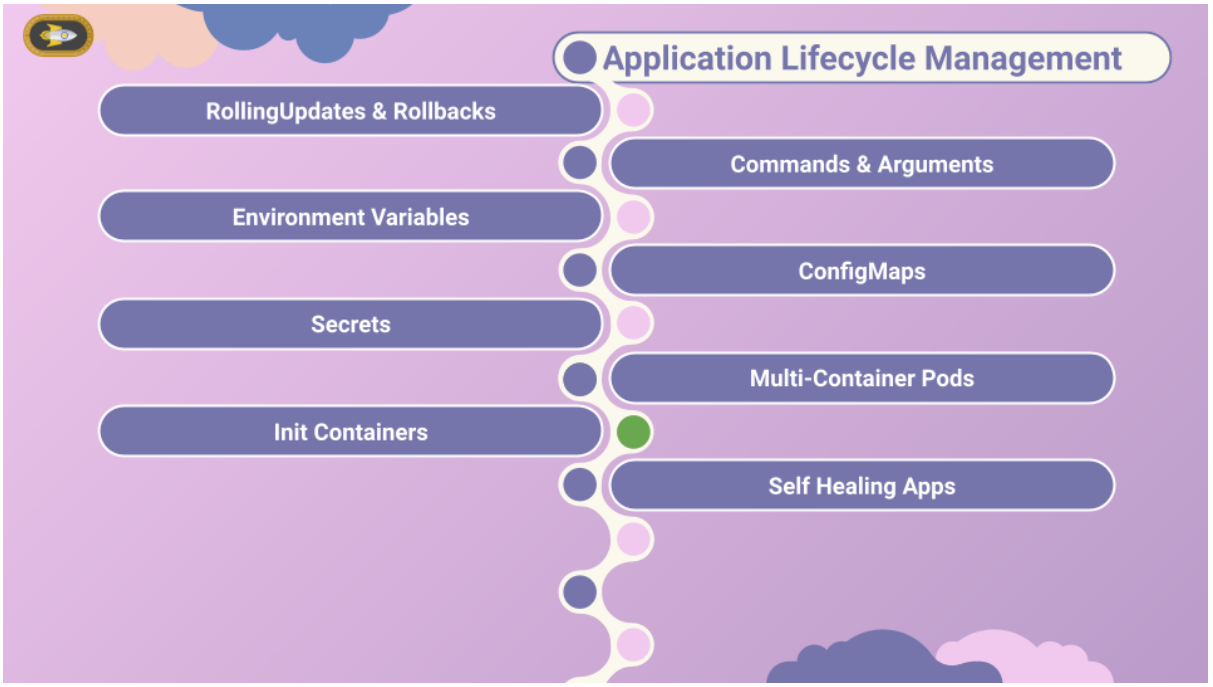
Существует три наиболее распространенных шаблона проектирования мульти-контейнерных PODs с учетом возложенных на них ролей.

Тот, который мы использовали сейчас называется Sidecar, и он используется для сбора метрик, перезагрузки настроек приложения в соседнем контейнера и т.д. Существуют и другие паттерны дизайна, например, Adapter и Ammassador.

Это больше относится к разработке приложений и мы будем глубже рассматривать в курсе разработчика приложений в Kubernetes.

Ну вот и все для этой лекции. Перейди в раздел упражнений и поработай с multicontainer PODs. Увидимся на следующей лекции.


Lab
Multi-Container
PODs



Привет, и добро пожаловать на лекцию, где мы обсудим init containers.

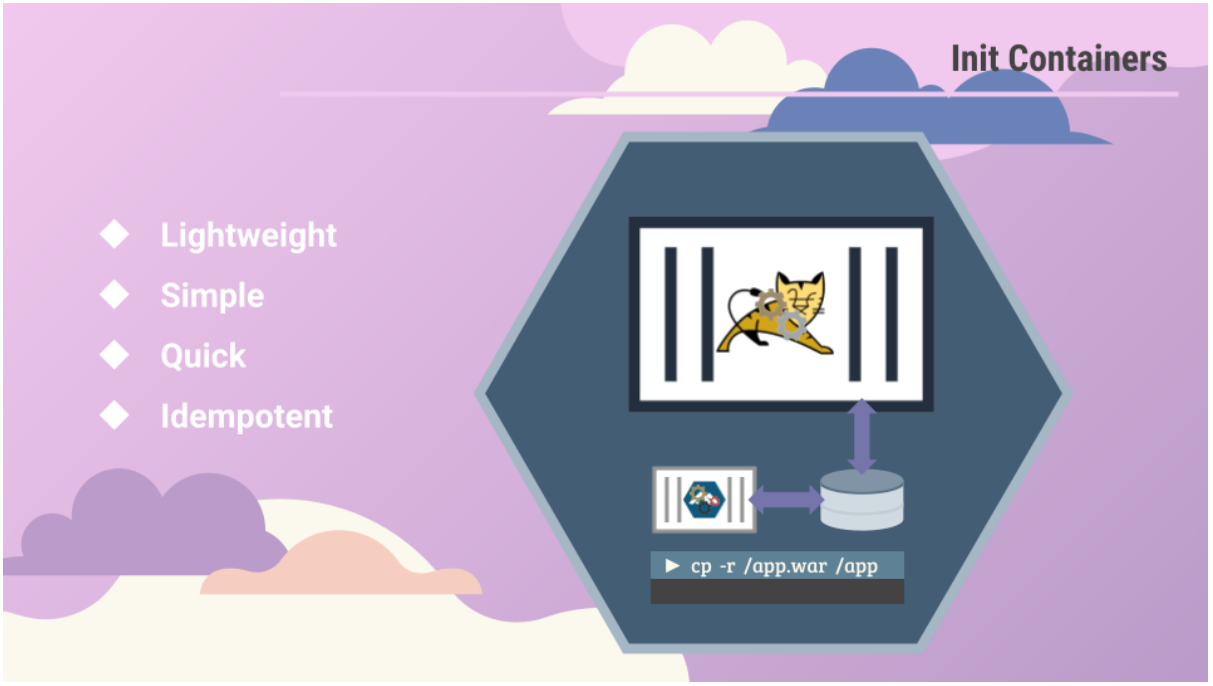
В POD с несколькими контейнерами ожидается, что каждый контейнер будет запускать процесс, который остается активным в течение всего жизненного цикла POD.

Например, в мульти-контейнерном POD, о котором мы говорили ранее, в есть веб-приложение и агент мониторинга ресурсов.

Зная о преимуществах PODs с несколькими контейнерами, мы полагаем что процесс, запущенный в контейнере агента останется активным, пока работает веб-приложение. А если какой-либо из root-процессов этих контейнеров выходит из строя, судьба контейнеров будет общей - POD перезапустится.

Но иногда тебе может понадобиться запустить процесс, который выполняется в контейнере-соседе, но результат этой работы будет использоваться основным приложением.

Например, процесс, который извлекает код или двоичный файл из репозитория или процесс, ожидающий запуска внешней службы или базы данных, необходимых основному контейнеру и т.д.



Эта задача будет запускаться только один раз при первоначальном создании POD, а в дальнейшем этот контейнер не требуется для функционирования основного. Вот тут-то нам и пригодятся init containers.

Раз эти контейнеры живут совсем недолго, мы можем сказать, какими они должны быть:

- легковесными
- простыми
- быстрыми
- идемпотентными

Ок, а что будет, если процесс в init container не найдет файл и завершиться с ошибкой? POD будет перезапущен.

Init container настраивается в POD, как и все другие контейнеры, за исключением того, что он указывается внутри раздела init containers, как ты видишь здесь.

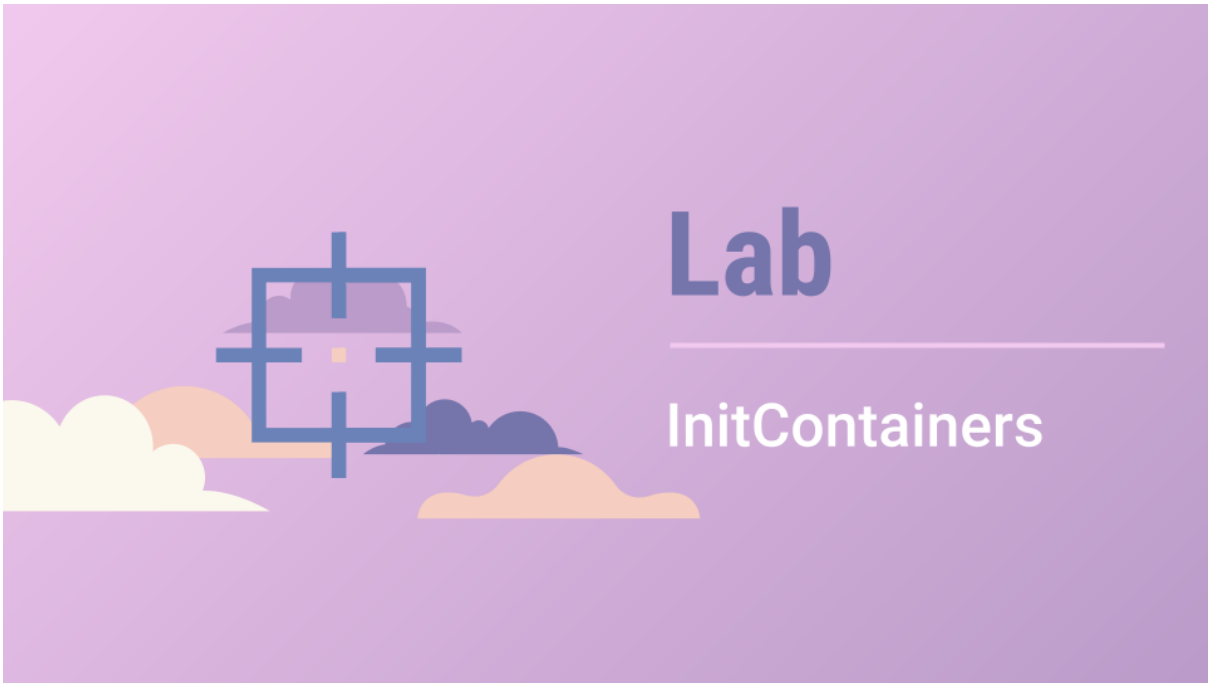
Когда POD создается, первым запускается init container, и процесс в init container должен завершиться до запуска реального контейнера, в котором размещено приложение.

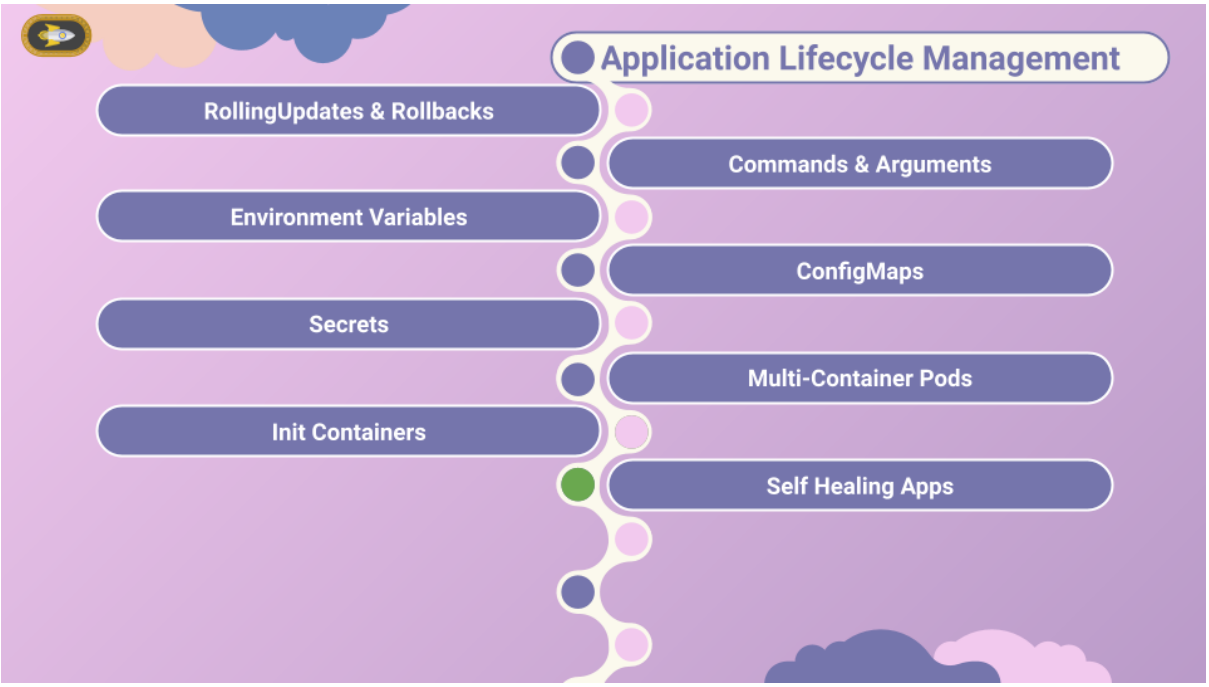
Мы также можем настроить несколько таких init containers. Тогда они будут запускаеться друг за другом последовательно.

Если какой-либо из init containers не может быть завершен, Kubernetes повторно перезапускает POD до тех пор, пока цепочка init containers не завершится успешно.

Ну разумеется, если ты не установил restartPolicy в Never.

А на этом все, переходи к практике с init containers, а я жду тебя в следующей лекции!





Привет и добро пожаловать. Это необязательная лекция для курса CKA, тут будет много общих рассуждений.

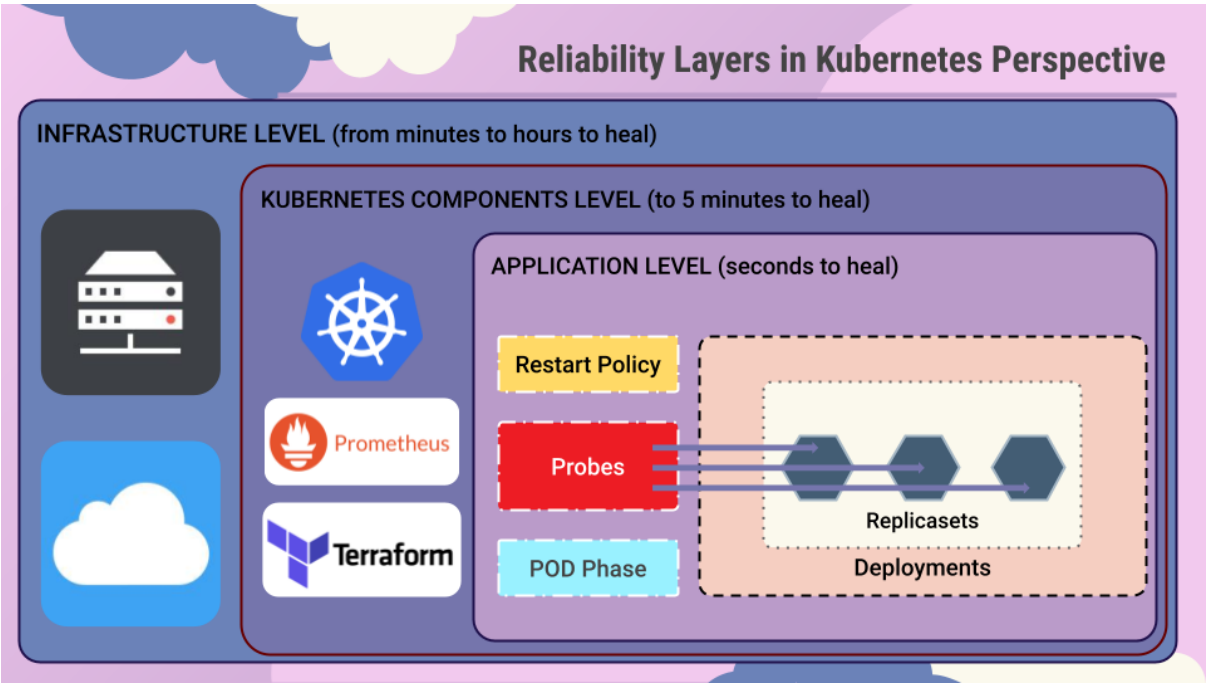
Можно нескромно сказать, что Kubernetes является королем самовосстановления.

Он позволяет запускать приложения надежным, высокодоступным и стабильным способом в нескольких инфраструктурах. Фактически, Kubernetes имеет уникальные возможности для выполнения этого обещания. Но тот факт, что у него есть возможность, не означает, что они доступны по умолчанию. И еще говорят, что свита делает короля. Поэтому и Kubernetes не может обойтись без помощников.

Когда говорят о самовосстановлении, обычно имеют в виду уровень приложения, но, далее мы увидим, этого недостаточно.

Итак первый уровень - уровень приложения. Kubernetes обеспечивает самовосстановлением через ReplicaSets. Контроллер репликации помогает обеспечить автоматическое воссоздание PODs при сбое приложения в POD. Это помогает обеспечить постоянную работу достаточного количества реплик приложения. Тем не менее здесь ОЧЕНЬ многое зависит от разработчика приложения и насколько хорошо контейниризировано приложение.

Как мы можем повлиять на самовосстановление с этого уровня, кроме деплоя качественных образов? Kubernetes предоставляет дополнительную поддержку для проверки работоспособности приложений, работающих в POD, и принятия необходимых действий с помощью внутренних проверок - `probes`. Мы не будем погружаться в их настройку, они не требуются для экзамена CKA. Это темы для экзамена Certified Kubernetes Application Developers (CKAD), которые и будут там рассматриваются.



Для анализа ситуации на уровне приложений нам также помогают механизмы Pod phase и Pod conditions. А для ремедиации проблем в какой-то мере может помочь настройка Container restart policy.

Теперь поднимемся выше на уровень компонентов. На каждом узле работает ряд компонентов Kubernetes, необходимых для поддержания работоспособности Kubernetes. Сюда входят kubelet, kube-proxy и среды выполнения контейнеров. Если один из этих компонентов выходит из строя, узел может перестать отвечать. Хотя Kubernetes примет к сведению, здесь могут пройти минуты, прежде чем он это сделает. В это время мы можем столкнуться с перебоями в обслуживании.

Правильного отката с помощью таких инструментов, как Terraform или kops, недостаточно. Нам нужен компонент, который постоянно будет отслеживать состояние всех компонентов Kubernetes, а также решение обеспечивающее быстрое восстановление с минимальным влиянием на остальную часть кластера.

Ок, переместимся еще на ступеньку выше и охватим взглядом нашу инфраструктуру. В любой момент виртуальная машина или диск могут выйти из строя. Хотя Kubernetes переназначит PODs наших приложений, как только обнаружит, что узел больше не доступен (что может занять 5 минут), он не сможет самостоятельно развернуть новый узел.

Все, что он может сделать, - это перетасовать ресурсы в рамках существующей инфраструктуры. Если свободных мощностей не хватит, не повезло.

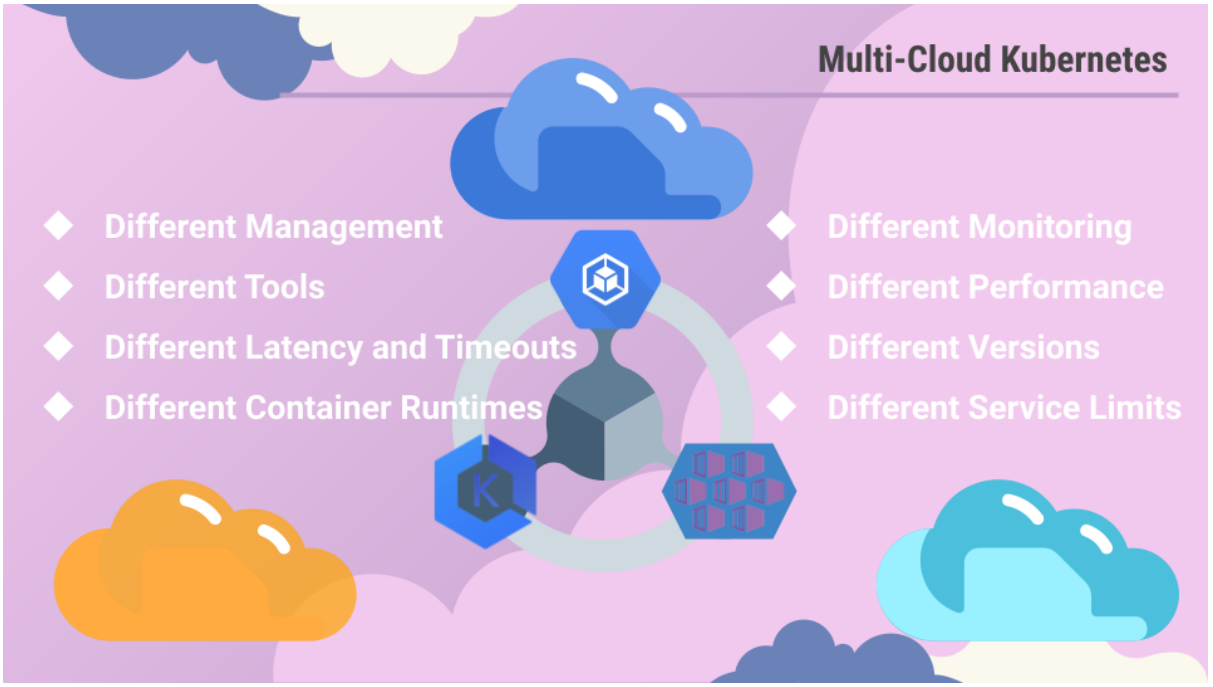
Чтобы предоставить новую инфраструктуру, когда это необходимо, тебе понадобится внешний механизм, обеспечивающий самовосстановление нод.

При использовании облака можно использовать облачные инструменты для мониторинга виртуальных машин и автоматического перезапуска или запуска новой в случае отсутствия healthchecks с узла. В Azure это могут быть scale sets. В AWS и GCP это autoscaling groups. Все, что нам нужно сделать, это убедиться, что система, которая предоставляет кластеры Kubernetes, должным образом использует эти возможности.

Локально с VMware или на baremetal нам понадобится еще какая-то внешняя система, для упреждающего мониторинга инфраструктуры и принятия мер для ремедиации, когда это необходимо. Например, с помощью VMware система может подготовить новый виртуальный узел, если старый больше не отвечает и т. д.

Как мы видели, Kubernetes обеспечивает самовосстановление приложений в PODs, и если один POD выходит из строя, Kubernetes перезапускает новый. Однако, если вся нода падает, Kubernetes обычно не может запустить новую.

Получается, что с перспективы самовосстановления инфраструктура может превратиться в самое слабое звено в цепочке, ставя под угрозу надежность наших приложений. Это потому, что надежность наших кластеров зависит от базовой инфраструктуры. И это означает, что даже лучшее решение для управления Kubernetes не может защитить нас от плохой подготовки инфраструктуры.



При использовании managed Kubernetes из облака тебе не нужно беспокоиться о надежности и самовосстановлении - облако заботится об этом за нас.

Но поскольку Multi-Cloud или Hybrid-Cloud развертывания становятся все более распространенными, надежность становится уже твоей ответственностью, даже если ты используешь разного рода фреймворки и инструменты вроде Terraform.

Также развертывание в нескольких облаках с управляемым Kubernetes не всегда возможно из-за несовместимости управляемых кластеров разных клауд-вендоров.

Managed Kubernetes разных облачных провайдеров обладает разным подходом к управлению, у каждого свой мониторинг, поэтому придется использовать разные инструменты, чтобы привести их в соответствие со своей системой управления. В разных облаках и их регионах разный перфоманс и задержки, а также у вендоров могут различаются стандартные таймауты в кластере. Еще один нюанс - это версии управляемого кластера. Если твои приложения используют передовые фичи Kubernetes, вендоры внедряют новые версии с задержкой. У разных провайдеров разный набор движков выполнения контейнеров, тебе нужно знать об этом, при планировании своих развертываний. Ну и как обычно, ресурсные квоты от облаков - их нужно знать и иметь в виду, как в обычной облачной стратегии, так и в мульти-клауд.

На самом деле не все так страшно, но в любом случае тебе нужно будет знать, как работает аварийное восстановление и как реализована надежность в каждой из твоих мульти-сред и подобрать подходящие для этого инструменты.

Подводя итог, можно сказать, что высокодоступные и надежные приложения (ориентированные на клиентов или критически важные приложения с нулевым временем простоя) требуют самовосстановления нод и надежной подготовки инфраструктуры. Если платформа не обеспечивает этого, производительность приложения не может быть гарантирована.

Что ж, это все в этой лекции. Увидимся в следующей.