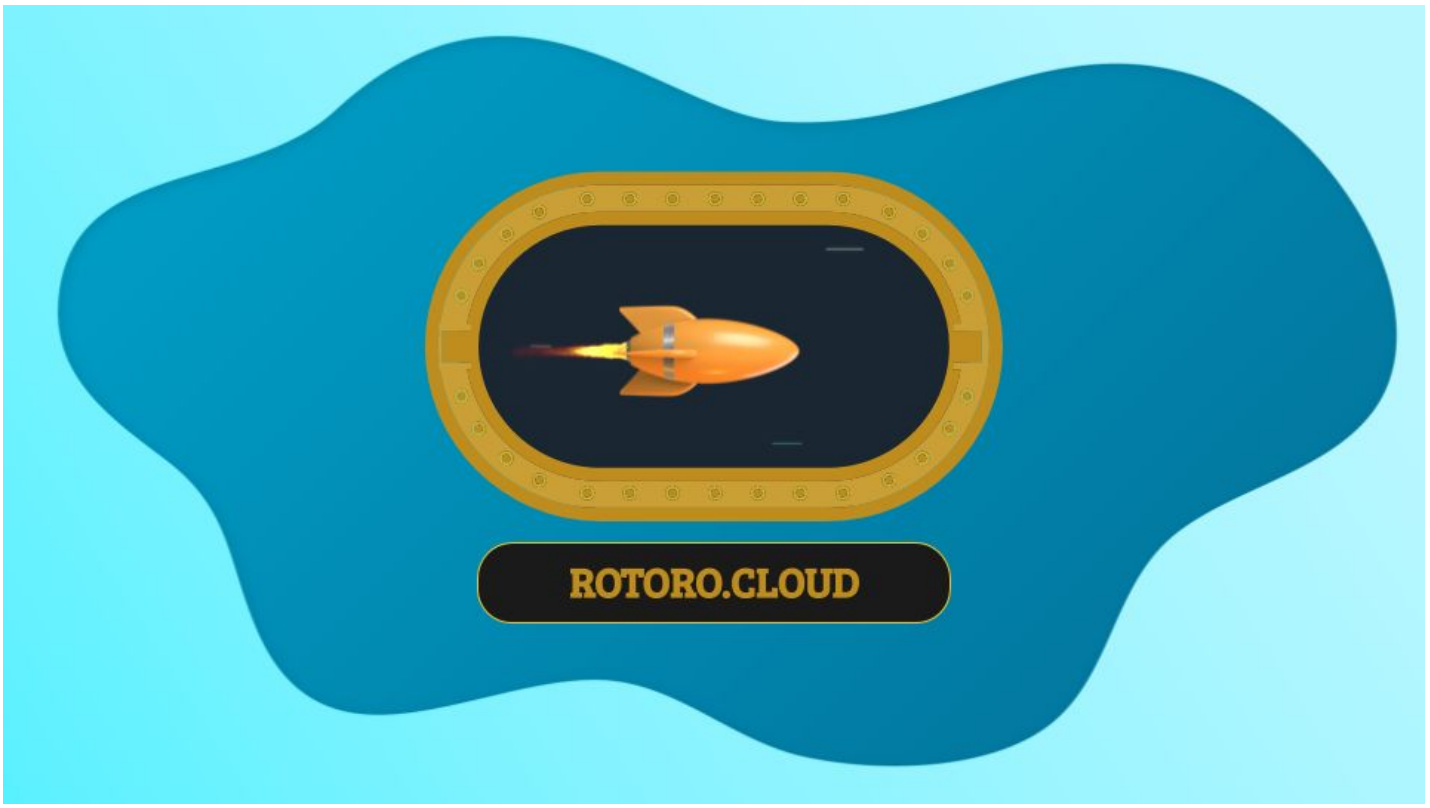
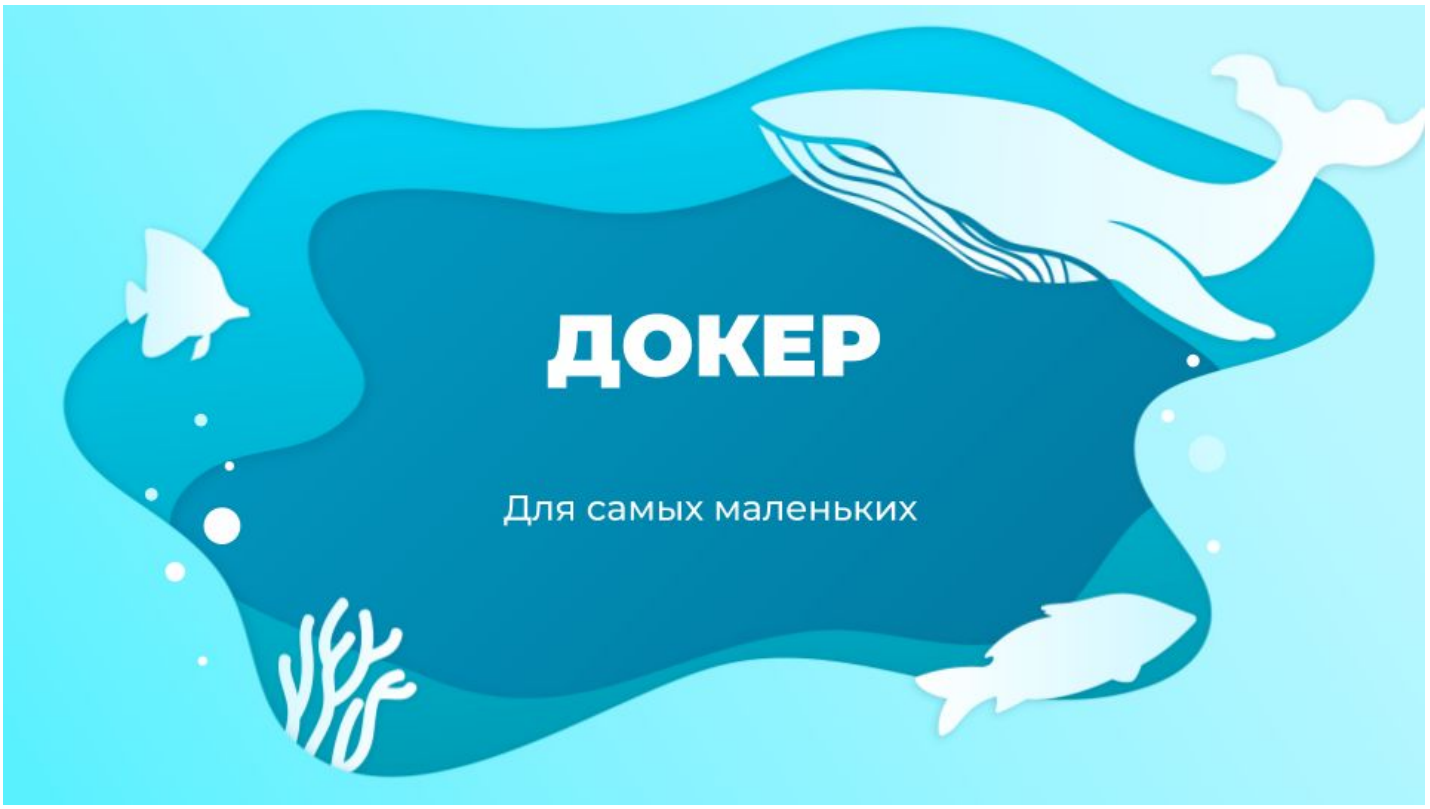
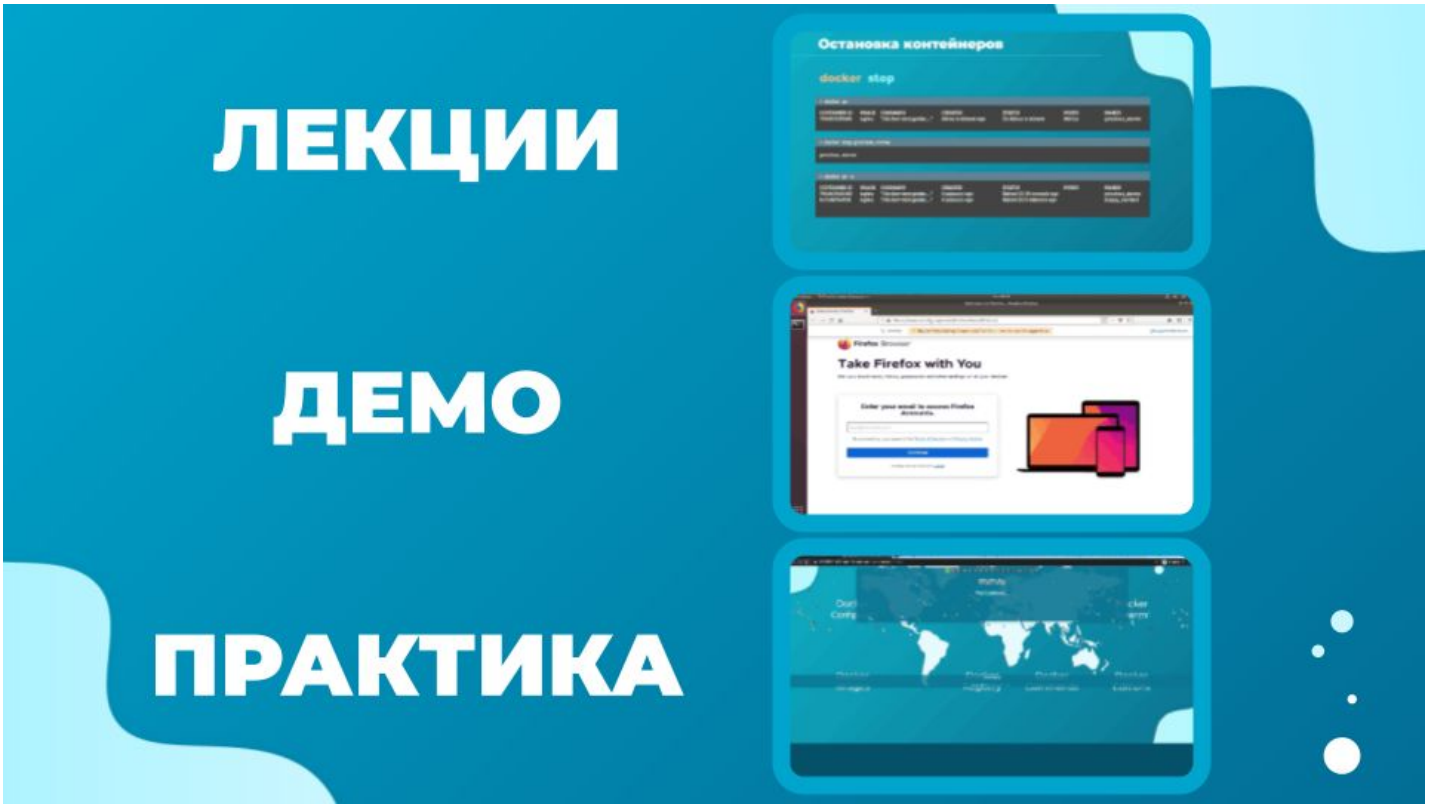




Привет, и добро пожаловать на курс Docker для самых маленьких.

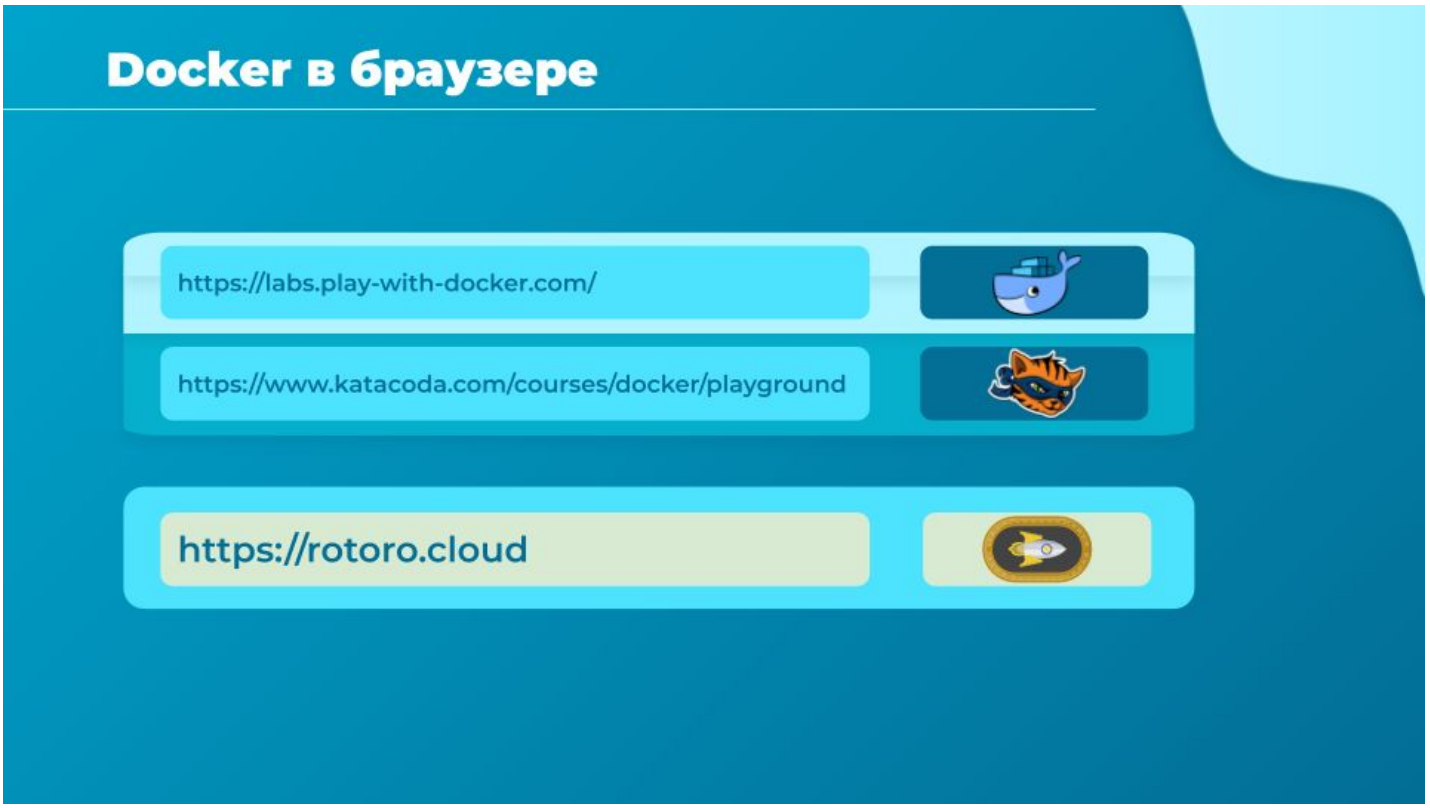
Меня зовут Андрей, я архитектор решений, работаю с клауд аутомейшен и практикую девопс.



- **Этот курс состоит из серии лекций, в которых в легкой и доступной форме я расскажу тебе об основных понятиях Docker.**
- **В демо я покажу как установить и начать работать с Docker.**
- **И еще в этом курсе у нас есть серия практических упражнений прямо из браузера.**

Как это работает я объясню чуть позже, а пока давай познакомимся с задачами курса.

- **Я объясню что такое контейнеры, какие задачи они решают, чем могут помочь и зачем они тебе.**
- **Я покажу как запускаются контейнеры и как создать свой собственный образ.**
- **Поговорим про сеть в докере, посмотрим на Docker-compose.**
- **Узнаем что такое private registry и как его установить.**
- **Немного заглянем “под капот” докера и поразбираемся на чем основана технология.**
- **Я объясню отличия докера для Windows and Mac, также поговорим про оркестрацию с Docker Swarm и Kubernetes.**

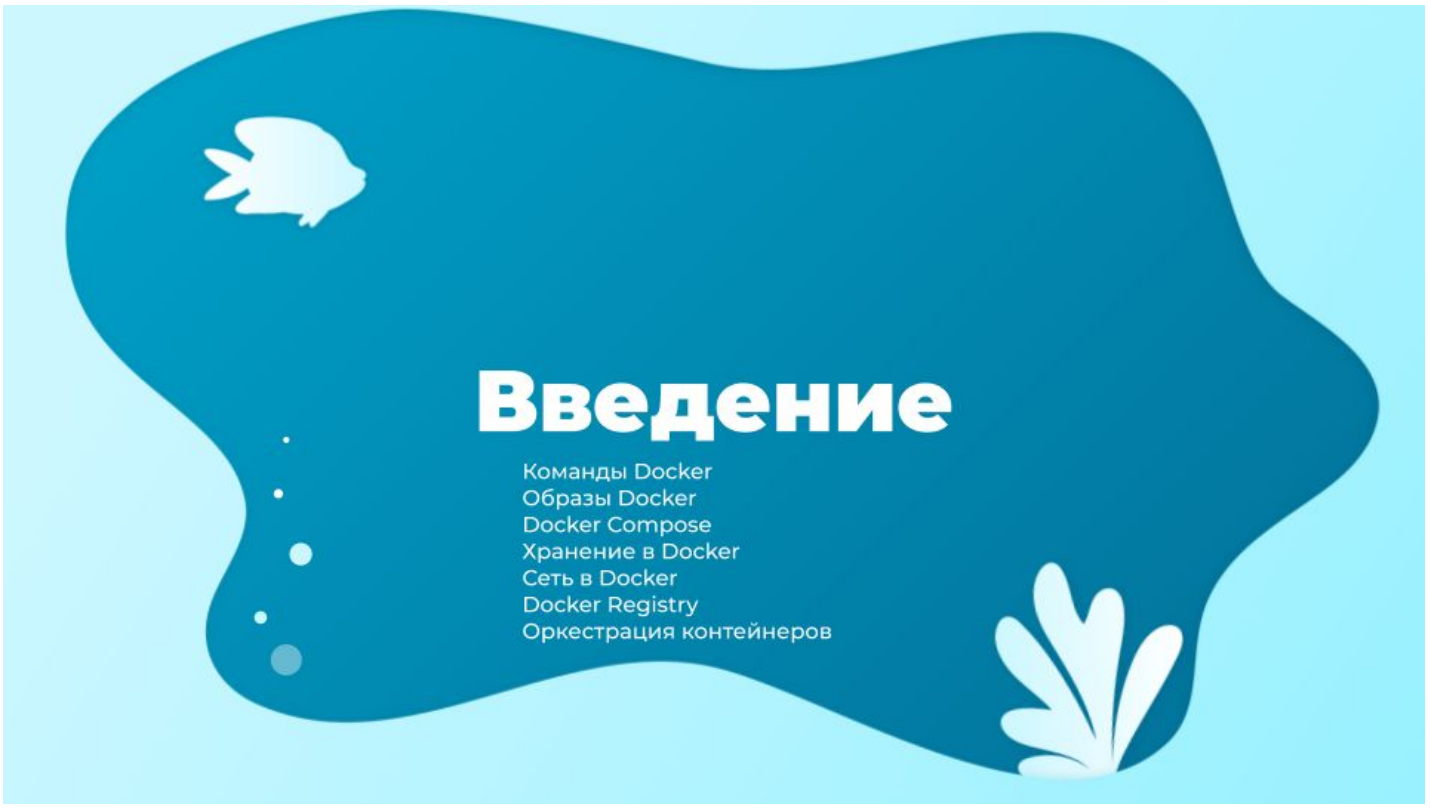


По поводу лабораторных.
Ты можешь установить Docker и попробовать пройти их на своем окружении, для этого в демо я покажу как установить Docker и ты можешь использовать команды из лекций.

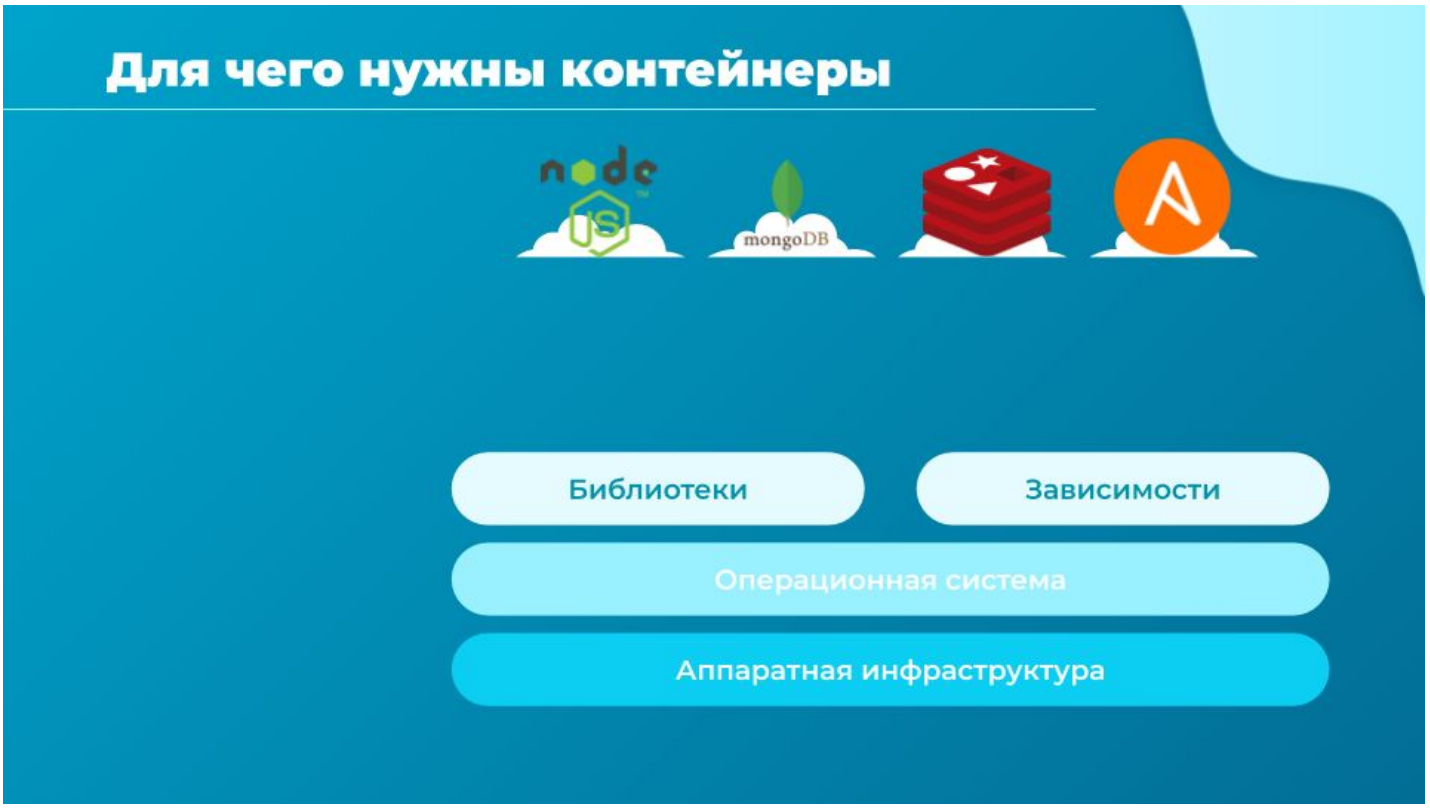
Но в рамках курса я предоставлю тебе полностью подготовленное окружение, в котором ты сможешь практиковать без какой-то подготовки прямо из браузера. Также я предоставлю тебе лабораторные, которые отражают материал из курса, которые ты сможешь запустить в удобное тебе время, столько раз, сколько тебе нужно. Специальный обучающий портал будет задавать вопросы, ставить задачи и проверять твои действия.

После упражнения ты сможешь поделится впечатлением об прохождении.
Это интересный интерактивный подход к обучению.

Надеюсь, я разжег твое любопытство, так что вперед, поехали!



Привет и добро пожаловать на лекцию где мы изучаем основы Docker.
В этой лекции рассмотрим общий обзор того, зачем нужен Docker и что он может для нас сделать. Мы начнем с Docker helicopter view, чтобы понять, что это на самом деле такое и как он поможет в твоих задачах.



Позволь мне начать с рассказа о том, как я познакомился с Docker. В одном из моих предыдущих проектов было требование настроить сквозной стек включающий различные технологии:

- **веб-сервер, использующий node JS**
- **базу данных, как mongoDB**
- **систему обмена сообщениями, вроде Redis**
- **оркестрацию с помощью Ansible**

Из-за того, что компоненты были разнородные, мы столкнулись с множеством проблем при разработке и поддержке этого приложения.

Во-первых, их совместимость с базовой операционной системой. Мы должны были обеспечить, чтобы все эти разные сервисы были совместимы с версией операционной системы, которую мы планировали использовать. Случалось, что некоторые версии этих компонентов были несовместимы с ОС. Нам приходилось останавливаться и искать другую ОС, совместимую со всеми этими службами.

Во-вторых, нам нужно было быть уверенными в совместимости сервисов и библиотек с зависимостями в ОС. У нас были проблемы, когда для одной службы требовалась одна версия зависимой библиотеки, а для другой службы требовалась другая версия.

Со временем архитектура нашего приложения менялась, нам пришлось обновить компоненты до более новой версии, поменять базу данных и так дальше.

И каждый раз, когда что-то менялось, нам приходилось проходить один и тот же процесс проверки совместимости между этими различными компонентами и базовой инфраструктурой.

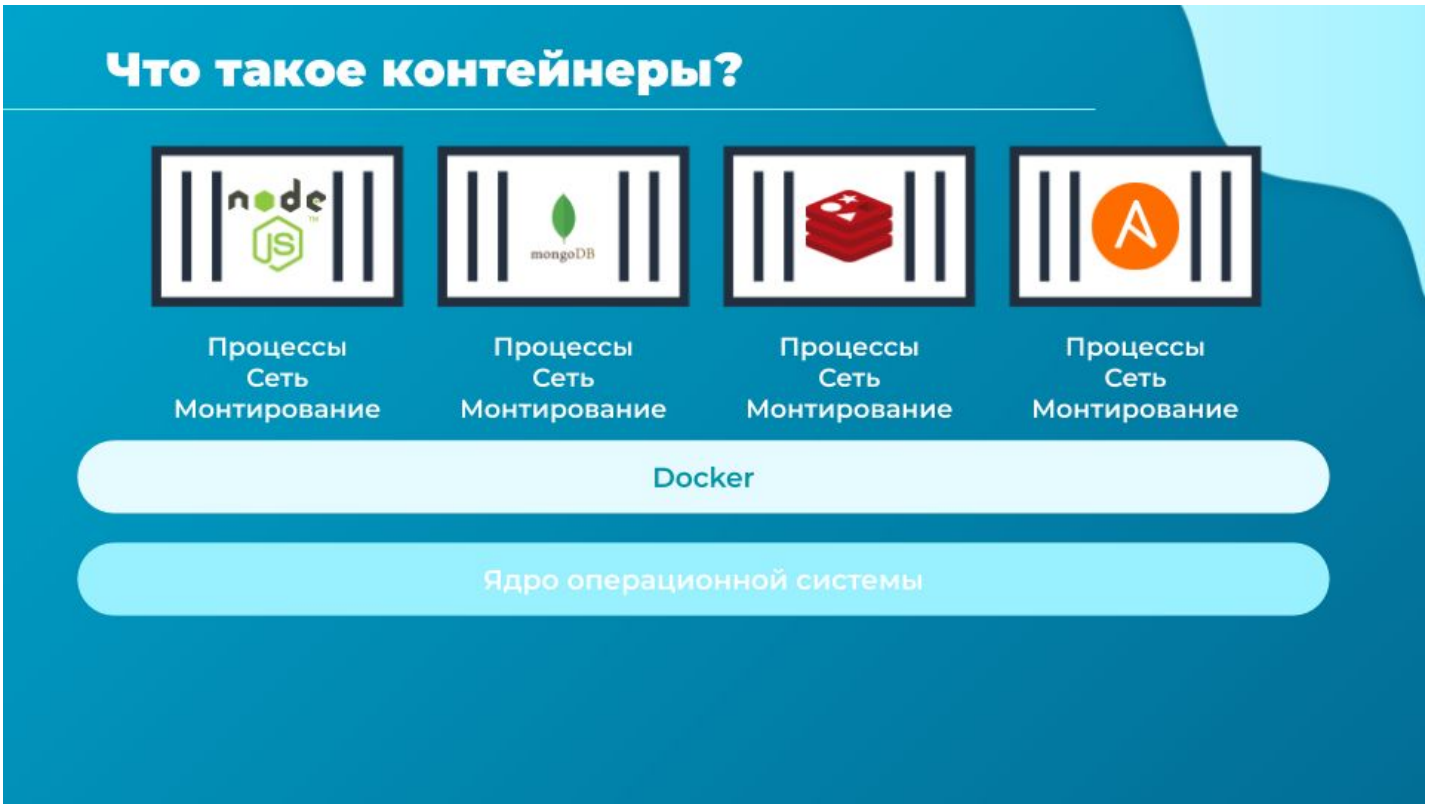
Эта проблема с матрицей совместимости обычно называется «Матрица из ада».
Далее каждый раз, когда у нас появлялся новый разработчик, нам было действительно сложно создать новую среду. Новым разработчикам приходилось следовать большому набору инструкций и запускать сотни команд, чтобы наконец настроить свою среду. Они должны были убедиться, что используют правильную операционную систему, правильные версии каждой из этих компонент и каждый разработчик должен был настраивать все это каждый раз сам.

У нас также были разные среды для разработки, тестирования и продакшена. Одному разработчику было удобно использовать одну ОС, а другим - другую. Поэтому мы не могли гарантировать, что созданное приложение будет работать ОДИНАКОВО в разных СРЕДАХ.

Итак, все это значительно усложнило нам жизнь в разработке, создании и доставке приложения.
Мне нужно было что-то, что могло бы помочь нам с проблемой совместимости.
Что-то, что позволит нам изменять или заменять эти компоненты, не затрагивая другие компоненты, и даже изменять базовые операционные системы по мере необходимости.

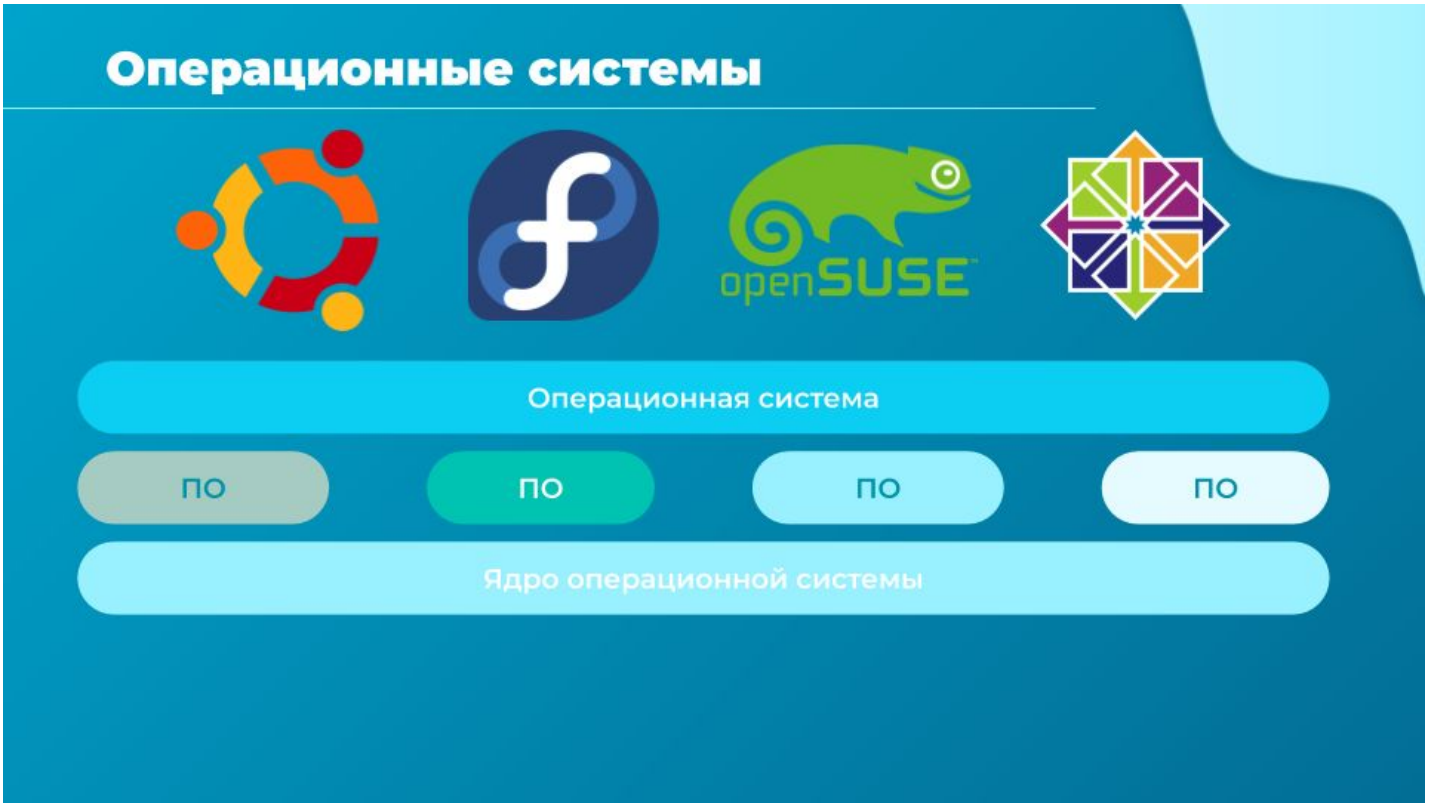
И этот поиск привел меня к Docker. С Docker я мог запускать каждый компонент отдельно, но со своими собственными библиотеками и зависимостями от одной и той же виртуальной машины и ОС. Все упаковывалось в отдельное окружение - контейнер. Нам просто нужно было один раз собрать конфигурацию Docker, и теперь все наши разработчики могли начать работу с простой команды запуска Docker.

Независимо от того, какую операционную систему они используют всё, что им нужно было сделать,
это убедиться, что в их системах установлен Docker.



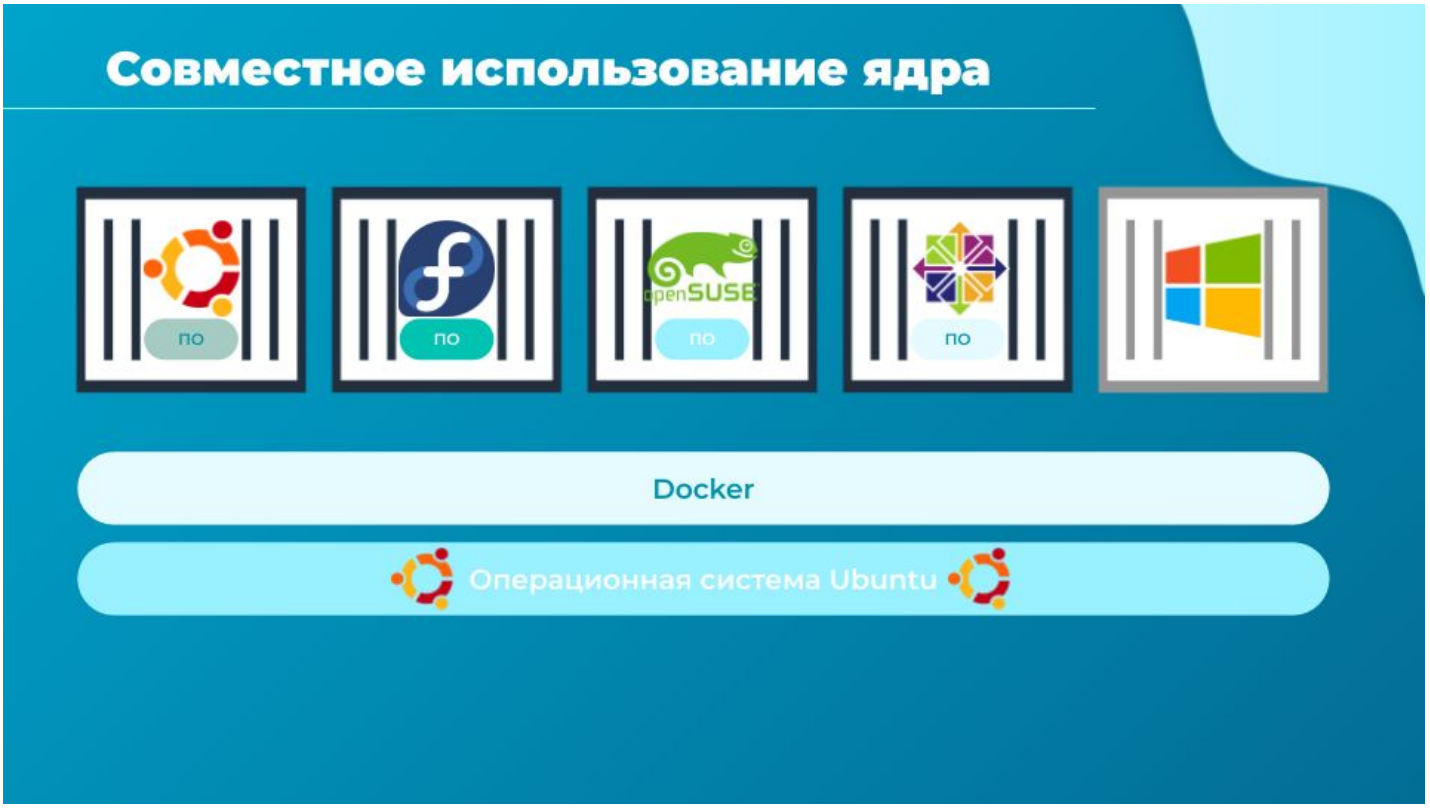
Так что же такое контейнеры? Контейнеры - это полностью изолированные среды. В них свои процессы и службы, собственные сетевые интерфейсы, собственные средства монтирования. Это похоже на виртуальные машины. За исключением того, что все они используют одно и то же ядро операционной системы. Мы немного рассмотрим, что это значит.

Также важно отметить, что контейнеры не новость в Docker. Контейнеры существовали около 10 лет. Вот некоторые из типов контейнеров: LXC, LXD, LXCFS и др. Docker использует контейнеры LXC. Настроить эти контейнерные среды сложно, поскольку они очень низкого уровня и именно здесь докер предлагает инструмент высокого уровня с несколькими мощными функциями, которые делают его действительно легким для таких конечных пользователей, как мы.



Чтобы понять, как работает Docker, давай сначала вернемся к некоторым основным концепциям операционных систем. Если ты посмотришь на такие операционные системы, как Ubuntu, Fedora, Suse или CentOS все они состоят из двух вещей: ядро ОС и набор ПО.

Ядро операционной системы отвечает за взаимодействие с базовым оборудованием.
В то время как ядро ОС остается тем же (в данном случае Linux), слой программного обеспечения, расположенный выше, отличается для разных операционных систем. Это программное обеспечение может состоять из интерфейса пользователя, драйверов, компиляторов, файловых менеджеров, инструментов разработчика и т. д. В итоге, у нас есть общее ядро Linux, используемое во всех операционных системах, и некоторое настраиваемое программное обеспечение, которое отличает одну операционную систему от другой.



Ранее мы говорили, что контейнеры Docker совместно используют базовое ядро. Что это на самом деле означает “совместное использование ядра”?

Допустим, у нас есть система с ОС Ubuntu с установленным на ней Docker. Docker может запускать поверх себя любую версию ОС, если все они основаны на одном ядре. В данном случае Linux. Если базовой операционной системой является Ubuntu, Docker может запускать контейнер на основе другого дистрибутива, например Debian, Fedora, Suse или CentOS.

В каждом докер-контейнере есть только дополнительное программное обеспечение, которое делает операционные системы разными. Это то, про что говорили на предыдущем слайде.

И Docker использует базовое ядро своего хоста, которое работает со всеми перечисленными выше операционными системами. А какая ОС не имеет такого же ядра? Windows.
И поэтому ты не сможешь запустить контейнер на базе Windows на докер-хосте с ОС Linux.
Для этого нам потребуется Docker на сервере Windows.

Ты спросишь, разве это не недостаток? Невозможно запустить другое ядро в ОС. Ответ - нет, потому что, в отличие от гипервизоров, докер не предназначен для виртуализации и запуска различных операционных систем и ядер на одном оборудовании. Основное назначение Docker - контейнеризация приложений, а также их отправка и запуск.

Ок, это подводит нас к различиям между виртуальными машинами и контейнерами. Вещи, что обычно делают те, кто занимается виртуализацией.

Как ты видишь справа, в случае с Docker у нас есть базовая аппаратная инфраструктура, затем операционная система и Docker, установленный на ОС. Docker может управлять контейнерами, которые работают с библиотеками и зависимостями сами, в одиночку.

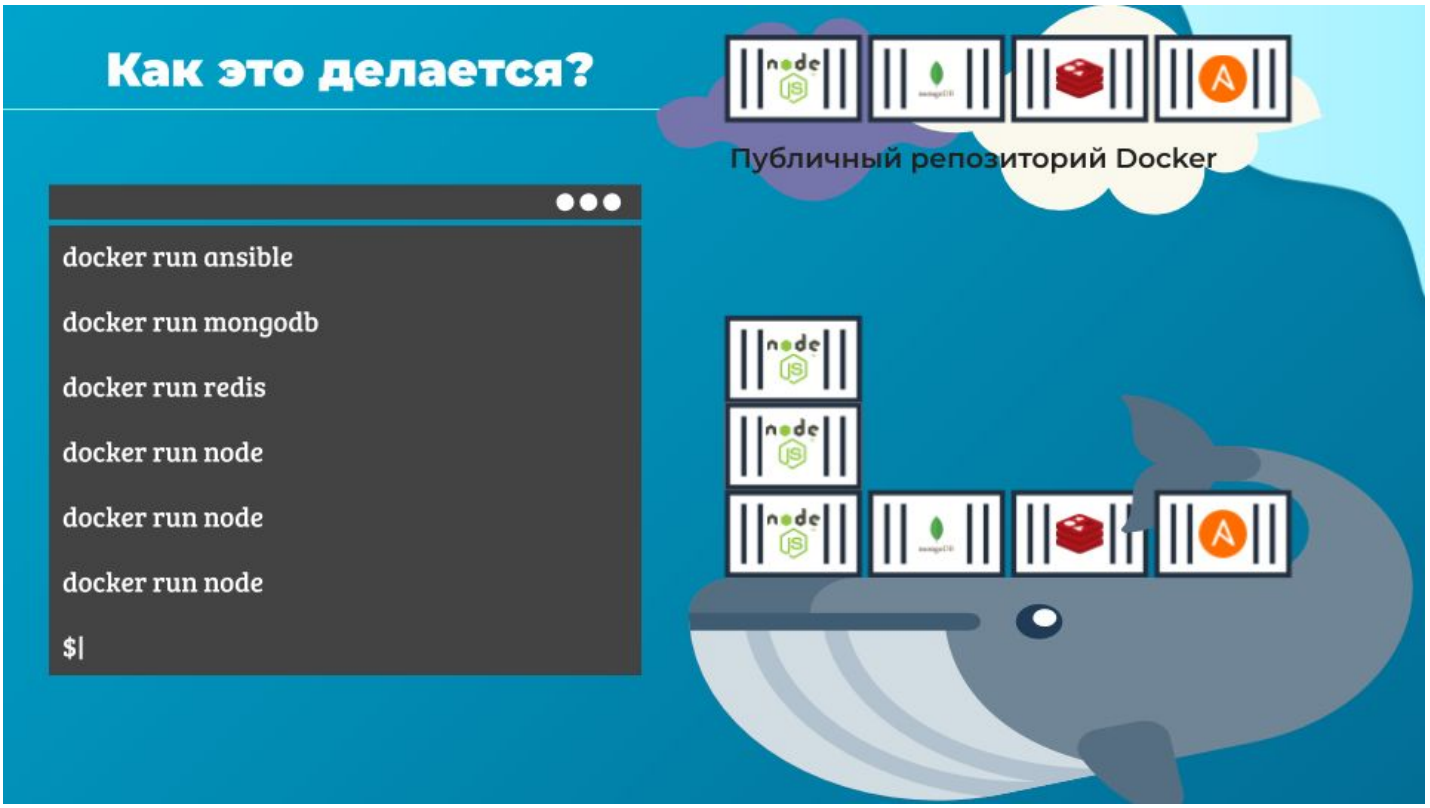
В случае виртуальной машины у нас есть ОС на базовом оборудовании, затем гипервизор, такой как ESX, или какая-то виртуализация, а затем виртуальные машины. Как видишь, у каждой виртуальной машины внутри своя операционная система. Толстый слой зависимостей, а затем приложение. Эти накладные расходы приводят к более высокому использованию вычислительных ресурсов,
поскольку приходится крутить несколько виртуальных операционных систем с их ядрами. Виртуальные машины также потребляют больше дискового пространства, поскольку каждая виртуальная машина тяжелая и обычно имеет размер в гигабайтах, тогда как контейнеры Docker легковесны и обычно имеют размер в мегабайтах. Это позволяет контейнерам докера загружаться быстрее, обычно за считанные секунды, тогда как виртуальные машины, как мы знаем, грузятся по несколько минут, так как для этого требуется загрузка всей операционной системы.

Также важно отметить, что Docker имеет меньшую изоляцию, поскольку больше ресурсов используется совместно между контейнерами, например ядро, тогда как виртуальные машины полностью изолированы друг от друга.

В случае виртуальной машины у нас могут быть разные типы операционных систем, такие как Linux или Windows на основе одного и того же гипервизора, тогда как на одном докер-хосте это невозможно.

Ну так все же что выбрать? Контейнеры или виртуальные машины?
Я скажу так - выбери лучшее от обеих технологий.
Ты можешь работать с контейнерами на виртуальных хостах Docker.
Виртуализация даст тебе возможность легко вводить, выводить из эксплуатации или перемещать большое количество докер-хостов, а контейнеризация быстро масштабироваться и легко обновляться.
С таким подходом тебе не нужно создавать определенную виртуальную машину под конкретное приложение. Теперь задача этой VM - запускать контейнеры и делать это хорошо.

И эта эластичность и гибкость отлично подходит к облачным вычислениям.
В облаке сегодня ты можешь иметь тысячи контейнеров на сотнях докер-хостов, а завтра легко вырасти или уменьшиться при необходимости.



Это были ключевые различия между ними. Итак, как нам сделать это руками?

На сегодняшний день доступно множество контейнерных версий приложений. Продукты большинства организаций хранятся в контейнерах и могут быть получены из общедоступного докер-реджистри под названием docker hub. Например, ты можешь найти образы наиболее распространенных операционных систем, баз данных и других сервисов и инструментов.

Как только ты определился с образом, тебе нужно установить Docker на свой хост. Для запуска приложения просто напиши команду docker run с именем образа. В этом случае запуск команды docker run ansible запустит экземпляр ansible на докер-хосте. Точно так же запустим экземпляры mongoDB, Redis и nodeJS с помощью команды docker run. Когда ты запускаешь nodejs, просто укажи расположение репозитория кода.

Если тебе нужно запустить несколько экземпляров веб-службы, просто создай требуемое количество экземпляров и настрой какой-либо балансировщик нагрузки на входе. В случае отказа одного из экземпляров просто уничтожь этот экземпляр и запусти новый инстанс.

Существуют и другие решения для обработки таких случаев, их мы рассмотрим позже в ходе этого курса.

Мы говорили об образах и контейнерах. Давай поймем разницу между ними.

Образ - это пакет или шаблон, аналогичный шаблону виртуальной машины, с которым ты, возможно, работал в мире виртуализации. Он используется для создания одного или нескольких контейнеров. Docker запускает экземпляры образов, которые изолированы, имеют свои собственные среды и процессы.

Как мы видели раньше, многие продукты уже докеризованы. Если ты не можешь найти, то что тебе нужно, ты можешь создать образ самостоятельно, отправить его в репозиторий Docker Hub и сделать его доступным для всех.

Если посмотреть традиционно, приложения разрабатывают разработчики. Потом передают апстриму для развертывания и управления в производственных средах. Они делают это, предоставляя набор инструкций, таких как информация о том, как должны быть установлены хосты, какие предварительные условия должны быть соблюдены и как должны быть настроены зависимости и т.д.

Команда ops использует это руководство для настройки приложения. Поскольку команда ops не разрабатывала приложение самостоятельно, они борются с его настройкой. Когда они сталкиваются с проблемой, они начинают дергать разработчиков.
С Docker основная часть работы, связанная с поднятием инфраструктуры, теперь находится в руках разработчиков в виде Dockerfile. Руководство, ранее созданное разработчиками для настройки инфраструктуры теперь просто ложится Dockerfile для создания образа приложения.

Этот образ теперь может работать на любой контейнерной платформе и гарантированно будет работать везде одинаково.

Таким образом, команда оперирования теперь может просто использовать образ для развертывания приложения. Поскольку образ работал, когда разработчик создал его, а другие операции не изменяют образ, он продолжит работать таким же образом при развертывании в производственной среде.

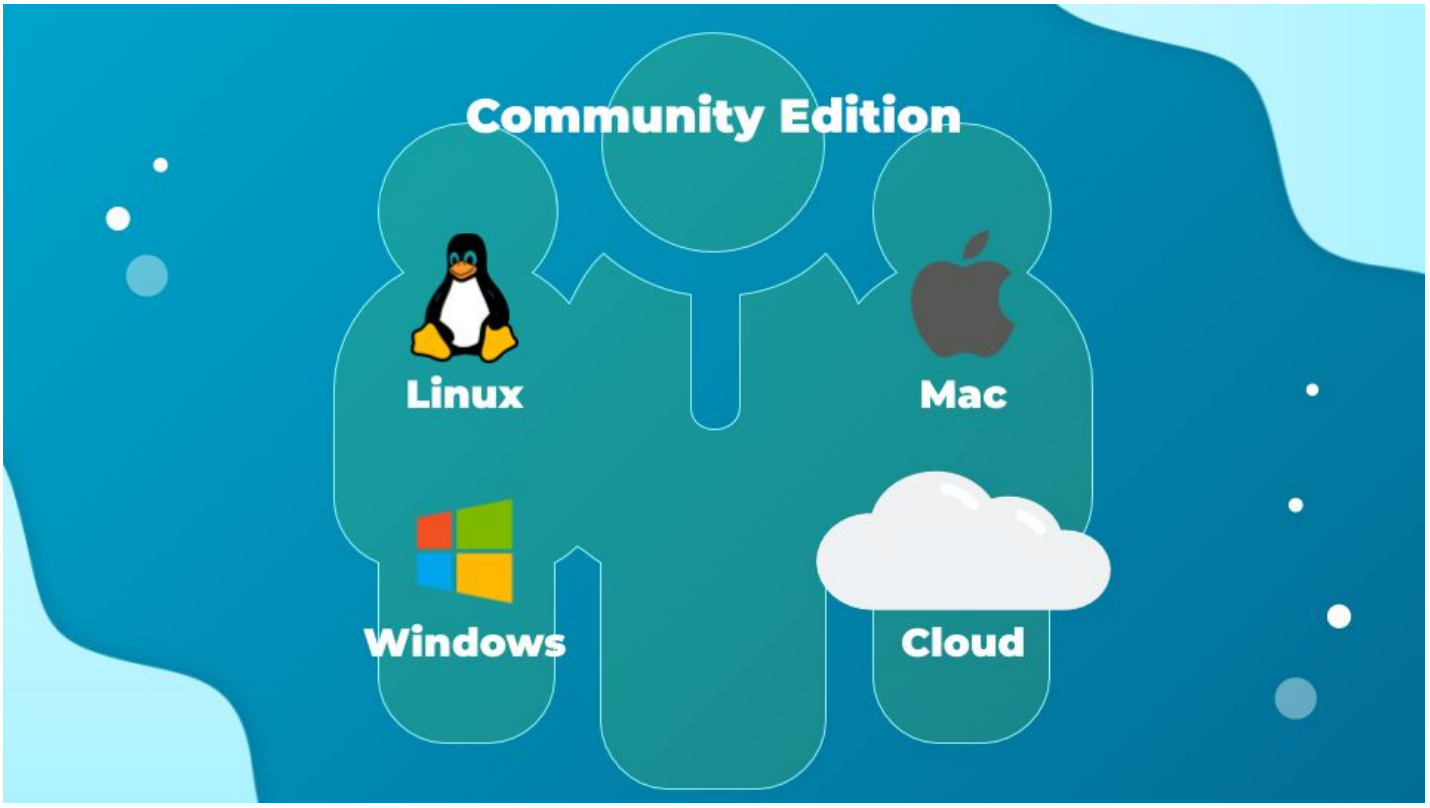
Привет, давай посмотрим, как начать работать с Docker.

В данный момент у Docker есть два варианта использования:

- **Community Edition**
- **Enterprise Edition**

Первый - это набор бесплатных продуктов, второй - это сертифицированная и поддерживаемая контейнерная платформа для предприятия, которая включает в себя расширения по управлению хранением, безопасностью и оркестрацией. Она уже стоит денег.

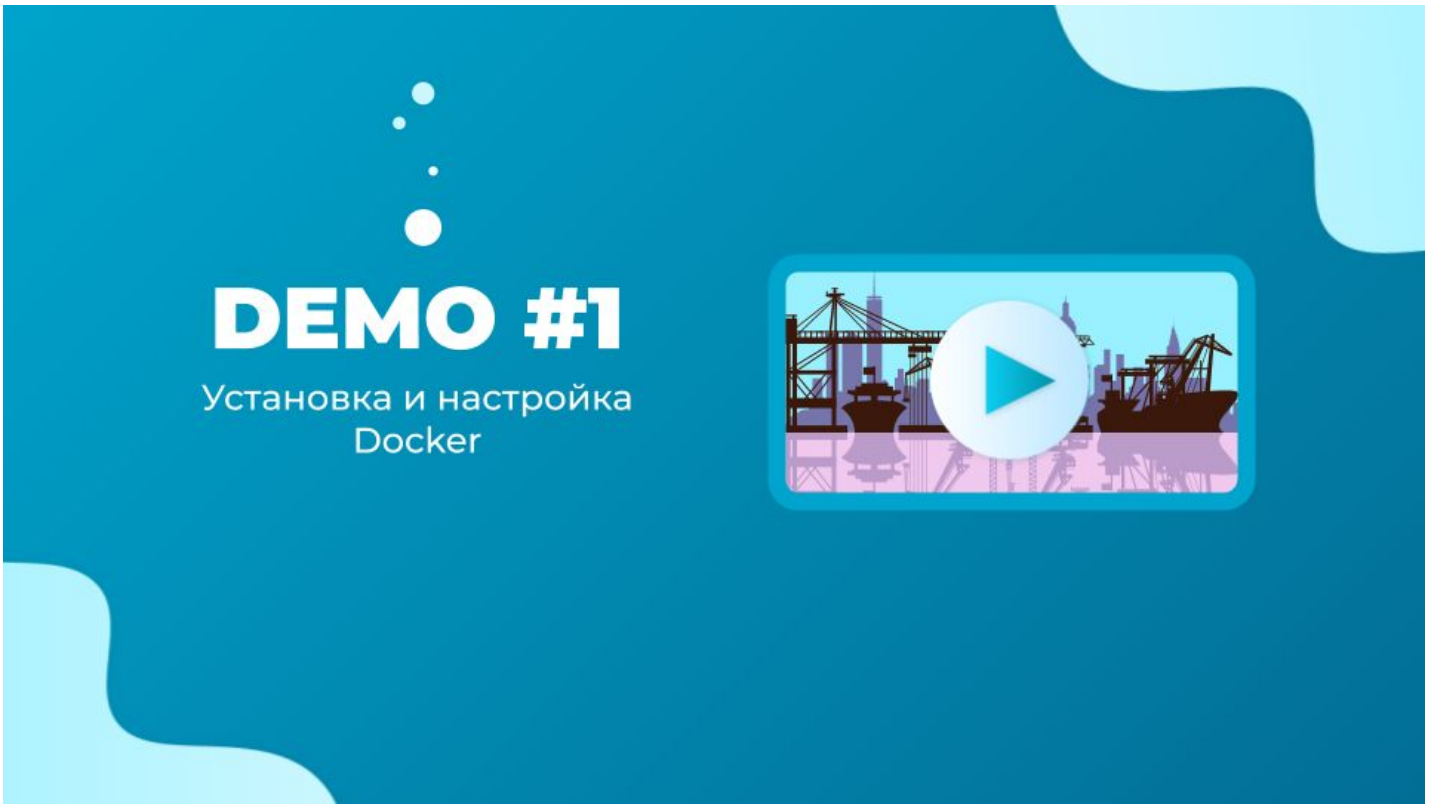
Решения по оркестрации контейнеров есть не только от Docker, мы это обсудим позже в этом курсе.



Здесь, в этом курсе, мы будем работать с community edition. Эта редакция доступна на Linux, Mac, Windows и на облачных решениях вроде GCP или AWS.

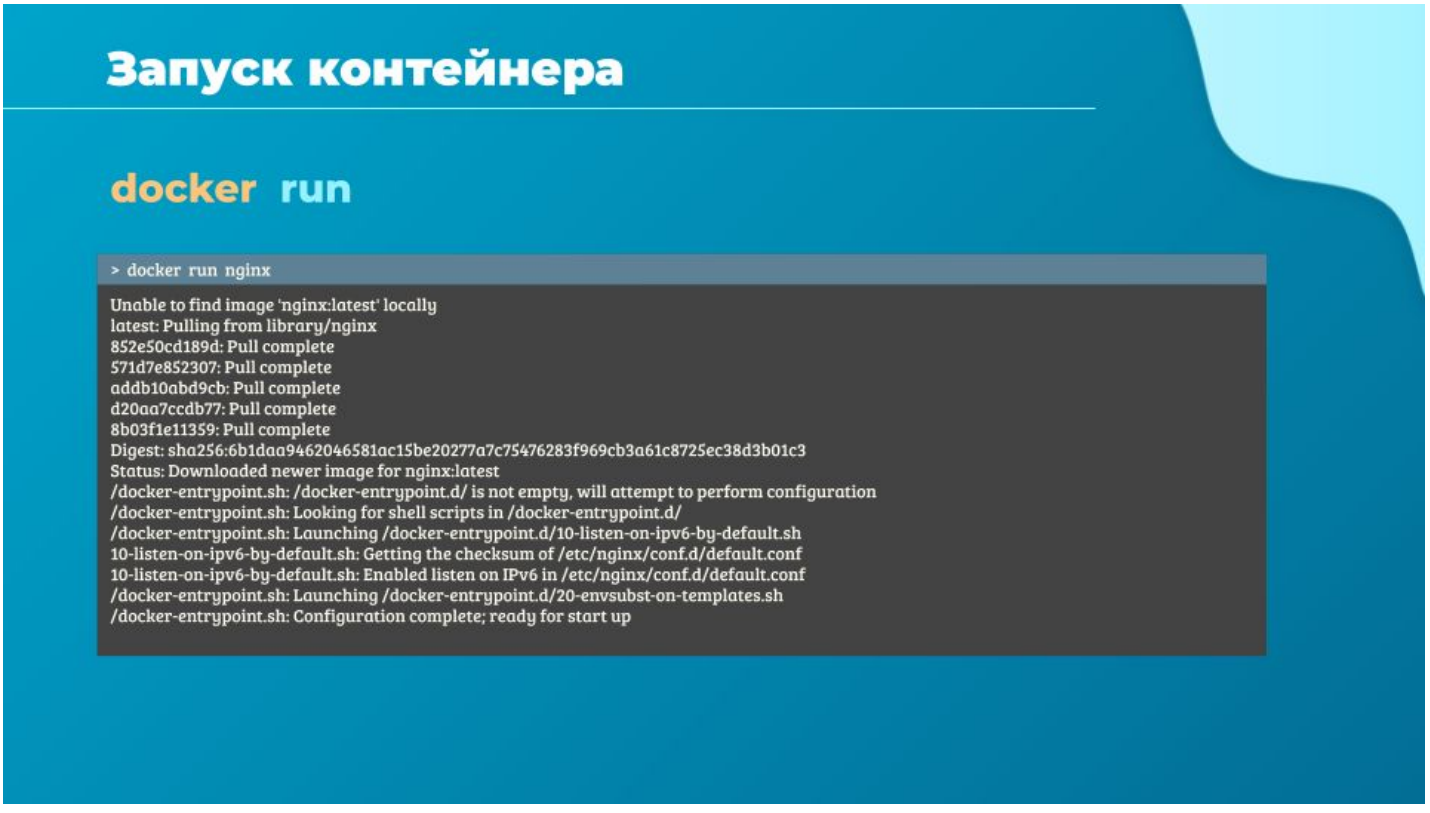
В демо мы посмотрим, как установить и начать работать с Docker на машине с Linux. Если ты на Mac или Windows, то есть два варианта. Первый - установить виртуальную машину с Linux при помощи Virtualbox или какой-то другой платформы виртуализации, а потом выполнить шаги из демо, которое идет после лекции. На мой взгляд это самый простой вариант начать с Docker.

Второй вариант - установить Docker desktop для Mac или для Windows, в соответствии с твоей системой. О различиях Docker для Mac и Windows я расскажу в конце курса, так что если ты выбрал второй путь, посмотри нужный раздел.



Теперь давай перейдем в демо, где посмотрим как установить Docker на Linux.

В этой лекции мы поговорим и посмотрим на команды. В конце я предложу тебе лабораторную, для закрепления твоего понимания предмета. Ты сможешь попрактиковаться с этими командами.

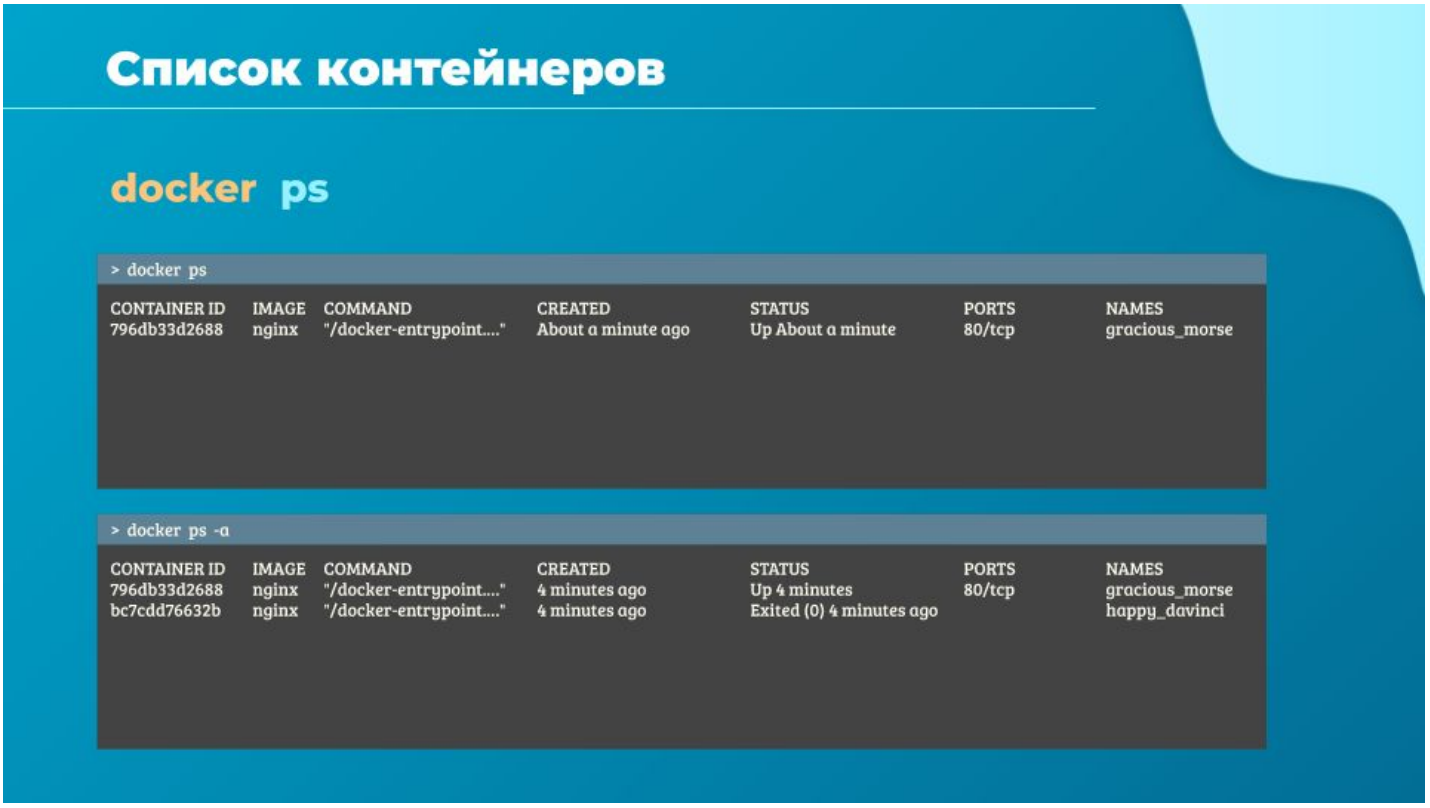


Начнем с команды Docker Run.

Как ты помнишь образ - это шаблон приложения и его зависимостей, собранных вместе, а контейнер это экземпляр образа. Docker run используется для запуска контейнера из образа.

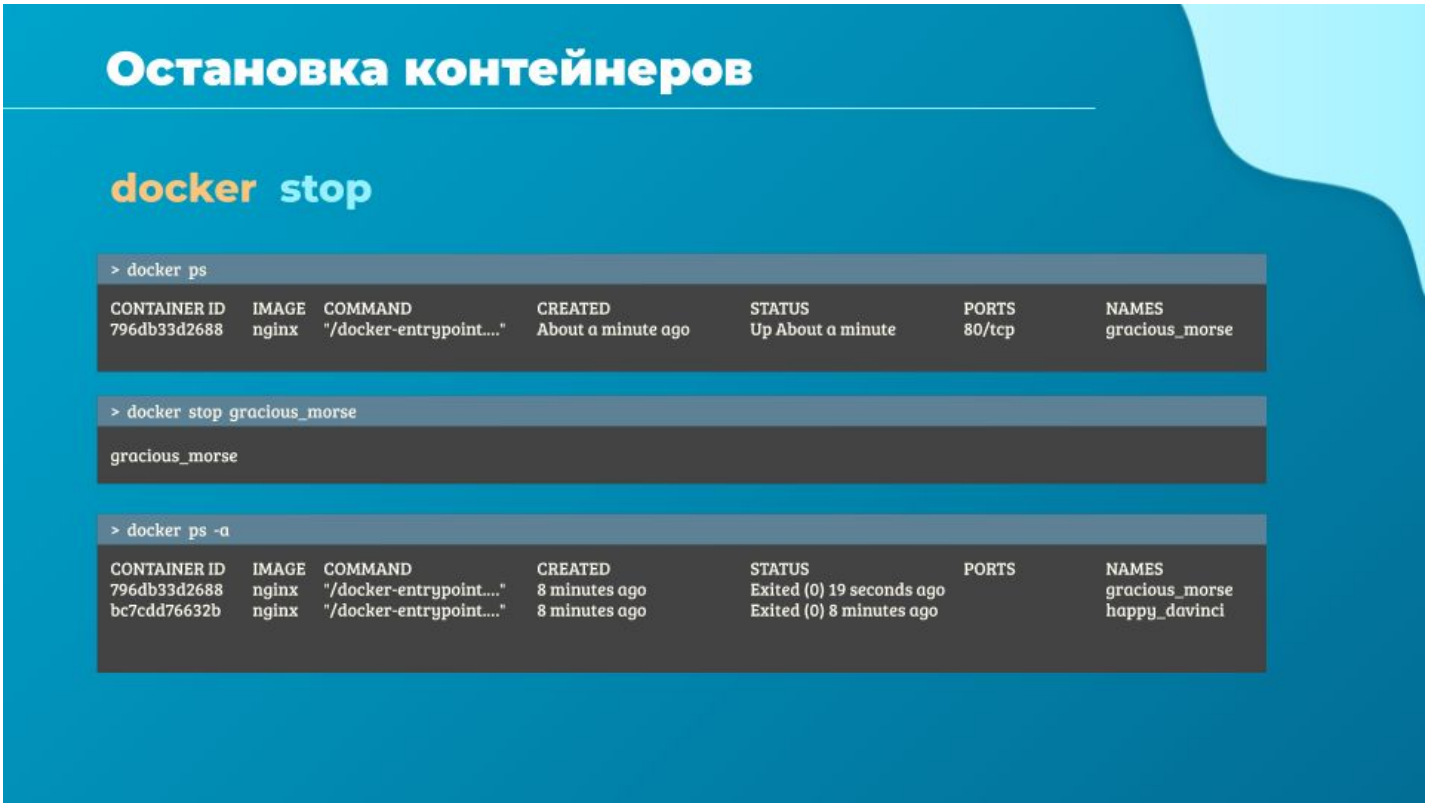
Команда `docker run nginx` запустит экземпляр приложения nginx на твоем докер-хосте, если этот образ уже присутствует локально. Если его нет на хосте, то Docker сходит на Docker hub и спуллит

этот образ себе. Это потребуется сделать всего один раз, при следующих запусках этого контейнера Docker будет использовать локальный образ.



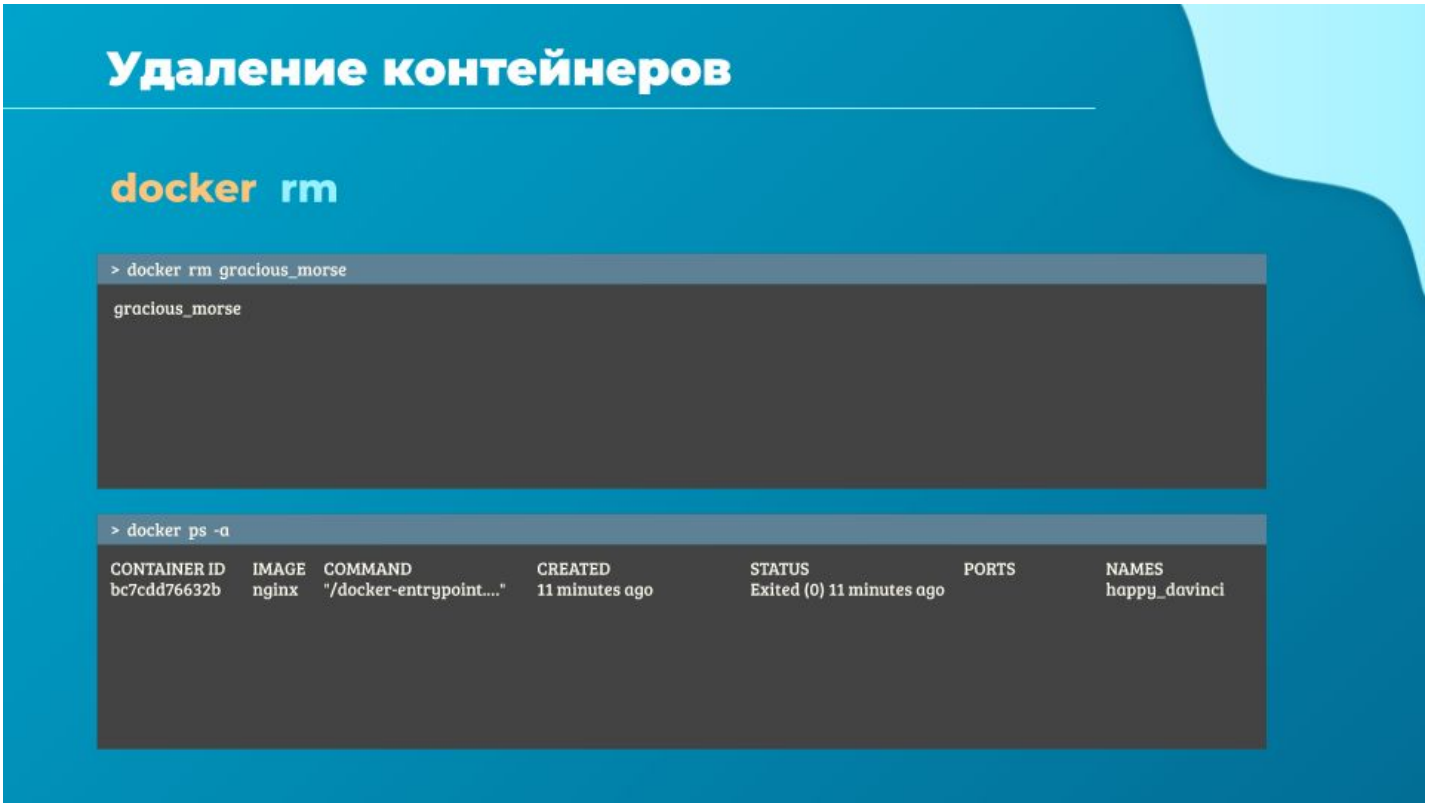
Ты можешь посмотреть список запущенных контейнеров и основную информацию о них, вроде имени контейнера, его ID, статуса, образа с помощью команды `docker ps`.

Каждый контейнер получает случайным образом выбранный ID и случайное имя, если оно не было указано при создании. Весь список контейнеров (и запущенных и нет) показывает команда `docker ps -a`. Как видишь в системе 2 контейнера, контейнер №1 работает, в №2 нет.

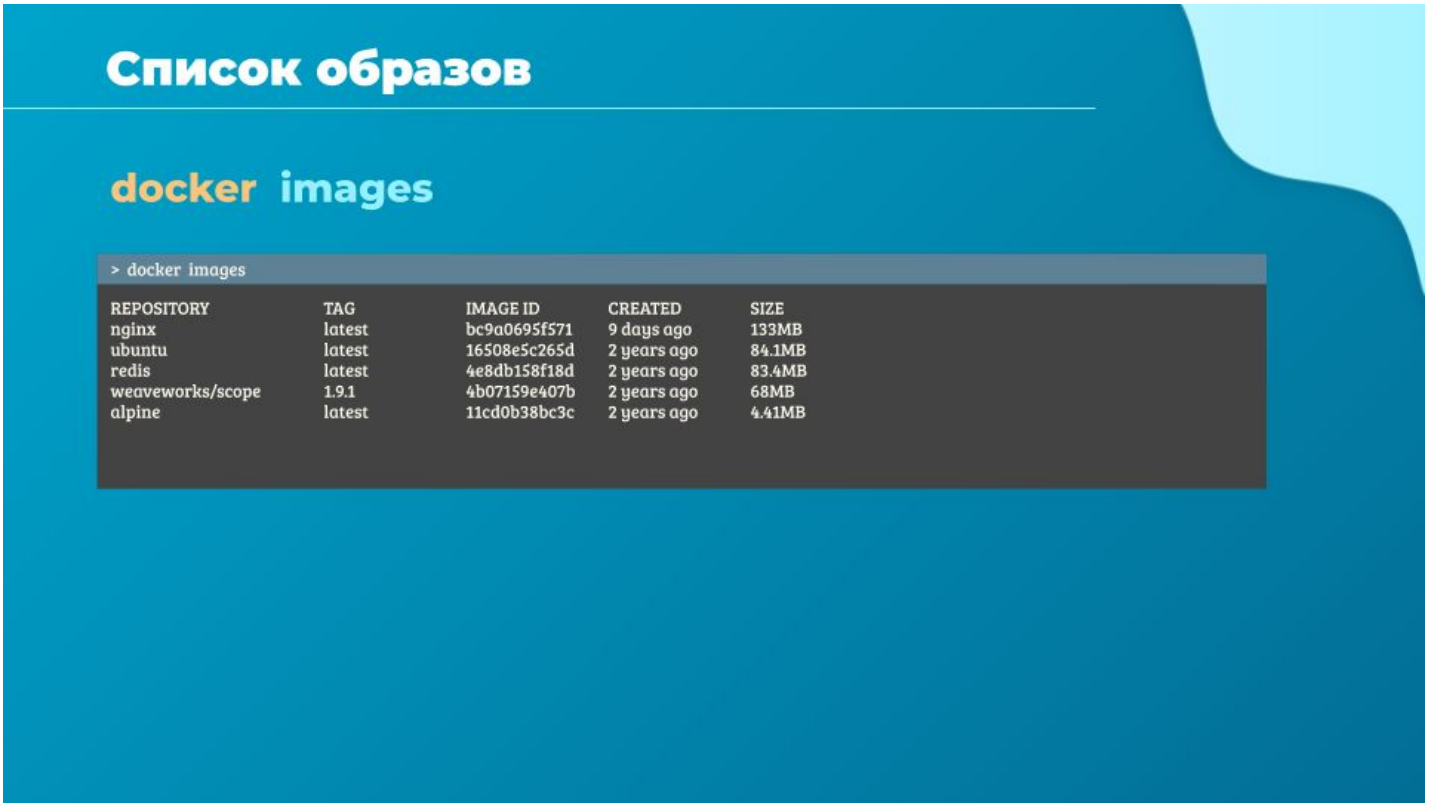


Как остановить работающий контейнер? Используй `docker stop` вместе с ID контейнера или его именем. Если ты не уверен в имени или ID лучше для начала вызови `docker ps`.

Еще раз `docker ps` - запущенные, `docker ps -a` все, которые есть на хосте, и работающие и нет.

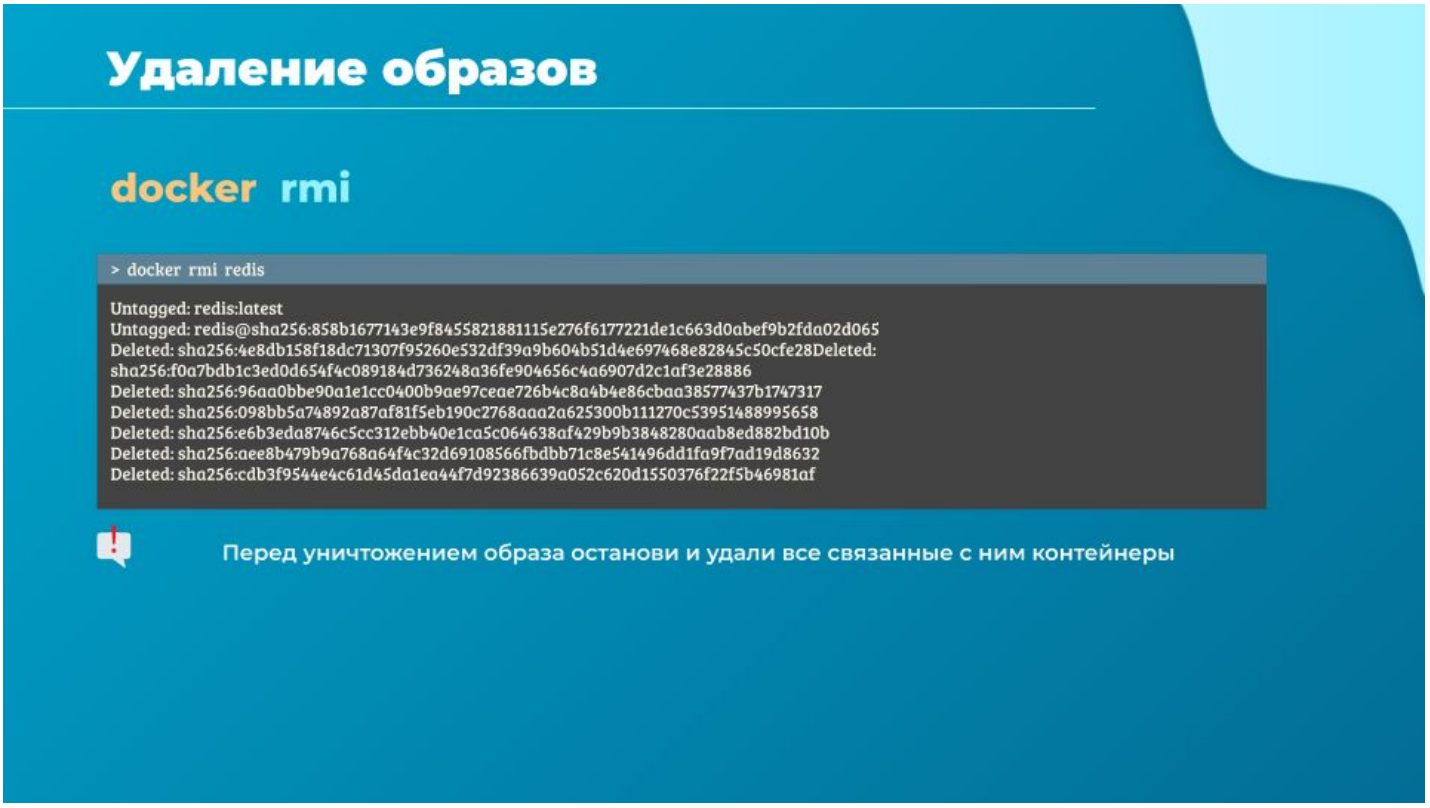


Контейнер `gracious_morse` в статусе exited. Этот контейнер был создан для разовой работы, он свою задачу выполнил, завершился и в будущем мне не понадобится. Теперь он просто занимает место на моем диске. Чтобы избавится от ненужных контейнеров используй `docker rm`.
Эта команда полностью удаляет остановленные (stopped) или завершенные (exited) контейнеры.
В случае успеха она вернет имя или id контейнера, для проверки всегда можно запустить `docker ps` и проверить, все ли верно удалилось.



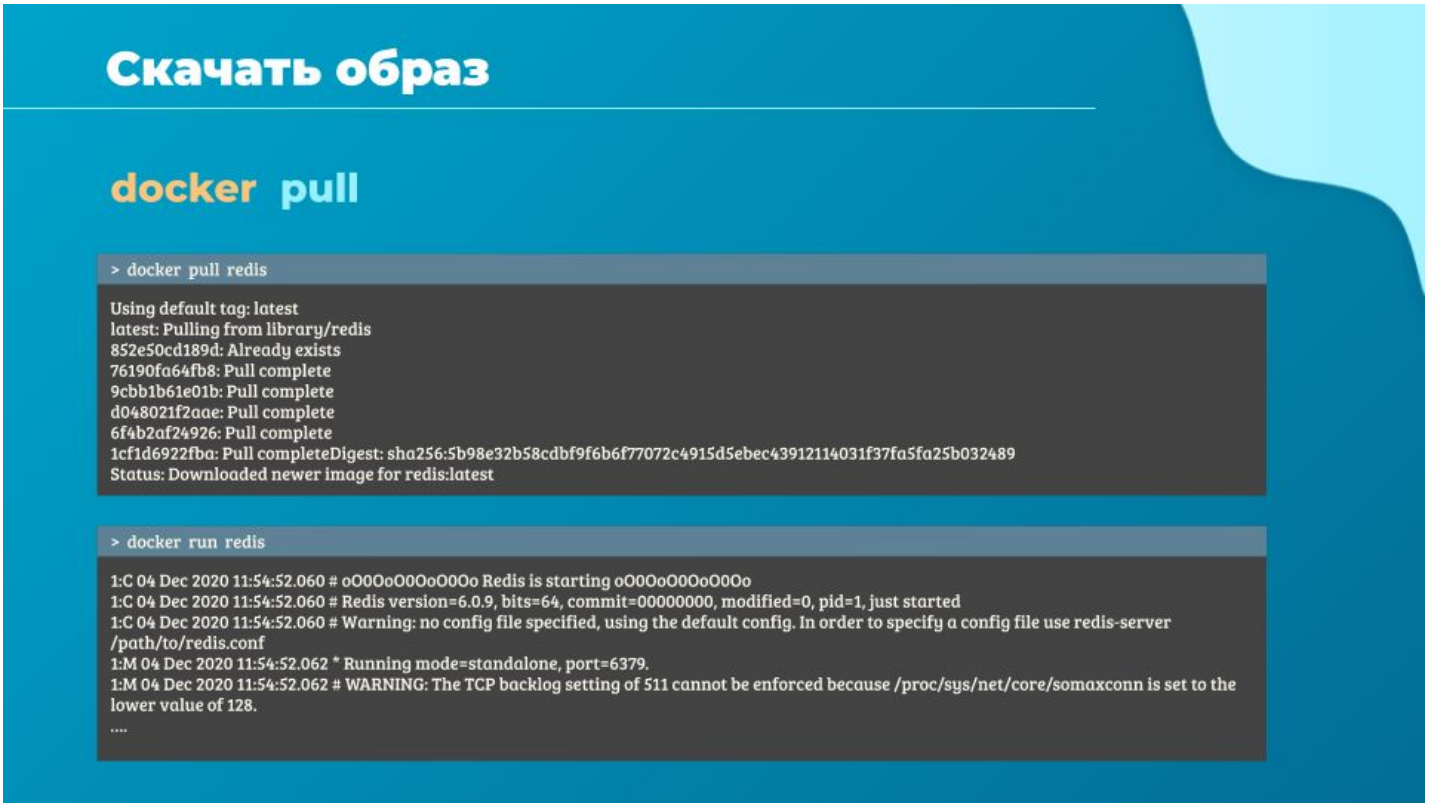
А что со скачанным образом nginx, который Docker скачал в начале? Он нам тоже не нужен, мы не собираемся его больше использовать. Нужно удалить этот образ, но для начала давай посмотрим

на список образов, представленных на нашем докер-хосте. Команда `docker images` покажет все доступные на хосте образы, их id, их теги, их размер.



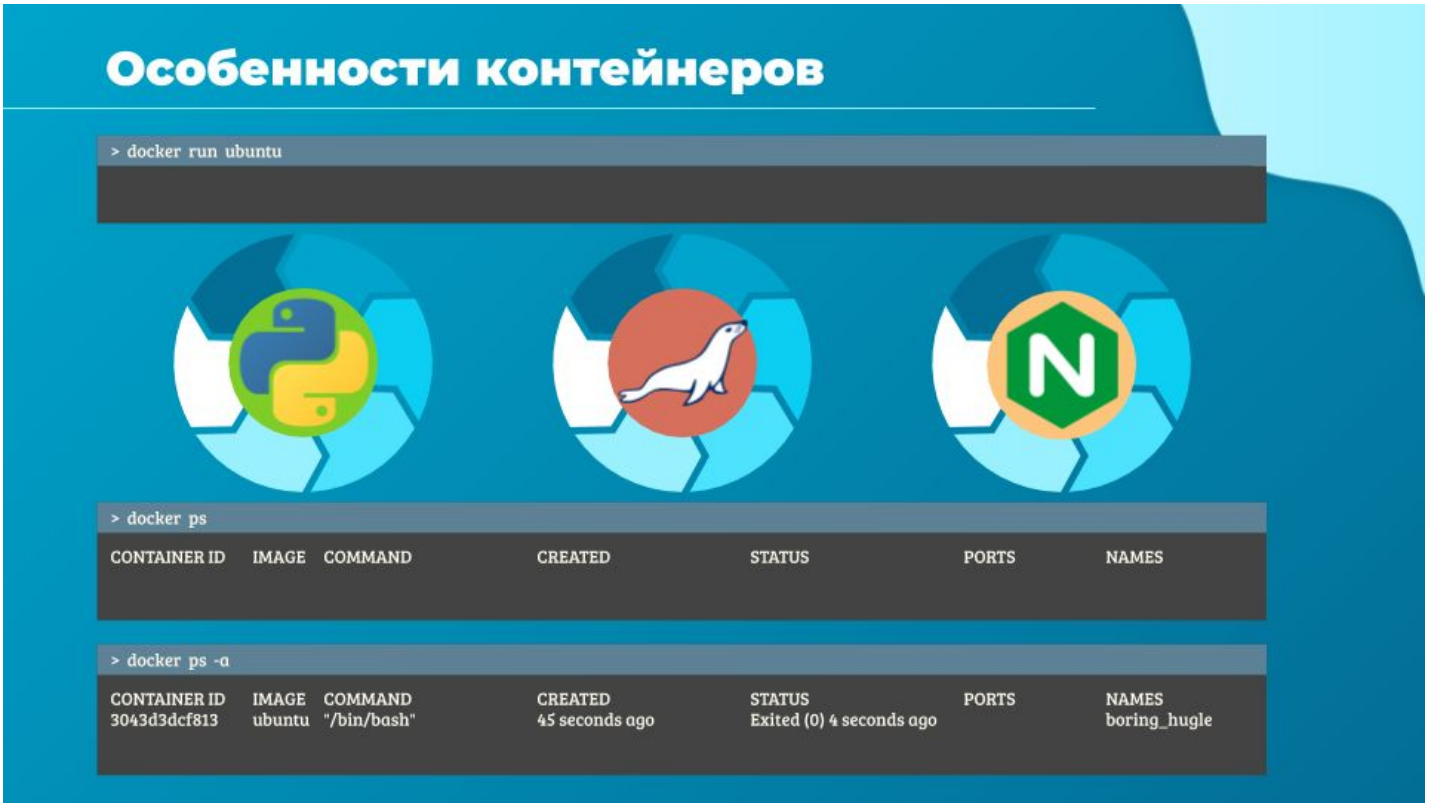
У нас представлено 5 образов локально: nginx, ubuntu, redis, weaveworks/scope и alpine. Чуть позже я расскажу про теги, а сейчас сфокусируемся на удалении образов. Запустим команду `docker rmi`.

Помни, перед тем, как удалить образ, тебе нужно убедиться, что контейнеры от этого образа не запущены в системе. Тебе нужно остановить и удалить все зависящие от этого образа контейнеры, прежде чем удалять сам образ.



Ранее я запустил команду docker run и она скачала образ redis, поскольку не нашла его локально на хосте. Что если мы просто хотим скачать образ и сохранить? Например для того, чтобы потом

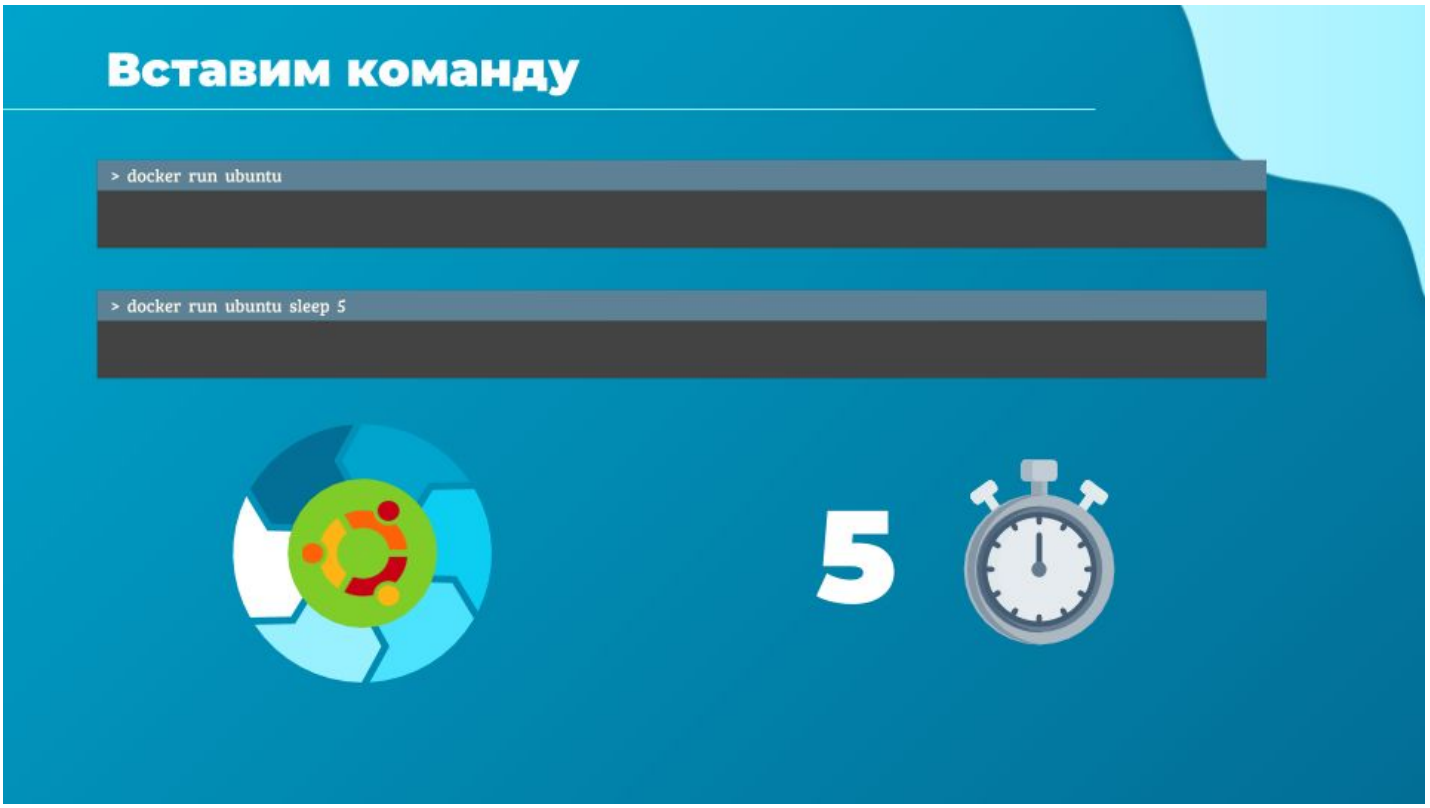
быстро запустить redis, не ожидая, пока его образ спуллится из репозитория. В этом нам поможет команда docker pull, которая просто просто скачает образ, но не будет его запускать.



Двигаемся дальше, если мы запустим контейнер из образа Ubuntu при помощи команды `docker run ubuntu` это запустит экземпляр ubuntu и немедленно выйдет из него. При просмотре списка работающих контейнеров мы его не увидим. Но если просмотреть список всех контейнеров, находящихся на хосте, включая остановленные, обнаружатся контейнеры с состоянием exited.

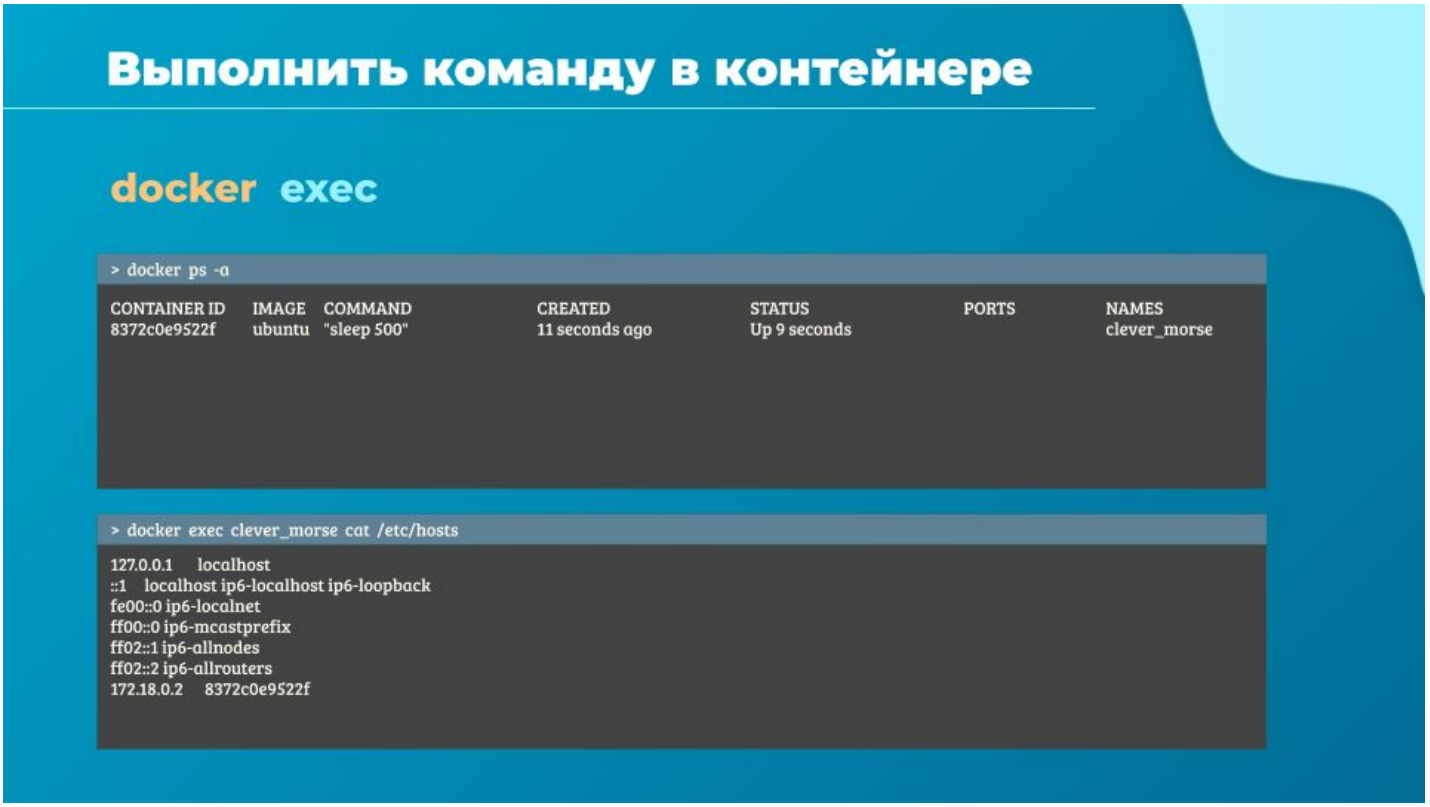
Итак, в этом еще одно отличие от виртуальной машины - контейнеры не предназначены для размещения ОС. Цель контейнера - выполнить работу и перестать пореблять ресурсы. Задания вроде анализа данных, расчета каких-то значений, размещение экземпляра БД или веб-сервера - эти процессы хорошо подходят для контейнеров.

Как только задача завершается, т.е. основной процесс внутри контейнера завершается, контейнер останавливается, становится exited. Получается контейнер живой, пока жив процесс, с которым запускался контейнер. Это важно. Если веб-сервер или база внутри контейнера скрашится или остановится, контейнер немедленно выйдет. Поэтому запустившийся контейнер с образом ubuntu моментально остановился. Потому что ОС сама является площадкой для запуска процессов, у нее не было процесса по умолчанию, того огонька, что поддерживал бы жизнь в контейнере.



Тем не менее, возможность запустить такой контейнер есть. Если в образе не запущена какое-либо приложение, как в случае с Ubuntu, мы можем попросить Docker запустить процесс с помощью команды `docker run`, например, команды сна с продолжительностью пять секунд.

При запуске контейнера в нем запустится команда sleep, которая будет работать 5 секунд. Она и будет тем процессом в контейнере, ради которого он живет. Когда команда завершиться, контейнер будет остановлен. А мы просто увидим на экране результат выполненной команды.

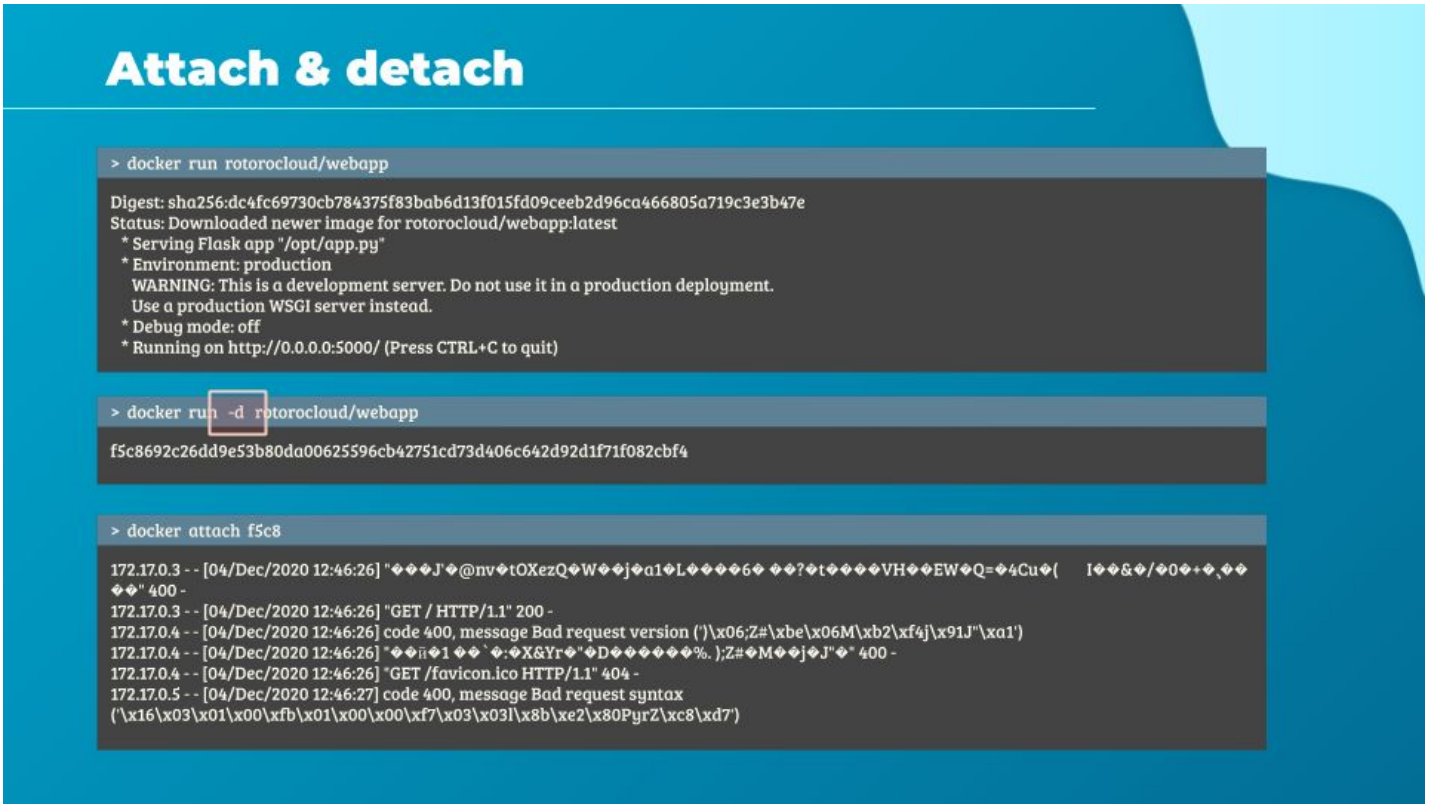


Ок, хорошо, мы научились запускать команду вместе с контейнером.
А как сделать так, чтобы выполнить команду на работающем контейнере?
Например, у меня выполняется контейнер с ubuntu.

Когда я делаю `docker ps -a`, то вижу, что его статус up, внутри него выполняется команда sleep 500. Я хочу увидеть содержимое файла внутри контейнера, как мне это сделать?

Используем `docker exec` для выполнения команд на моем работающем контейнере.

В данном случае я вывел содержимое файла /etc/hosts. Это файл из контейнера, а не ОС докер-хоста. Если просто написать `cat /etc/hosts`, то вывод будет другим, поскольку я обращусь к хостовой ОС, а не к контейнеру.



Перед тем, как перейти к практике, давай посмотрим еще на один момент. Для обучения у нас есть простое веб-приложение, его код есть на моем гитхабе, если тебе интересно. Его собранный образ лежит на docker hub в rotorocloud/webapp. Это приложение запускает небольшой веб-сервер, который доступен на порту 5000.

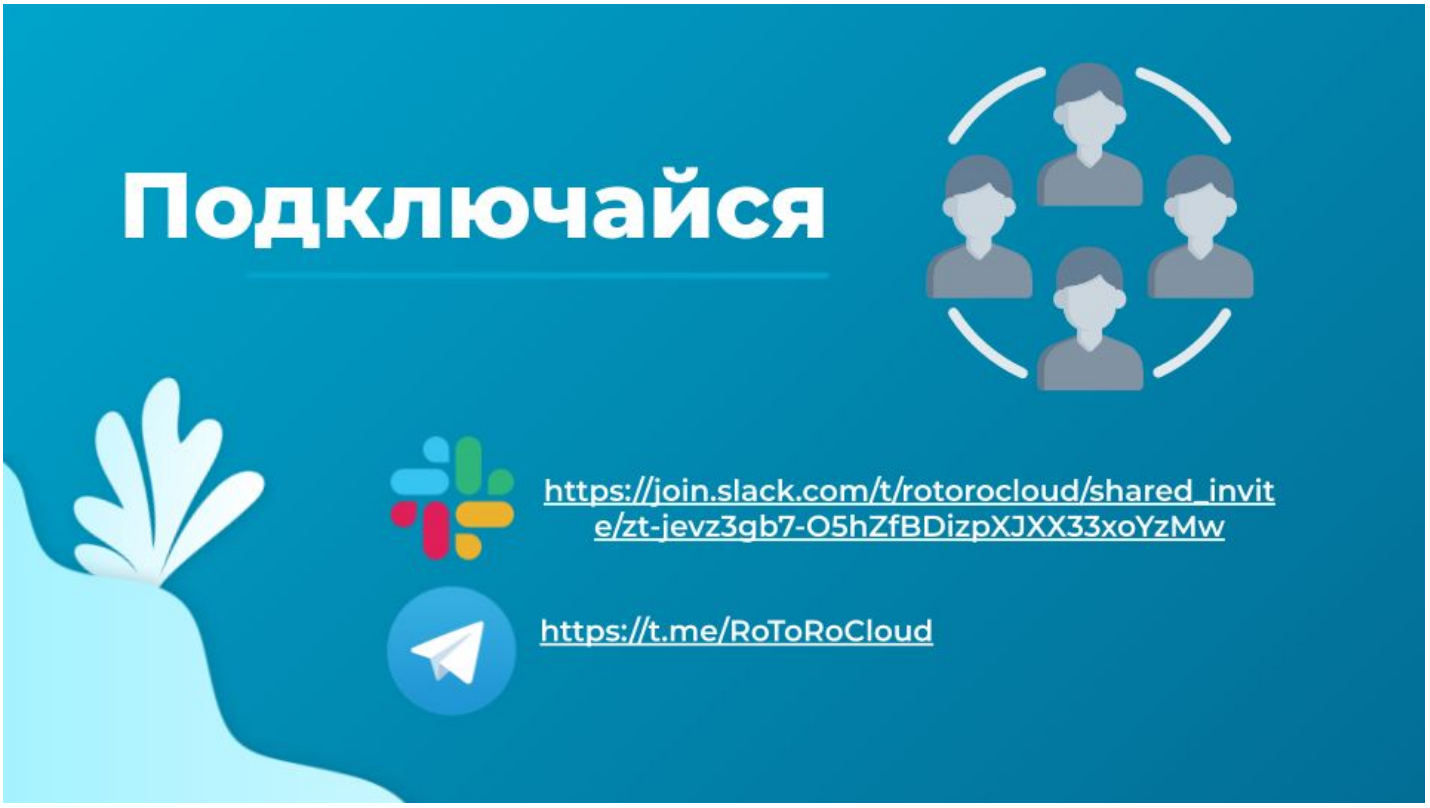
После запуска команды `docker run rotorocloud/webapp`, приложение может быть развернуто в фоне или прикреплено к твоей консоли. Последнее значит, что стандартный output контейнера будет перенаправлен на твой терминал и ты будешь видеть весь вывод процесса веб-сервера на своем экране. Ты не сможешь никак взаимодействовать с консолью пока она прикреплена к приложению до тех пор, как контейнер не остановится. Для принудительной остановки используй комбинацию CTRL + C. Это остановит контейнер, приложение перестанет выполняться и ты попадешь обратно в приглашение своей ОС.

Есть другой способ запустить докер-контейнер - использовать detached mode. Пишется как -d. Это запустит контейнер в фоновом режиме и ты сразу же сможешь взаимодействовать со своей оболочкой. Контейнер продолжит свое выполнение, чтобы проверить это запусти команду `docker ps`. Ты в любой момент можешь снова прикрепить контейнер к консоли запустив команду `docker attach` с указанием ID или имени контейнера.

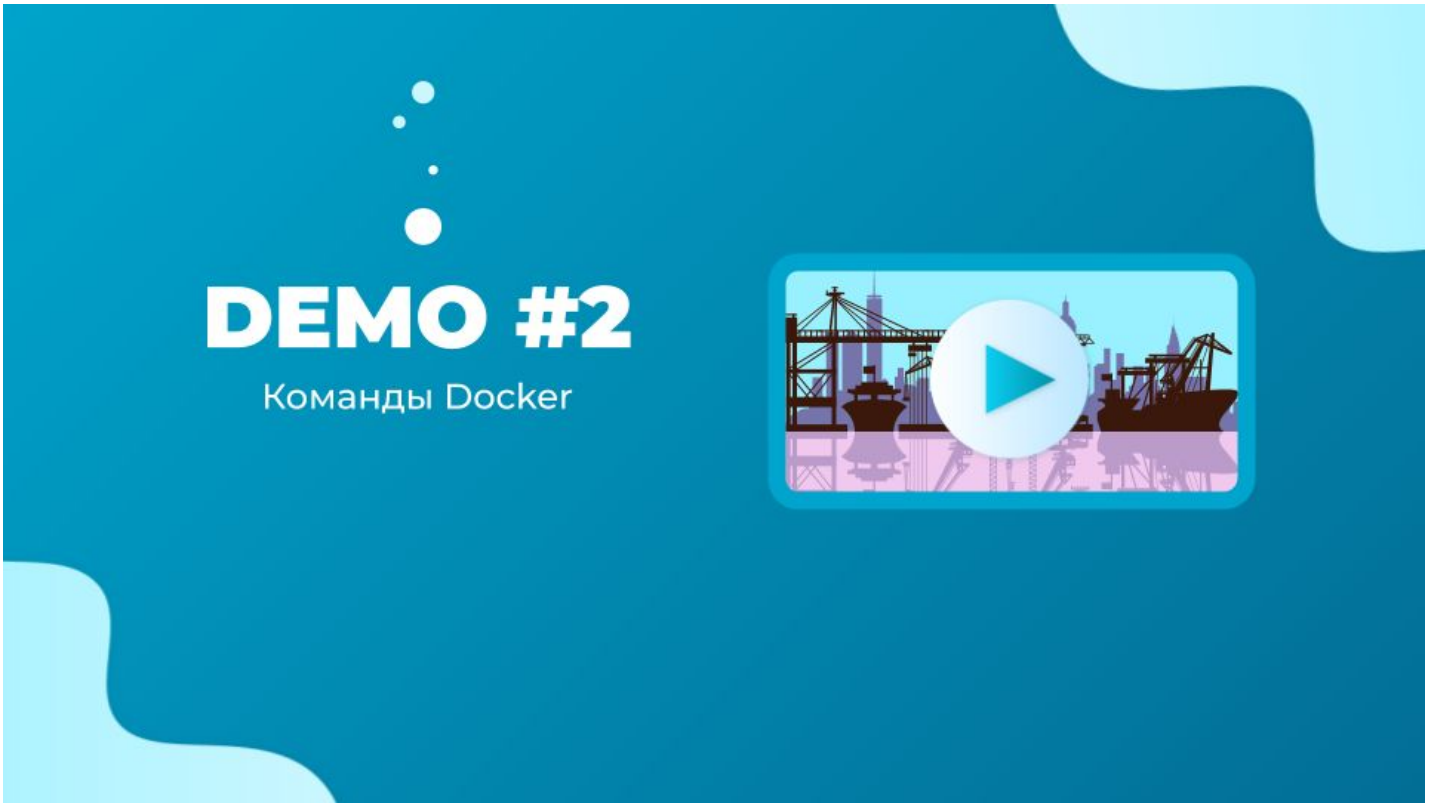
Обрати внимание, что при использовании ID не обязательно писать такие длинные строки, достаточно указать первые несколько символов любого ID. В моем случае f5c8. Однако, если на

хосте в данный момент есть дублирование первых символов в каких-то ID, то тебе придется указать больше символов, например пять и т.д.

В данный момент нам не интересна работа самого веб-сервера, нас интересует, как запустить контейнер и как это выглядит из консоли. К работе приложения мы еще вернемся в следующих лекциях.

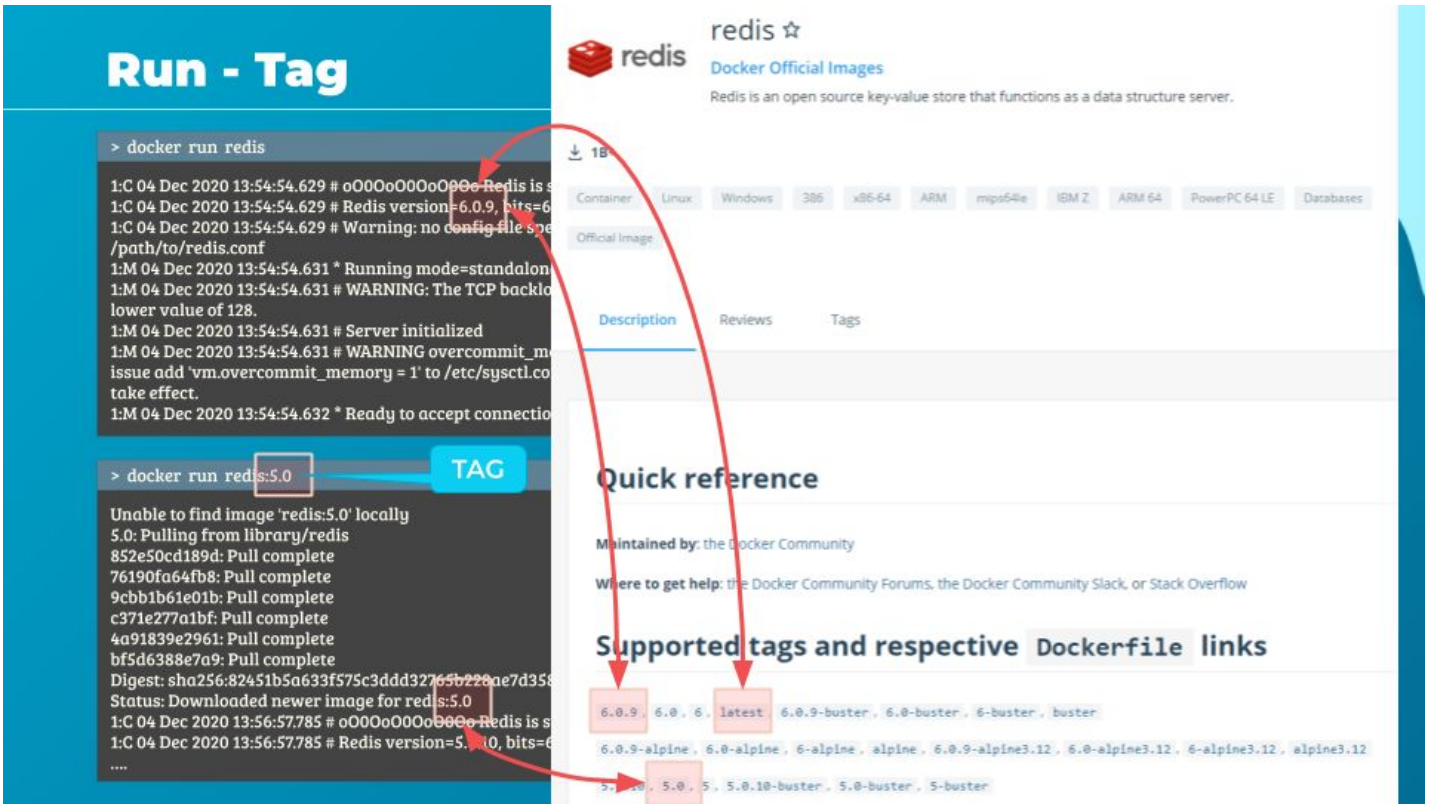


Технологии быстро меняются, и если что-то в курсе устарело, или я где-то спешу, или что-то непонятно объясняю, пожалуйста напиши мне в Слак или Телеграм.



Переходим к демо, где я запущу эти команды и покажу наше лабораторное окружение.

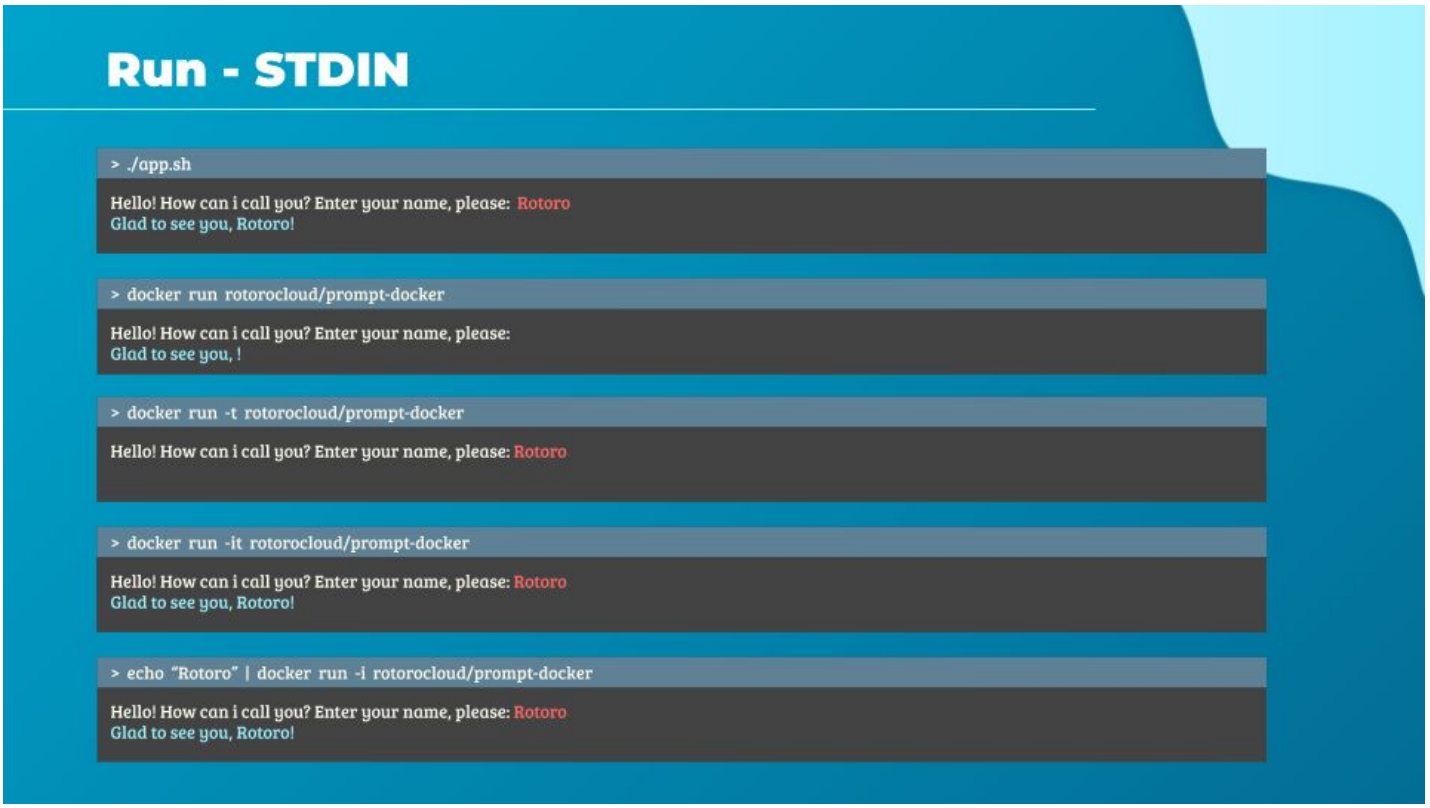
Привет, и добро пожаловать на лекцию в которой мы подробнее рассмотрим особенности команды `docker run`. Это одна из наиболее часто используемых команд и у нее, наверное, больше всего параметров для запуска. Этот ряд особенностей тебе необходимо будет знать и уметь применять на практике, если ты хочешь, чтобы Docker облегчил твою жизнь. Мы попрактикуем это на лабораторной.



Ок, мы уже умеем запускать контейнеризированный Redis командой `docker run redis`.
В этом случае запустился Redis версии 6.0.9. Но что если мне нужна другая версия? Скажем, я хочу запустить Redis версии 5.0.

Для этого надо указать версию через разделитель - (:) двоеточие после имени образа. Это называется tag. В этом случае Docker скачает образ Redis версии 5.0 и запустит его.
Обрати внимание, если ты не указал тег, то Docker будет искать образ с тегом "latest". Тег latest обычно связывают с последней версией этого ПО и за его установку ответственны создатели продукта. Но как нам, как пользователям, найти информацию какие версии доступны у приложения и какая из них последняя?

Поискать на https://hub.docker.com/. Найдем имя образа redis и список применимых к нему тегов, а также описания каждой версии для соответствующего тега. Каждая версия может иметь несколько коротких и длинных тегов, ссылающихся на нее. Как видишь версия 6.0.9 также еще и имеет тег latest.



Теперь давай поговорим о вводе. Для пояснения у меня есть маленькое приложение, оно доступно на докерхабе, если захочешь запустить. Итак, на первом шаге у меня был простой шелл-скрипт, который при запуске спрашивал имя, а после ввода писал на экране приветствие и те данные, которые ему передали при вводе.

Я произведу некоторые манипуляции, о которых поговорим чуть позже, и сделаю из этого скрипта докеризированное приложение. При его запуске таким образом: `docker run `rotorocloud/prompt-docker` мы не получим приглашение к вводу. Приложение напишет только то, что жестко прописано в скрипте, т.к. оно не получило никаких данных. Это произошло потому, что по умолчанию докер-контейнеры не слушают стандартный ввод. Даже с присоединенной консолью он не сможет прочитать введенные тобой данные. У него нет терминала для чтения ввода, он работает в неинтерактивном режиме.

Давай добавим возможность ввода, подключив терминал, откуда мы эти данные введем. Чтобы это сделать используй параметр -t (-t означает использование псевдотерминала). В этот раз приложение сразу не вышло. Я получил приглашение на ввод данных, но что-то идет не так - когда пытаюсь ввести имя и нажать ввод ничего не происходит. Не работают даже клавиши, останавливающие приложение. Что же произошло?

ОС внутри контейнера видит мой терминал, а команда, выполняющаяся в данный момент ждет с него данные. Проблема в том, что мы не связали стандартный ввод (stdin) своего хоста с stdin контейнера. И контейнер изолирован в части получения данных из стандартного ввода. Команда останется в вечной петле ожидания, пока мы не убьем контейнер.

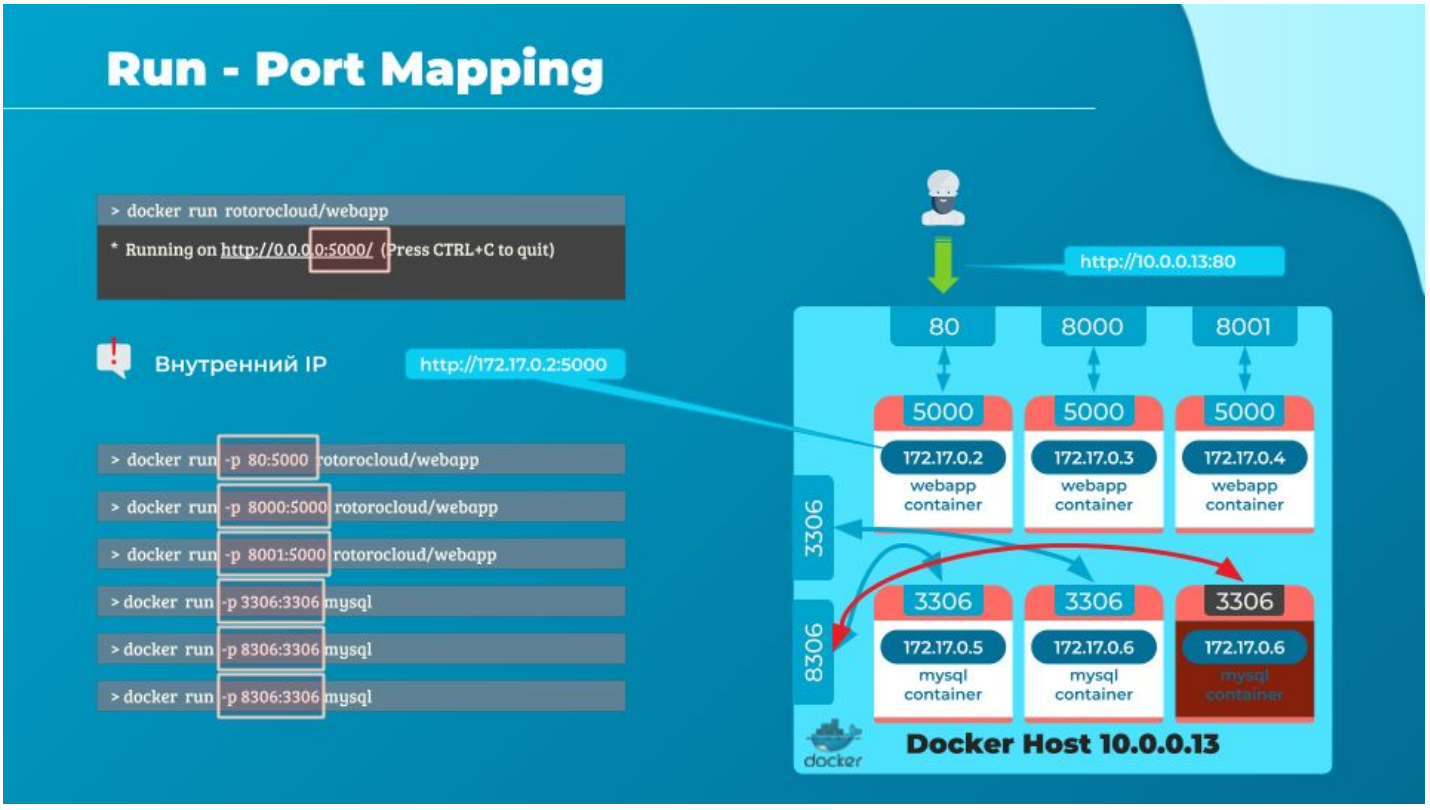
Чтобы контейнер стал принимать данные на свой стандартный ввод с терминала или какого-либо pipe используй параметр -i. Параметр -i включает интерактивный режим, и когда мы вводим свои данные, они успешно поступают на ввод нашего контейнера. Теперь мы можем ввести данные с клавиатуры, а команда их получит, и мы увидим ожидаемый вывод.

Итак, комбинация -it дала нам возможность подключится и взаимодействовать с контейнером в интерактивном режиме. Хочешь подключиться к контейнеру - будь из IT, легко запомнить :)

Параметр -it нужен для непосредственного подключения человека, потому что терминал дает возможность контейнеру получить некоторые управляющие команды, а также обеспечивает дополнительные возможности вывода на консоль пользователя. В случае с автоматизацией обычно достаточно -i.

Вот случай автоматического pipe, команда:
`echo "Rotoro" | docker run -i rotorocloud/simple-prompt-docker`

Команда echo и контейнер в пайпе сами договорятся без всяких терминалов.



Самое время разобраться с публикацией портов или как их еще называют пробросом портов в контейнер. Давай вернемся к примеру с нашим веб-сервером на 5000 порту, с которым мы познакомились в предыдущей главе. Запустим его на нашем докер-хосте. Как ты помнишь, хост, на который установлен Docker в части запуска контейнеров, называется Docker host или Docker engine.

Когда мы запускаем контейнеризированное веб-приложение, оно начинает работать, и мы видим, что сервер запустился. Но как нам дать доступ пользователям к этому приложению?

Как видишь, мое приложение слушает на порту 5000, я могу получить доступ через этот порт. Но какой IP мне использовать в своем браузере?

Тут есть два варианта.

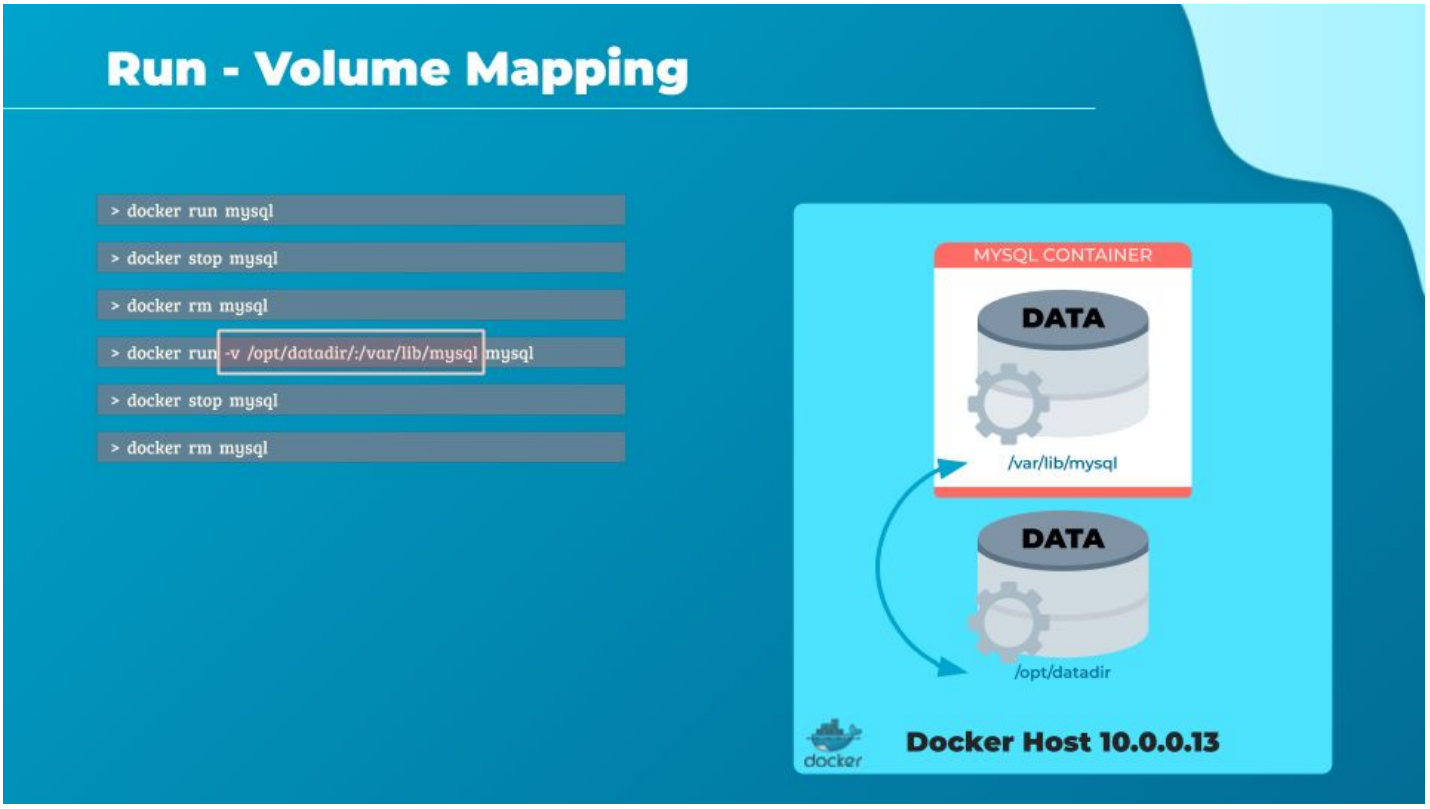
Первый использовать IP докер-контейнера, т.к. у каждого контейнер есть свой собственный адрес, каждый докер-контейнер получает IP присвоенный по умолчанию. В данном случае 172.17.0.2. Но помни, что этот адрес внутренний и будет доступным только внутри докер-хоста, где контейнер выполняется. В общем, если ты запустишь на своем докер-хосте браузер, введешь в нем http://172.17.0.2:5000 и получишь доступ. Но пользоваться этим сервисом смогут только внутренние пользователи этого хоста. Т.к. это внутренний IP внешние пользователи не смогут использовать этот URL.

Для внешних запросов мы могли бы использовать IP, по которому внешние пользователи ходят на наш хост, 10.0.0.13 в нашем случае. Но для того, чтобы это заработало, мы должны сопоставить порт внутри контейнера со свободным портом на докер-хосте. Допустим, я хочу, чтобы пользователи получили доступ к моему приложению на порту 80 нашего докер-хоста. Тогда мне нужно сопоставить порт 80 localhost с портом 5000 в контейнере Docker.

Для этого используй параметр -p в команде `docker run`. И пользователи смогут получить доступ к приложению по URL http://10.0.0.13:80. Весь трафик, полученный на этот порт 80 будет маршрутизирован на порт 5000 внутри контейнера. Таким манером ты можешь запускать много экземпляров разных приложений и сопоставлять их различным портам на докер-хосте.

Например, я развернул экземпляр mysql, который запустил БД и начал слушать на дефолтном порту mysql 3306. Также я запустил еще один инстанс базы, смаппив его порт 3306 на порт 8306 хоста. Итак, ты можешь запустить столько приложений, сколько тебе нужно и пробросить столько портов, сколько тебе требуется. Разумеется есть ограничение в том, что ты не можешь назначить один порт докер-хоста несколько раз.

Мы еще вернемся к проброску портов и сетевом взаимодействии в лекции о сетях в Docker. А сейчас давай познакомимся с тем, как нам хранить постоянные данные при использовании контейнеров.



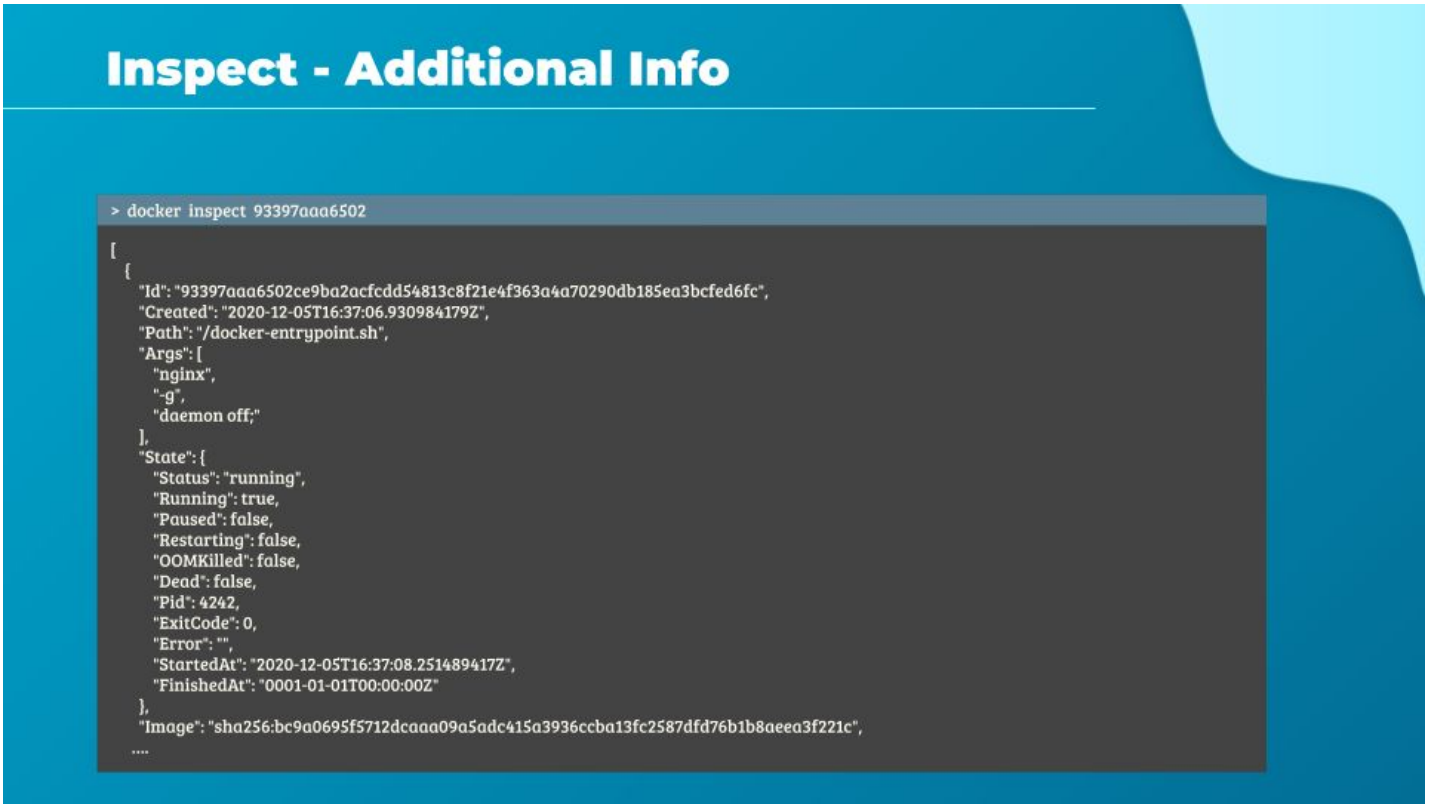
Допустим у нас запущен контейнер с mysql. Когда базы и таблицы будут созданы, физически они будут сохранены по по пути /var/lib/mysql в докер-контейнере. Запомни, что контейнер имеет свою собственную изолированную файловую систему и любые изменения происходят только в ней, они не имеют отношения к файловой системе хоста. Теперь мы зальем большой дамп данных в эту базу. Что произойдет если мы удалим этот mysql-контейнер?

Как только произойдет удаление, мы потеряем все данные. Это связано со слоистой структурой контейнеров, о чем мы поговорим позже в этом курсе. Как же хранить постоянные данные в контейнерах?

С помощью специальной директории, размещенной на докер-хосте, данные в которой не будут относится непосредственно к контейнеру, а будут подключены как бы извне. Как будто внешний жесткий диск. Нам потребуется связать две директории: одну на хосте, а вторую в контейнере. В этом случае я создал директорию /opt/datadir и пробросил ее в контейнер по размещению /var/lib/mysql.

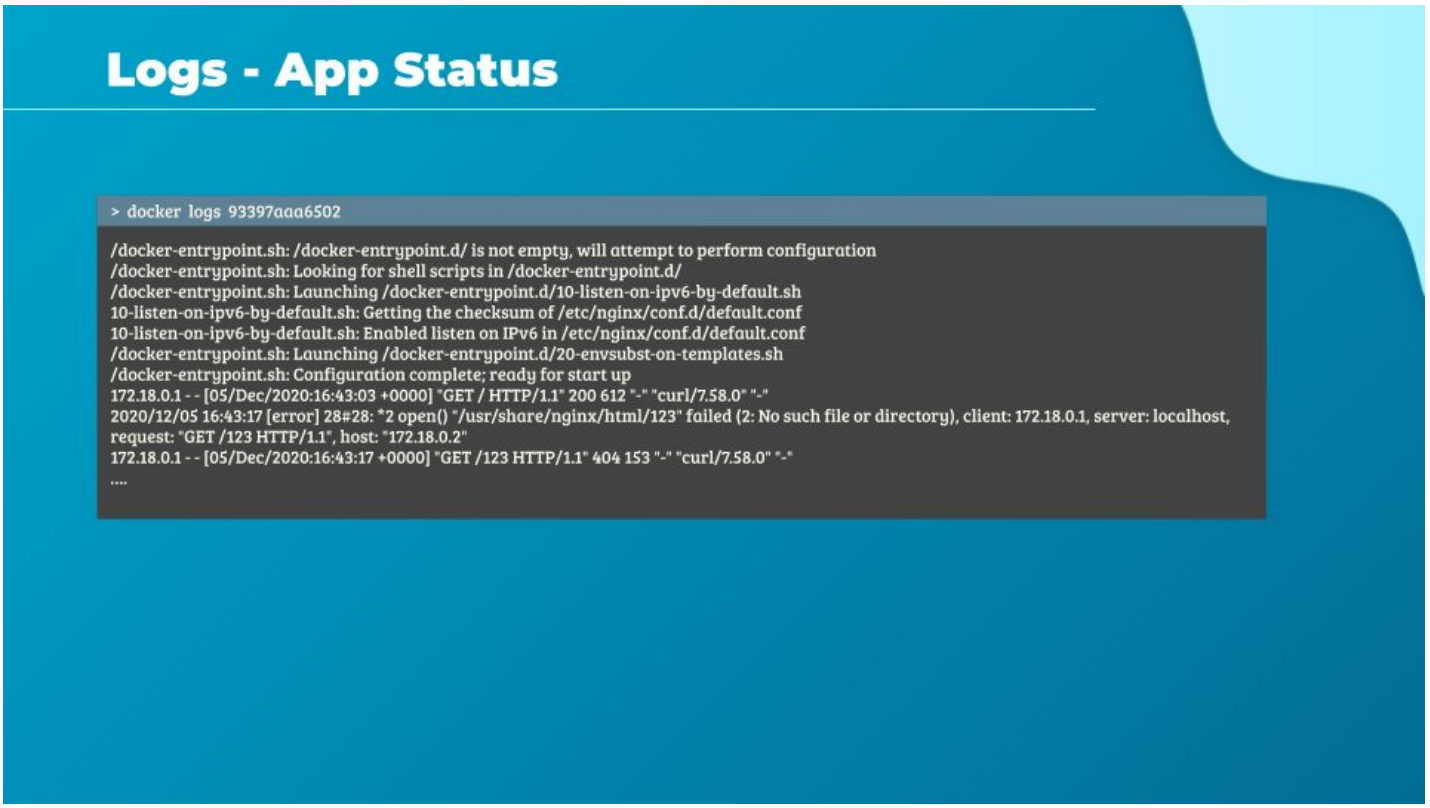
Для создания связи используется параметр -v, а далее пишется путь до директории на хосте, потом двоеточие-разделитель и папка внутри контейнера. Сначала откуда - с хоста, потом куда - в контейнер. Легко запомнить.

Таким образом, при запуске докер-контейнер неявно монтирует внешнюю директорию в свою внутреннюю. Теперь все данные спокойно лежат во внешнем томе в директории /opt/datadir докер-хоста и не зависят от жизни контейнера. Они останутся там, даже если контейнер будет удален.



Команда `docker ps` хорошо подходит для быстрого сбора основные деталей о контейнерах в нашей системе. Таких как имена, ID и т.д. Но периодически нам будет требоваться развернутая информация о каком-то контейнере. В подобных случаях нас выручит команда `docker inspect`, она запускается с именем или ID контейнера.

Она вернет нам детальную картину этого контейнера в JSON формате. Его состояние, что смонтировано, как сконфигурировано, настройка сети и т.д. Не забудь про нее, когда тебе нужно будет узнать детали запущенного и забытого кем-то контейнера или в похожей задаче.

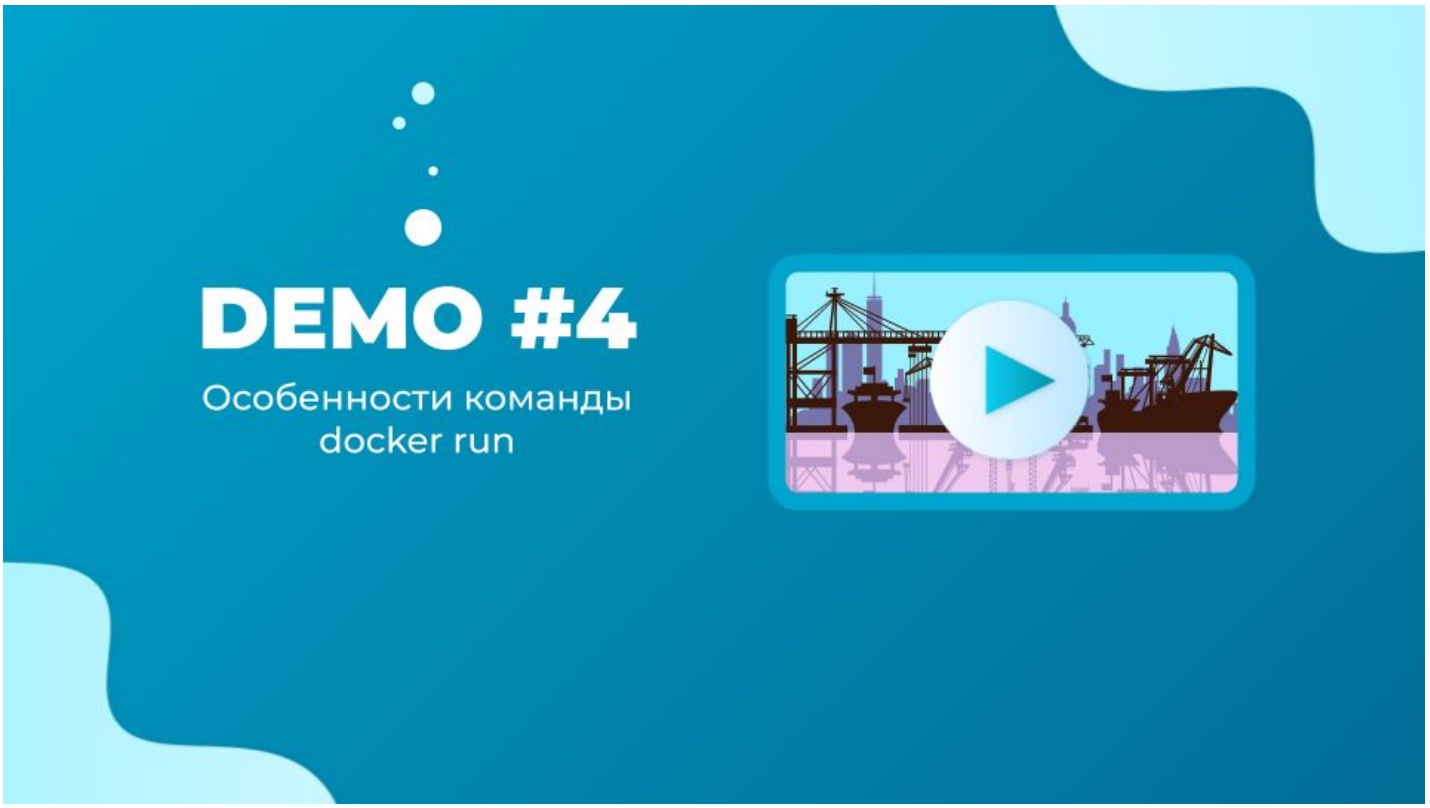


И наконец траблешутинг. Когда что-то идет не так, мы смотрим логи. Как их смотреть в контейнерах?

С помощью команды `docker logs`. Чтобы посмотреть эту команду давай запустим наше веб-приложение в фоновом режиме используя параметр -d.
`docker run -d rotorocloud/webapp` запустит его в detached mode.

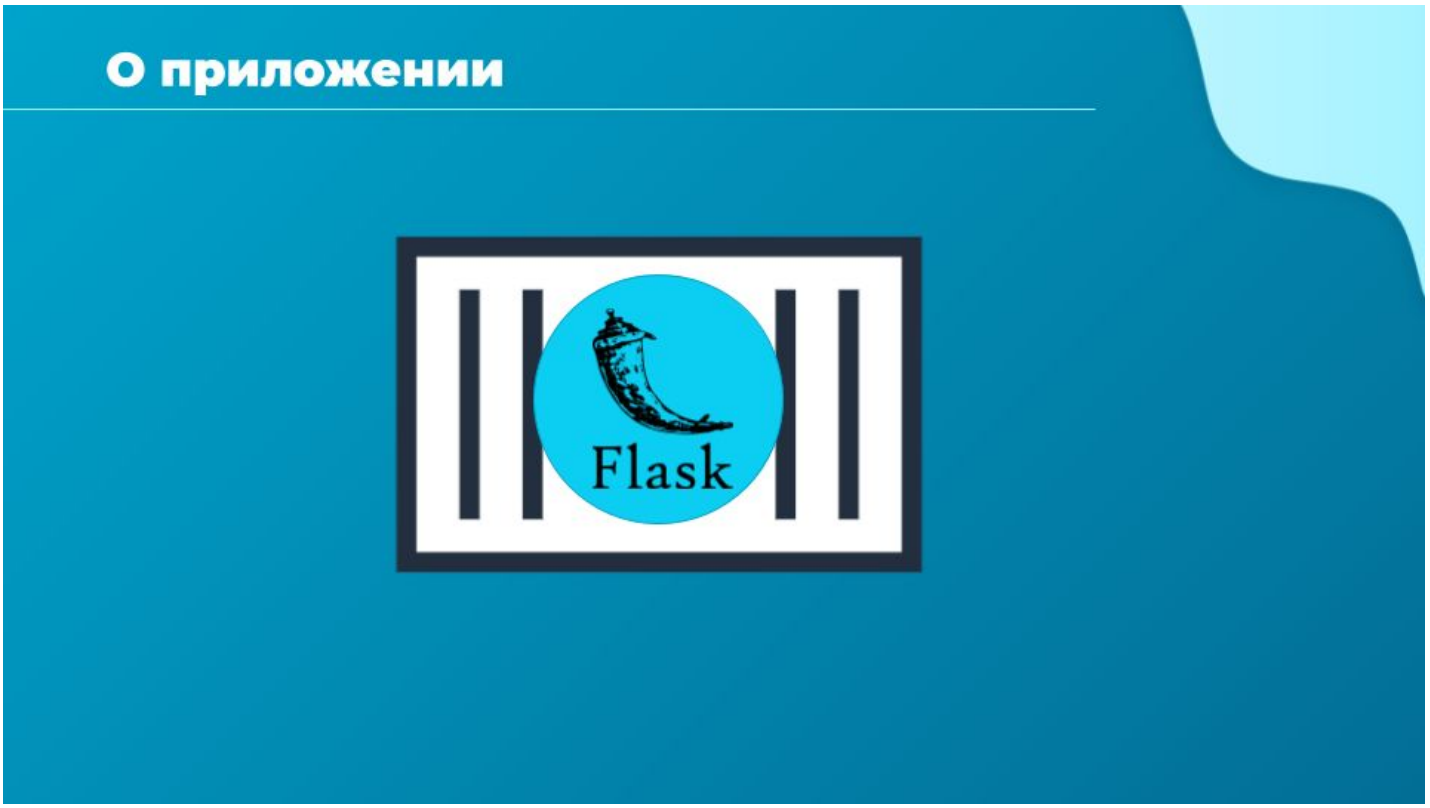
Теперь запустим команду `docker logs`, а дальше имя контейнера или его ID. Мы видим журнал, сформированный из stdout того контейнера.

В мире разработки контейниризированных приложений часто практикуется не хранить логи в файлах, а передавать информацию на стандартный вывод, где ее собирает агент мониторинга. Об этом мы подробно говорим в курсе разработчика приложений для Kubernetes.



Это все в этой лекции. Давай перейдем к демо и практическим упражнениям.

Привет и добро пожаловать в лекцию об образах Docker.

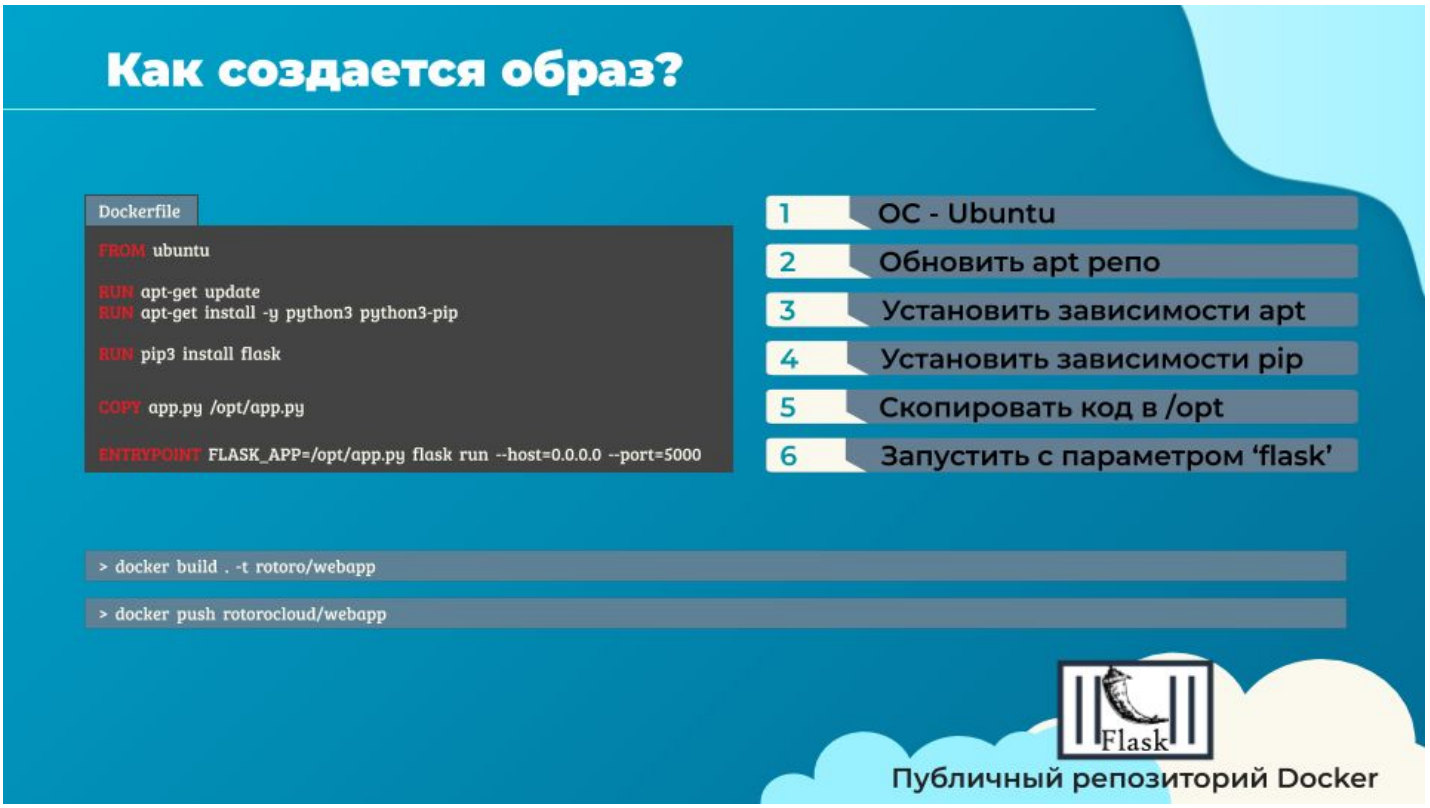


Итак, мы решили создать свой собственный образ. Но для начала давай определимся зачем мы это делаем, зачем нам свой образ?

Возможно мы не можем найти какой-то компонент или службу для своего приложения. Или мы хотим контейнеризировать свое приложение для обеспечения простоты тестирования, доставки и развертывания. А быть может мы собрали несколько уникальных окружений для компиляции исполняемых файлов и используем это в своих CICD-конвейерах. Перечислять можно много, важно, что нам не обойтись готовым решением из публичных репозиториев.

В этом случае я собираюсь контейнеризировать свое простое веб-приложение. Оно написано на python с фреймворком flask. Вначале нужно понять, что мы положим в контейнер или с каким приложением мы собираемся создавать образ.

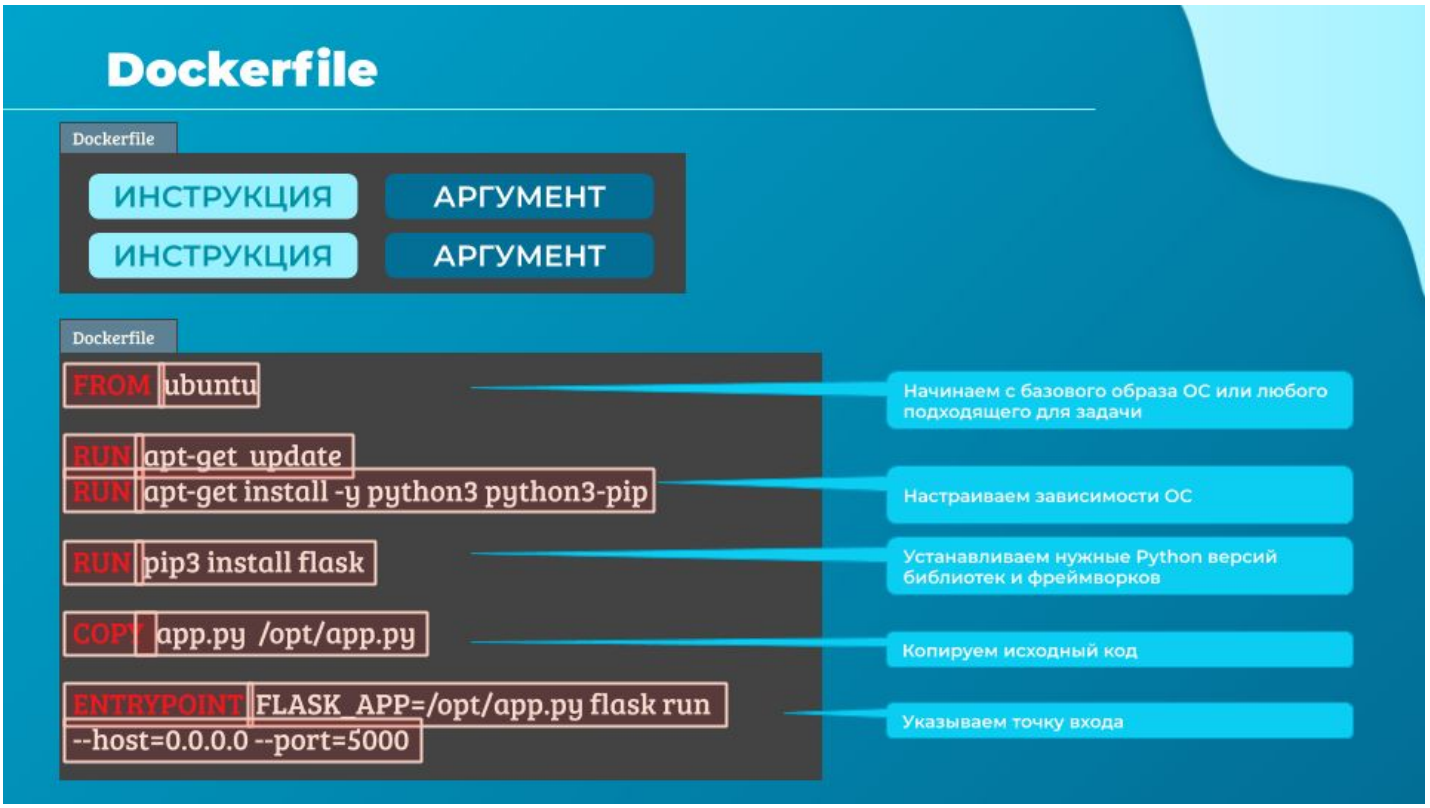
Давай поразмышляем, как бы это было, если бы мы развёртывали это приложение вручную. Записав требуемые шаги в правильном порядке мы создадим план для создания нашего простого приложения.



Итак, для начала в ручном режиме я бы установил себе операционную систему, например ubuntu. Далее я бы обновил apt-репозитории используя `apt update`. После этого с помощью `apt install python` установил бы python с соответствующими зависимостями и утилитами. Затем с помощью `pip install flask` создал бы необходимый для этого фреймворка базис. Осталось скопировать мой исходный код в нужную папку, например /opt/. Последний шаг - запустить код нашего веб-сервера, используя специфичную для flask команду. Ок, теперь с помощью этих инструкций мы запросто создадим свой первый Dockerfile.

Вот краткий обзор процесса создания собственного docker-image:
создай файл с именем Dockerfile и запиши туда инструкции по настройке своего приложения, такие как установка зависимостей, место, куда копировать исходный код приложения, какая точка входа будет использована и т.д. После этого, создай свой образ при помощи команды `docker build`, указав Dockerfile и теги для образа. Это создаст локальный докер-образ в твоей системе.

Чтобы сделать его доступным в публичном докер-репозитории, вроде Docker Hub используй команду `docker push` command с указанием имени только что созданного образа. В этом случае команда будет содержать имя моего аккаунта на Docker hub - rotorocloud, а дальше имя образа - webapp.



Давай познакомимся поближе с этим Dockerfile. Он пишется в специальном формате, который понятен Docker и состоит из инструкций и аргументов.

Все, что в Dockerfile написано слева заглавными буквами будет считаться инструкцией. Я выделил эти места красным цветом текста. В нашем случае это: FROM, RUN, COPY, ENTRYPOINT. Все они являются инструкциями. Каждая из них указывает Docker выполнить определенное действие в процессе создания образа.

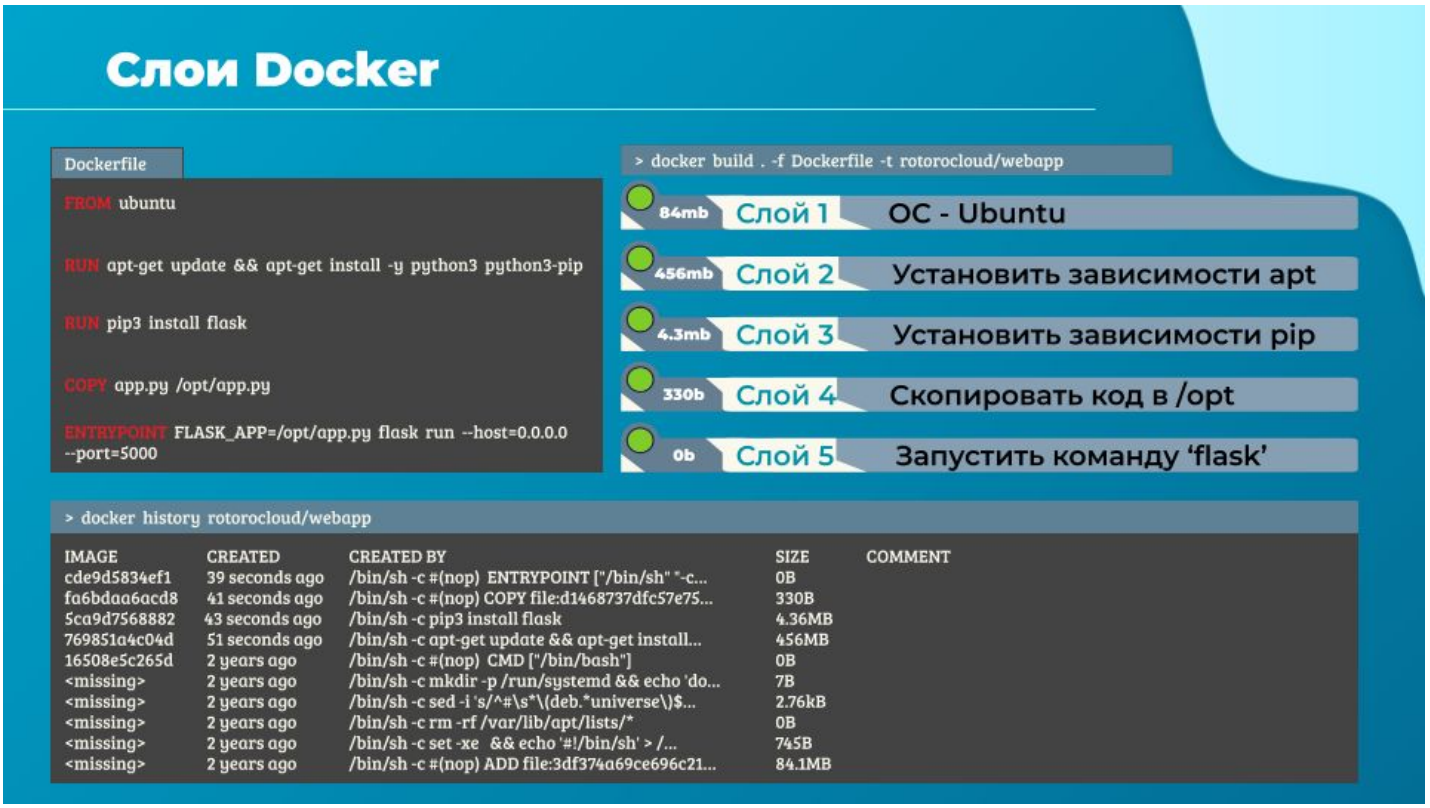
Все, что правее инструкций - это аргументы. Т.е. в первой строке FROM - инструкция, а Ubuntu - аргумент.

Наша первая строка определяет, какая базовая ОС будет использоваться для строительства образа и запуска контейнера из него в дальнейшем. Каждый докер-образ базируется на подобном базовом image. Т.е. на образе ОС или другом образе, который был создан на основе образа ОС. Официальные релизы популярных ОС размещены на Docker Hub, ты их сможешь легко найти для своих экспериментов. Еще раз отмечу, что все Dockerfiles начинаются с инструкции FROM, не забывай об этом.

Инструкция RUN дает указание Docker запустить данные тобой команды на этом базовом образе. Мы видим, что Docker запустил команду apt-get для получения обновленных пакетов и установки их на наш базовый "голый" образ Ubuntu. Такой же инструкцией RUN он запустил команду pip install, тем самым установив зависимости для python. Напомню, что все действия производятся в изоляции и к нашей хостовой ОС отношения не имеют.

После завершения инструкций RUN, инструкция COPY копирует файлы из директории нашей локальной ОС в этот собирающийся образ. В этом случае исходный код нашего python-приложения находится в папке, из которой мы запустили команду `docker build` и мы скопируем его в расположение /opt внутри образа Docker.

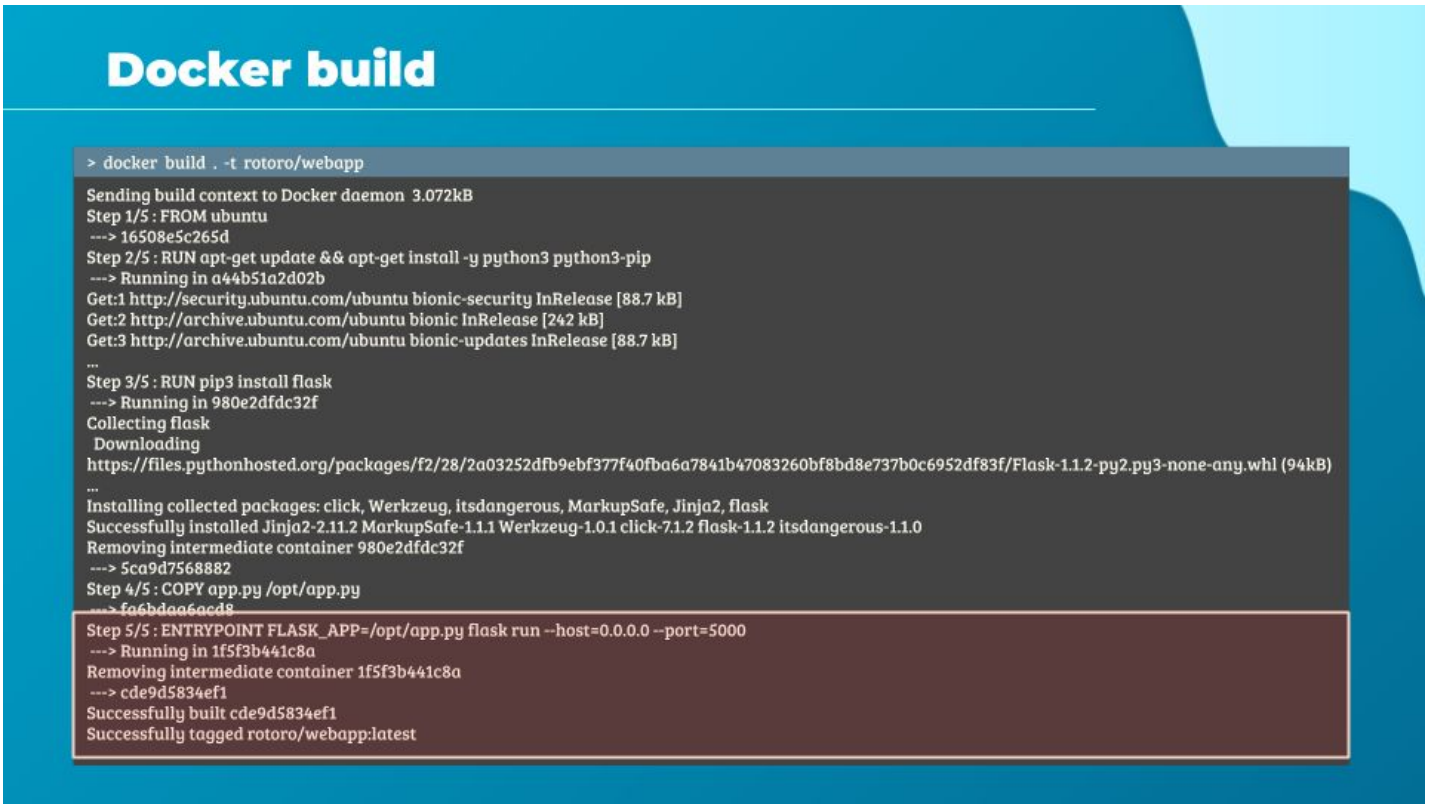
И, наконец, последняя здесь инструкция ENTRYPOINT. Она позволяет нам указать команду, которая сработает, когда образ будет запущен в качестве контейнера.



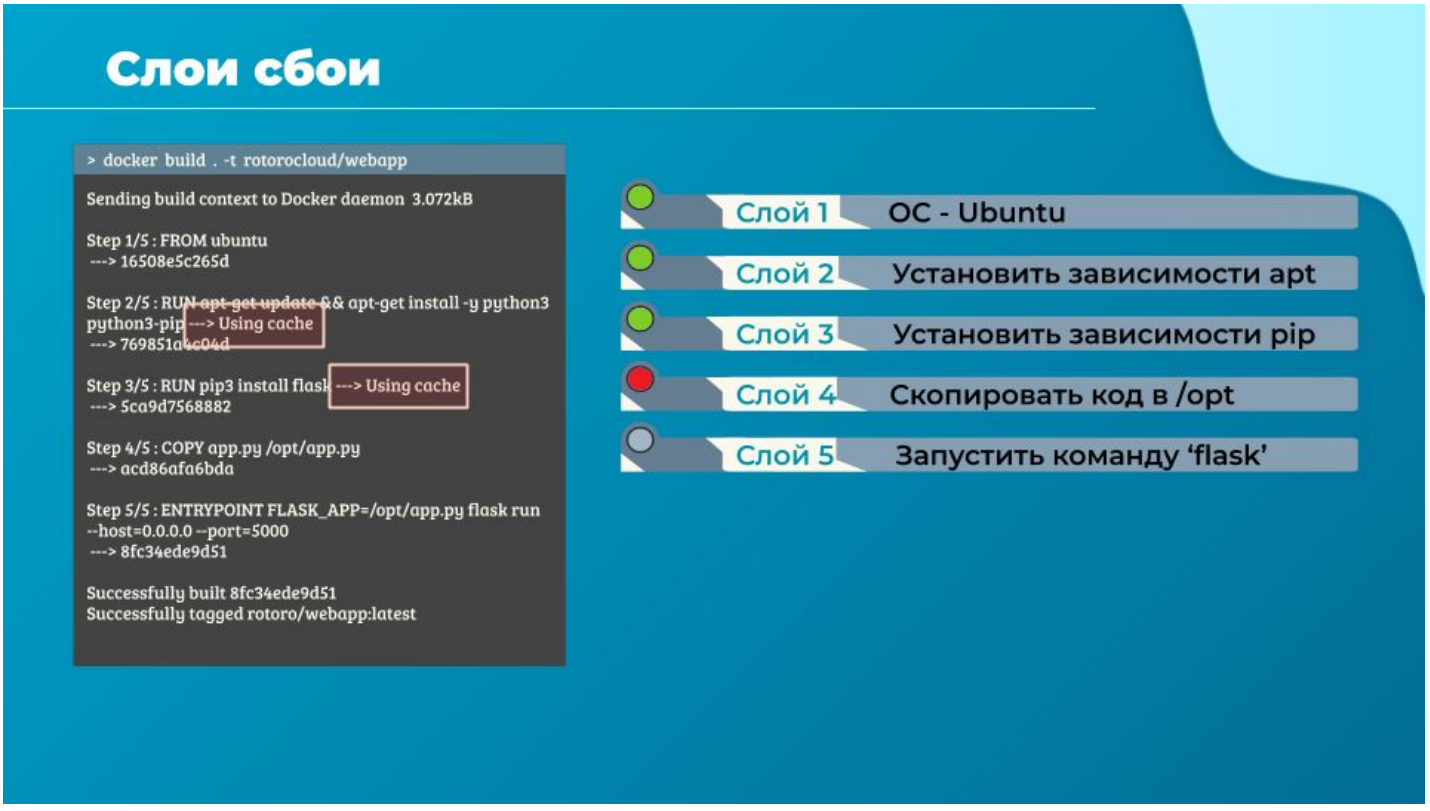
Когда Docker создает образы они строятся по слоеной (layered) архитектуре. Каждая новая инструкция создает новый слой в докер-образе, который хранит в себе отличия от предыдущего слоя.

Например, за первым слоем базовой ОС Ubuntu следует инструкция, которая создаст второй слой, в котором будет установлены apt-пакеты, за ним последует третий слой, который создаст третья команда, установившая пакеты python. Четвертый слой скопирует исходный код и последний пятый слой обновит точку входа этого образа.

Поскольку каждый слой хранит только изменения предыдущего слоя, это значительно отражается на размере. Взгляни, базовый образ Ubuntu имеет размер в районе 84mb. Apt установит нужные нам пакеты - это еще около 456mb. Оставшиеся слои еще меньше. Ты можешь посмотреть информацию о слоях с помощью команды `docker history` указав имя образа, который создавал.



Во время сборки образа команда `docker build` выводит все шаги, которые она делает, какие команды запускает и их результат. Все слои кэшируются, поэтому слоеная архитектура помогает тебе как в случае ошибок при сборке, так и при проведении повторной сборки образа. Тебе не нужно начинать все сначала - Docker перестроит только те слои, в которые были внесены изменения.
У каждого слоя есть свой хеш, по которому Docker отслеживает их уникальность.



Например, при сборке на четвертом шаге произошел сбой. Мы разобрались и уточнили причину сбоя. Теперь мы перезапустим сборку и увидим, что старые три слоя были переиспользованы из кэша, а фактическое исполнение команд Docker начал только с четвертого шага.

Тоже самое будет, если ты изменишь одну из команд - низлежащие под ней слои (т.е. слои, которые были выполнены до этой инструкции) не будут затронуты, но будет перестроен слой измененной команды и все слои после нее. Эти новые слои также добавятся в общий кэш докера. Это позволяет создавать образы быстрее, тебе не нужно ждать команды, которые уже до этого были выполнены.

Это очень полезно при обновлении исходного кода приложения. Обычно исходный код в контейнере подвергается изменению гораздо чаще, чем библиотеки и зависимости, поэтому копирование файлов приложения находится обычно в конце Dockerfile.

Мы уже видели ряд продуктов, содержащихся в контейнерах, таких как операционные системы, системы управления базами данных и т. д. Но это еще не все. Ты можешь создавать контейнеризированные версии практически любых приложений от серверных высоконагруженных до клиентских повседневных. Контейнерные версии бизнес-приложений тоже уже очень популярны. Это могут быть большие приложения или простые утилиты, главное, чтобы решение в контейнере было удобнее в использовании, чем классическое с инсталляцией.

Также большое удобство здесь в том, что тебе сразу может быть доступно несколько версий одного приложения, обычно несовместимых в одной "песочнице". Второе удобство, что контейнер после себя не оставляет столько мусора, как в случае с простой установкой приложения и от него очень просто избавиться.

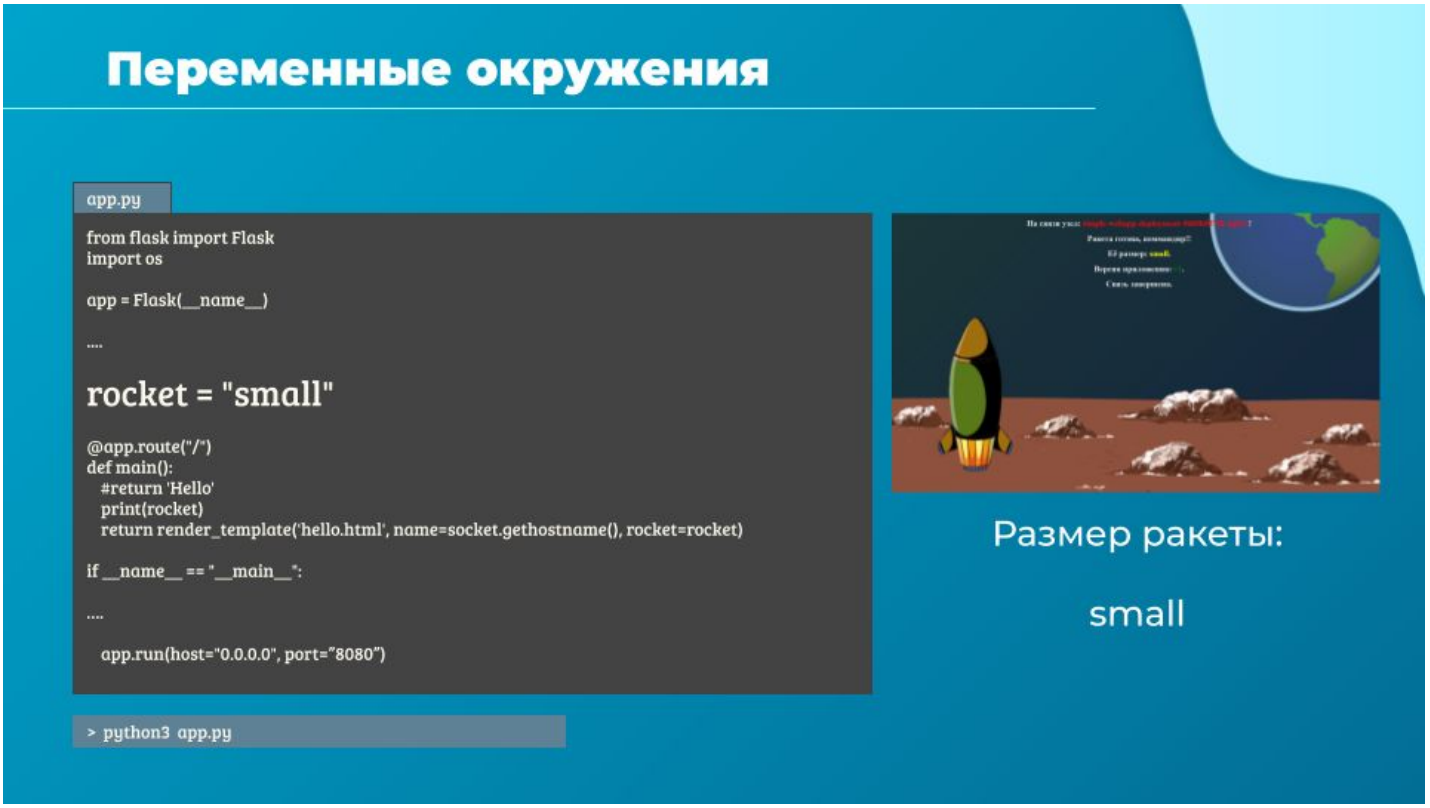
Это все в этой лекции, увидимся в следующей!

DEMO #5

Создание
docker образа

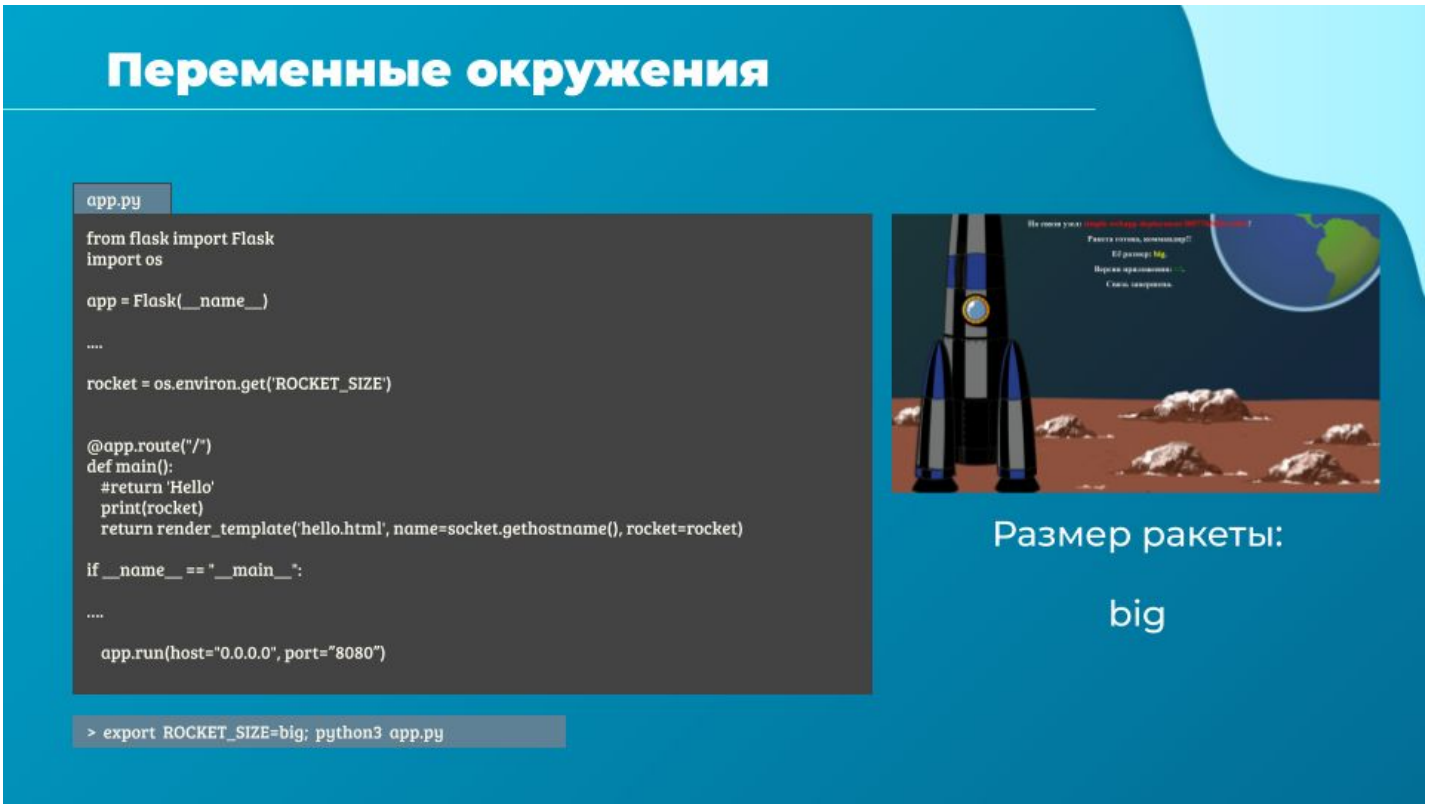
LAB #3
Docker Images

Привет, в этой лекции мы поговорим о переменных окружения.

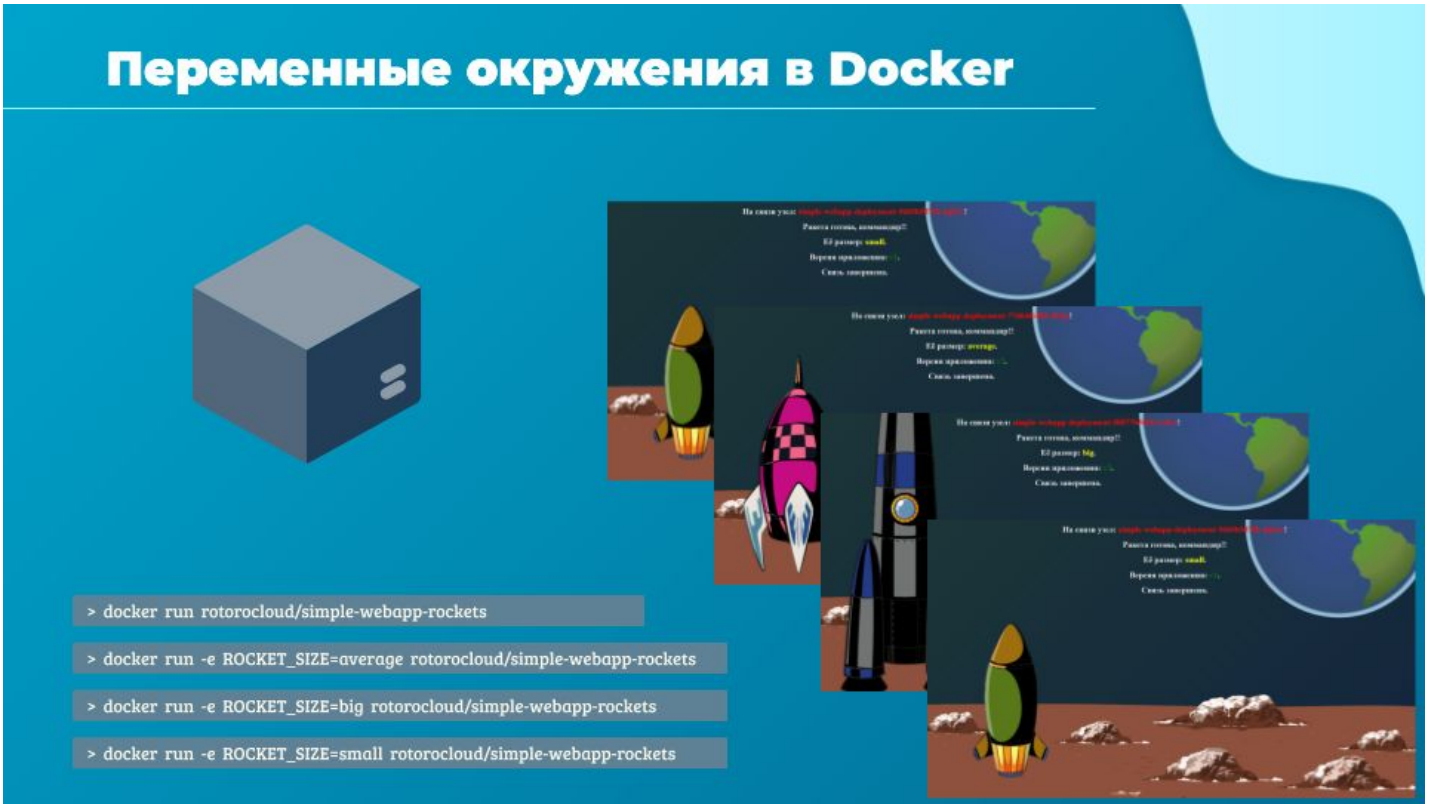


Начнем с нашего простого приложения, показывающее ракеты. Оно написано на python. Взгляни на этот участок кода. Он создает веб-сервер, который демонстрирует веб-страницу с ракетой. Если ты посмотришь внимательно, ты найдешь строку, которая определяет размер ракеты как small.

В данный момент все это работает прекрасно. Однако, если ты решишь изменить размер ракеты в будущем, то придется менять код приложения. Лучшая практика в таком случае вынести информацию из кода приложения в переменную окружения с именем ROCKET_SIZE.



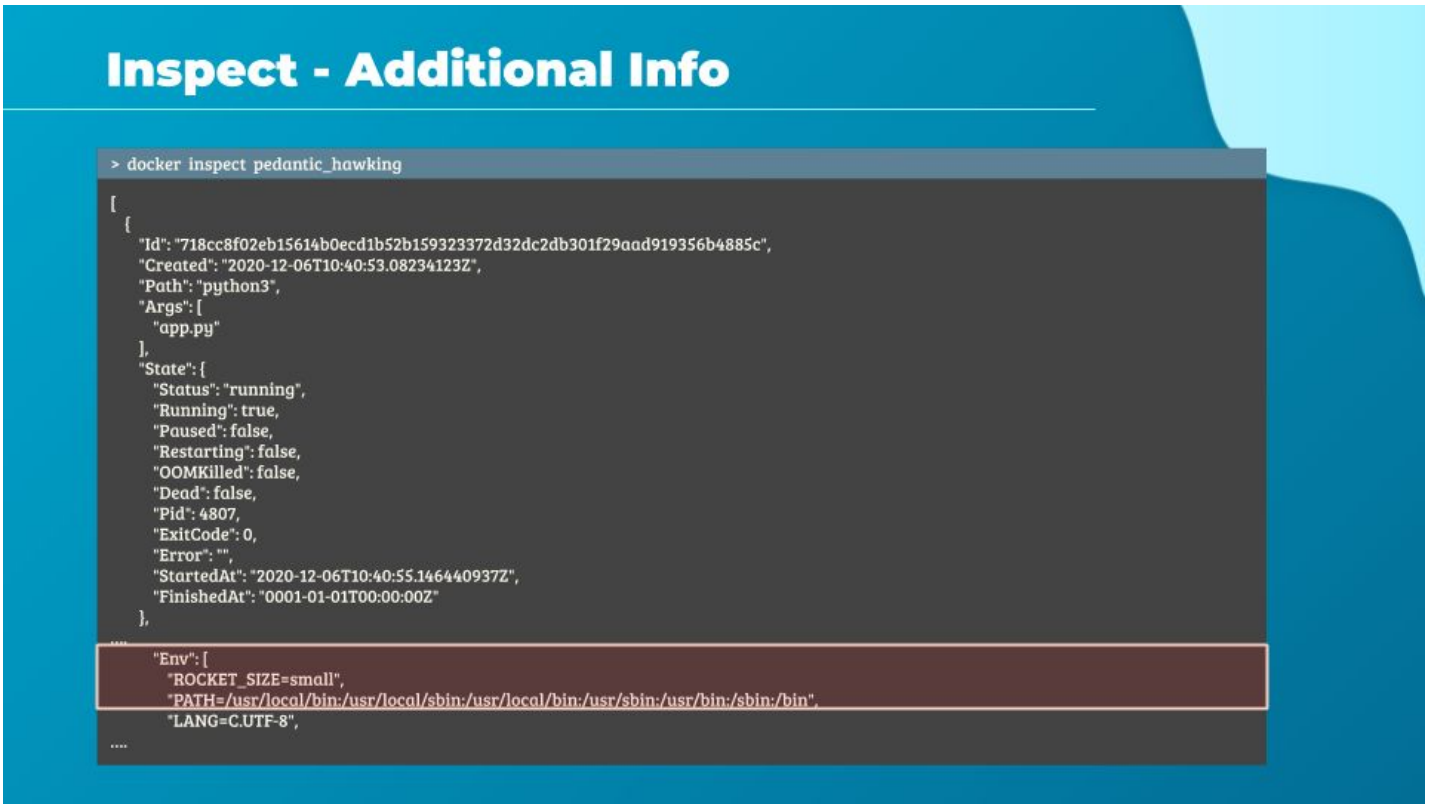
Внесем изменения в код. Теперь при запуске приложения установим переменную окружения ROCKET_SIZE в желаемое значение при помощи команды `export ROCKET_SIZE=big; python3 app.py` и запустим приложение. Как видишь такой подход может избавить нас от сборки отдельного билда приложения под каждый размер ракеты.



Пересоберем образ и запустим: `docker run rotorocloud/simple-webapp-rockets`

Приложение покажет нам ракету размера small. Совсем не то, что ожидали. Это потому, что мы назначили переменную окружения в нашей хостовой среде, но внутри контейнера ее не существует. Для того, чтобы установить эту переменную в контейнере используй параметр -e при запуске.
В нашем случае: `docker run -e ROCKET_SIZE=big rotorocloud/simple-webapp-rockets`

С этой опцией ты можешь запустить много одинаковых контейнеров, которые будут демонстрировать разные ракеты. Просто назначь разные переменные окружения этим контейнерам при старте.

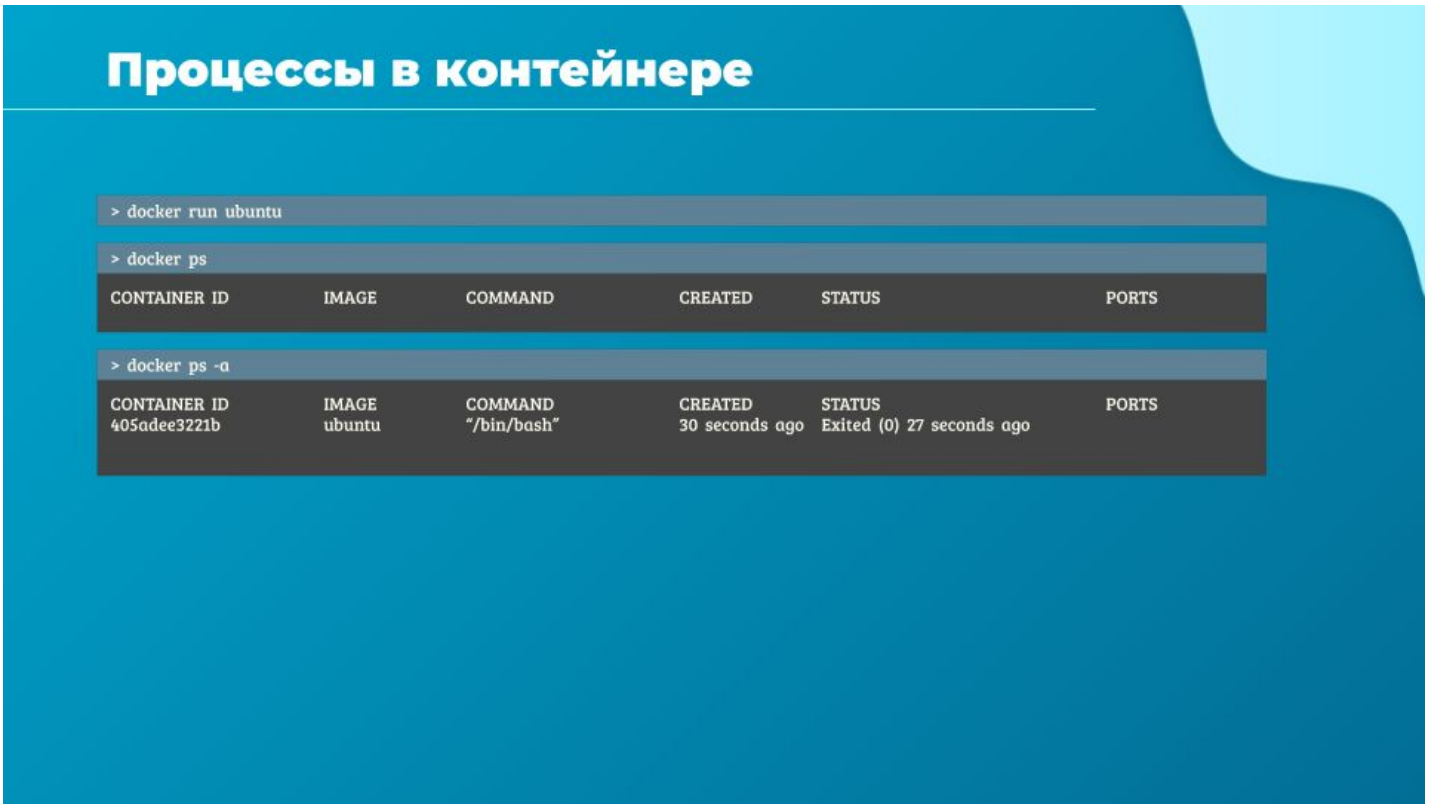


Как узнать, какие переменные существуют в уже запущенном контейнере и какие у них значения?

С помощью команды `docker inspect`. Эта команда показывает развернутую информацию о работающем контейнере, в том числе в секции config ты найдешь список всех переменных окружения этого контейнера.

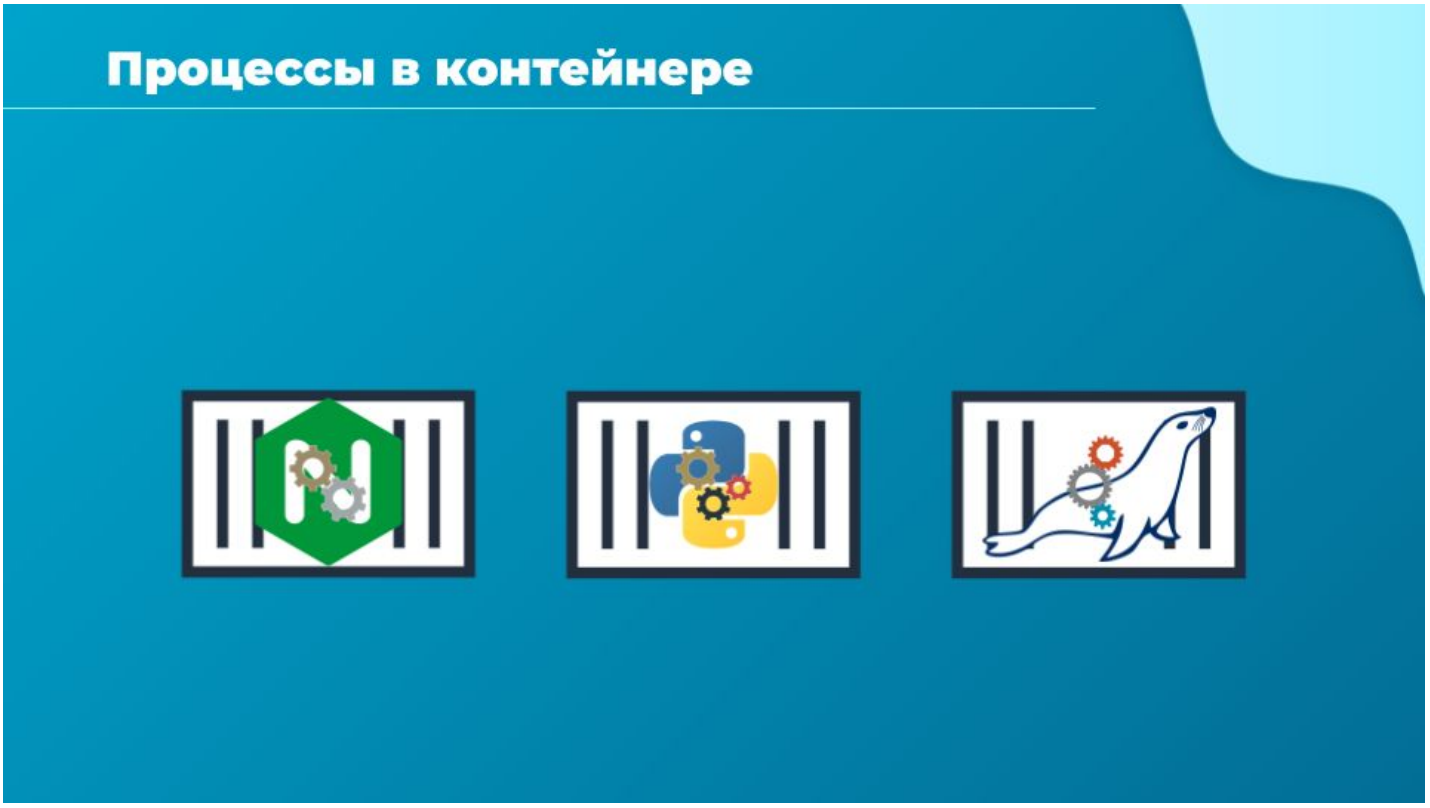
Это все в этой короткой лекции, переходим к практике.

Привет и добро пожаловать на лекцию, где мы поговорим о аргументах команд и точках входа в Docker.



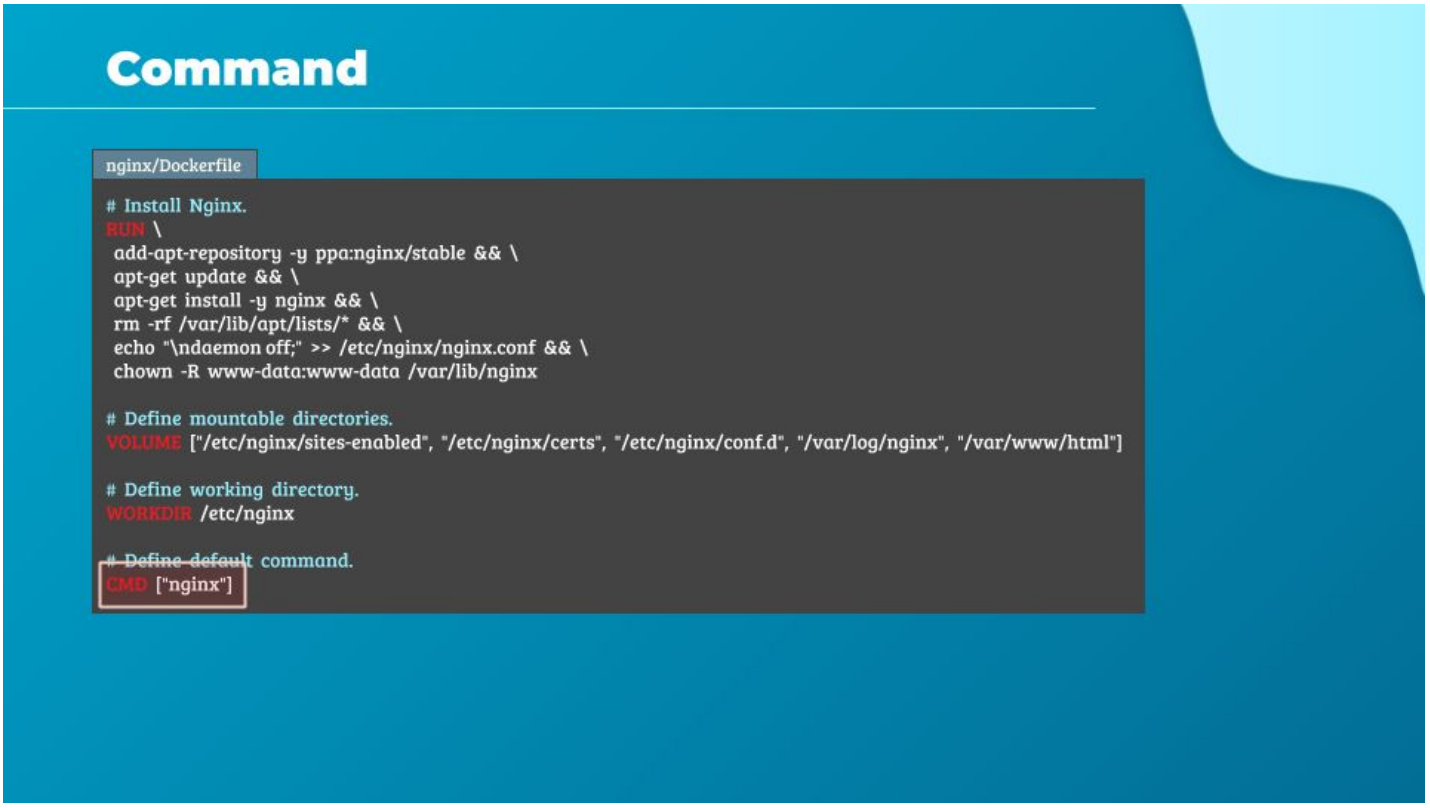
Начнем с простого сценария. Предположим, что мы запустили контейнер с образом ubuntu. Когда мы ввели команду `docker run ubuntu`, Docker запустил экземпляр образа ubuntu и немедленно сделал выход из контейнера.

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?

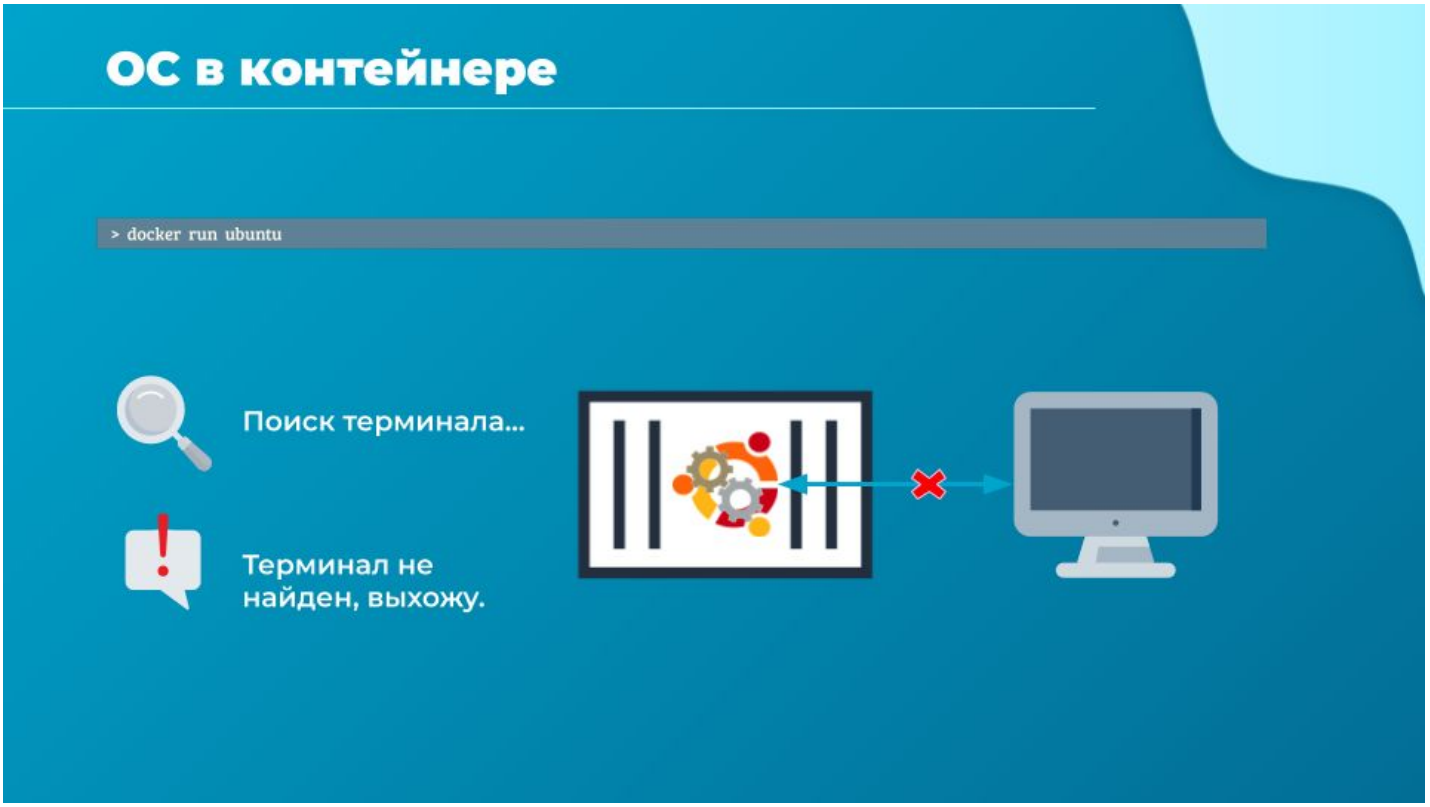


Как мы уже говорили, контейнеры не предназначены для размещения операционных систем.

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?

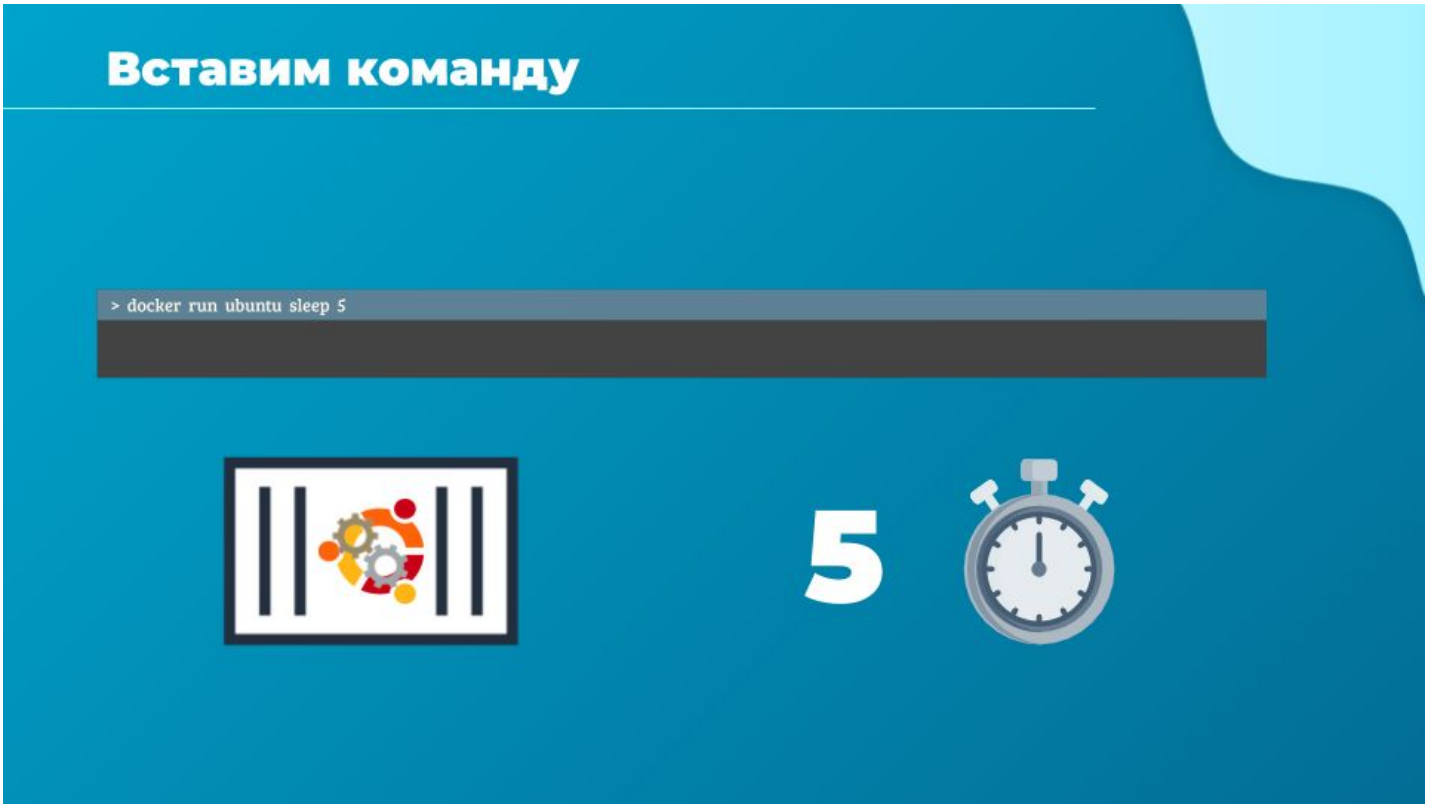


Если посмотреть на Dockerfile такого популярного приложения как nginx ты увидишь инструкцию CMD. Эта инструкция определяет команду, которая будет запускаться в контейнере при запуске его из образа. Это команда nginx. Для официального образа MySQL в Dockerfile будет команда mysqld.



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.

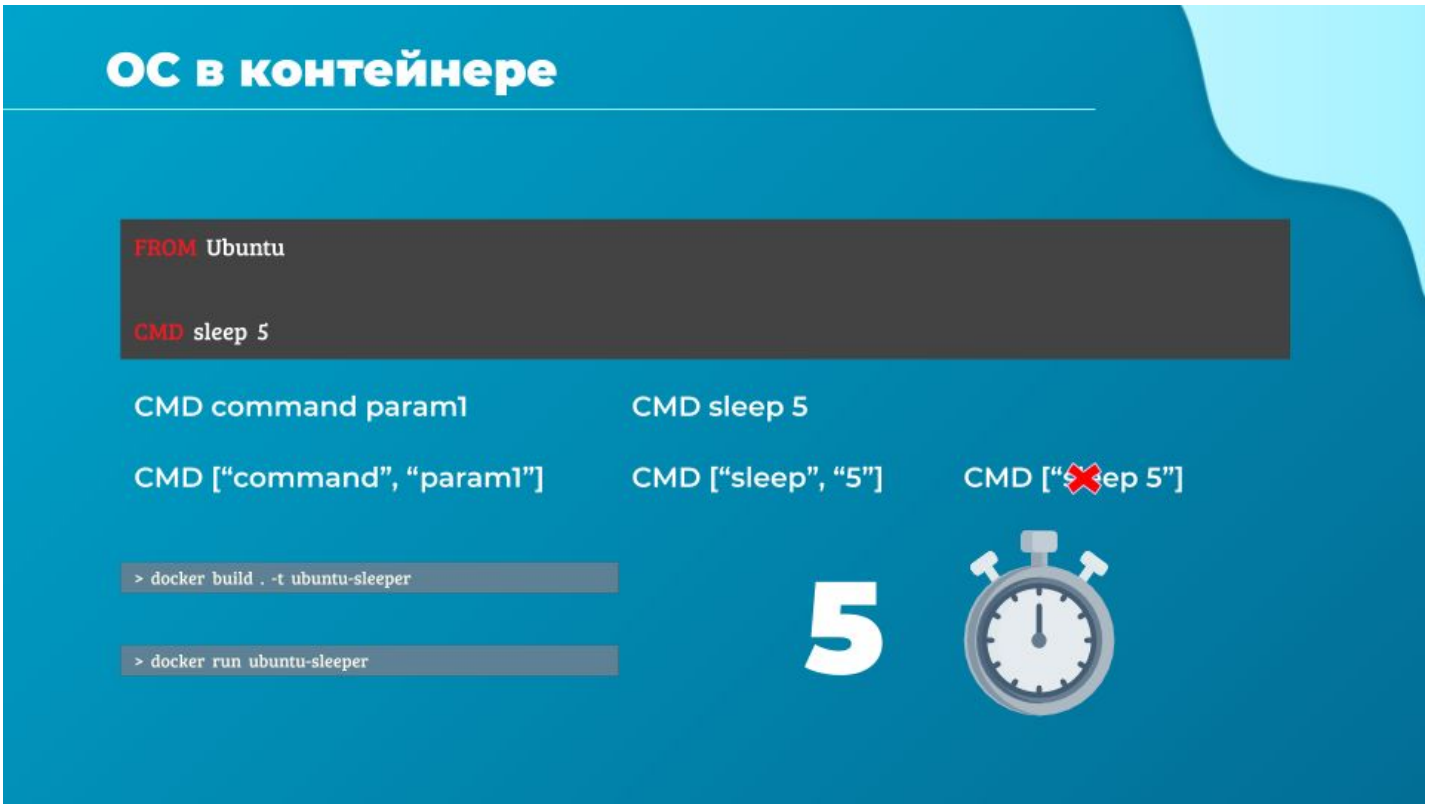
Таким образом, программа bash не находила терминал и завершала работу.
А поскольку процесс, который был запущен при создании контейнера, завершался, то и контейнер также останавливался. Как нам указать другую команду при старте контейнера?



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:

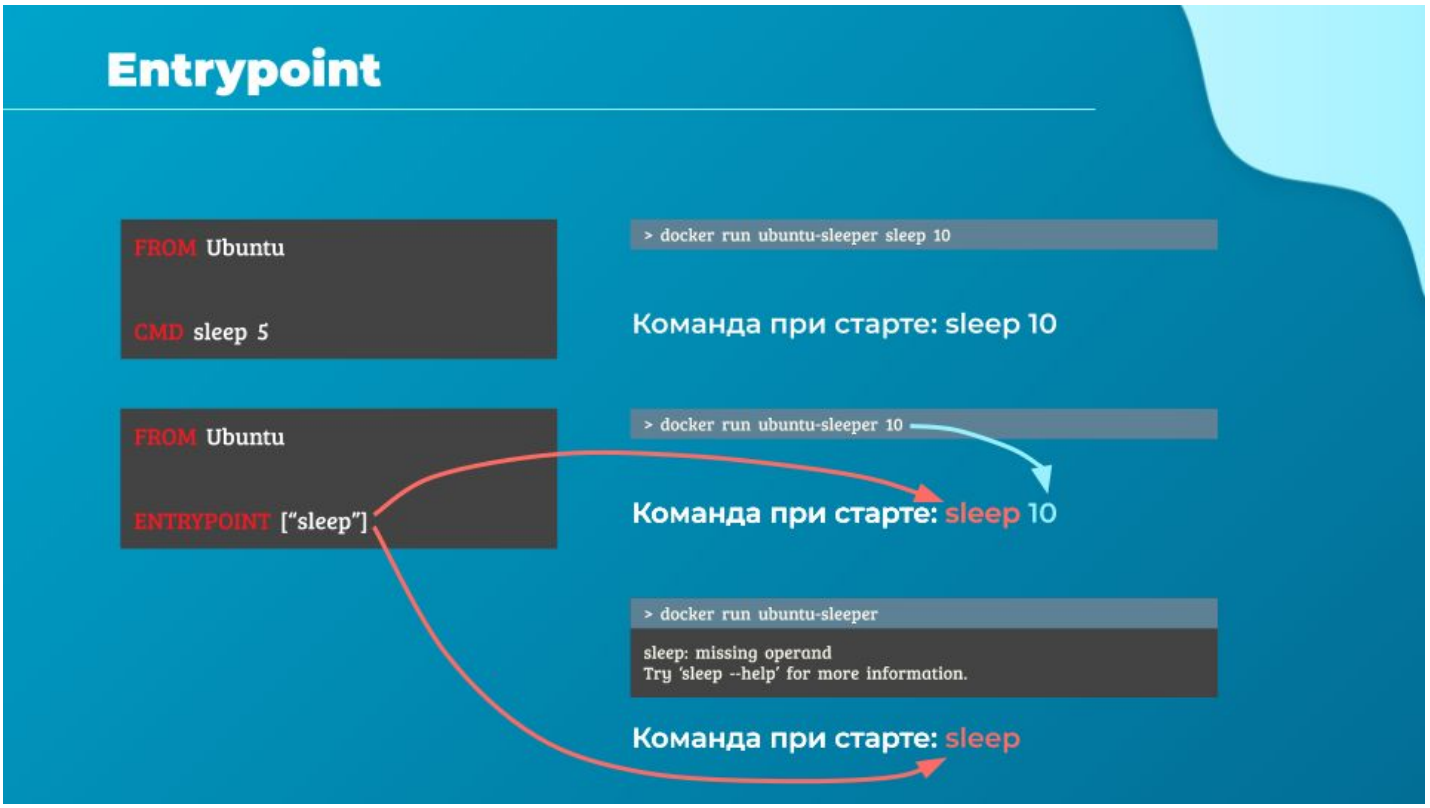
- **как ты ее пишешь в шелле**
- **в виде JSON-массива**

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`

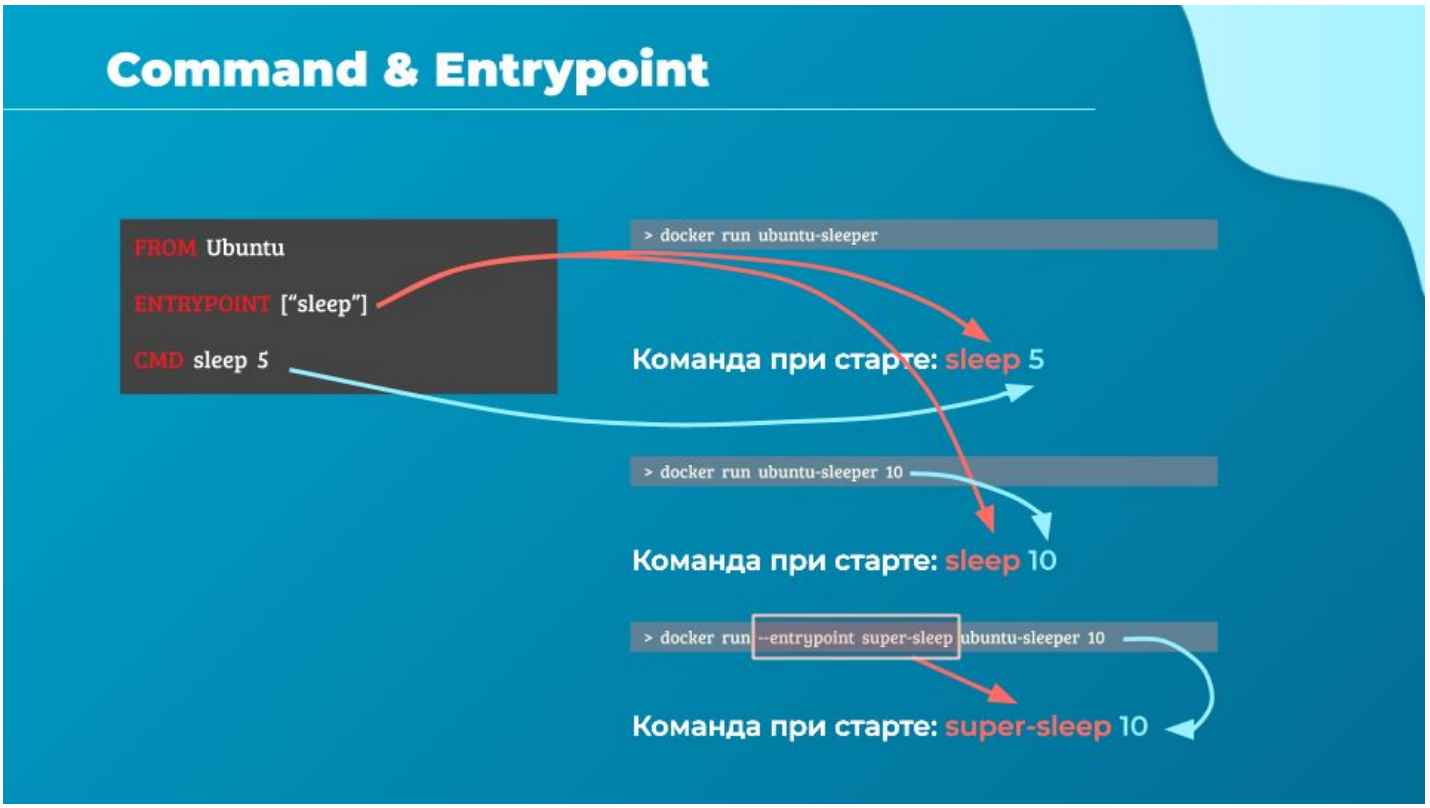
В этом случае команда sleep 10 выполнится при старте и успешно отработает положенные ей 10 секунд. Но это решение так себе. Само название ubuntu-sleeper подразумевает, что контейнер будет спать, не хотелось бы дополнительно указывать инструкцию вроде "проспи 10 секунд".

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.

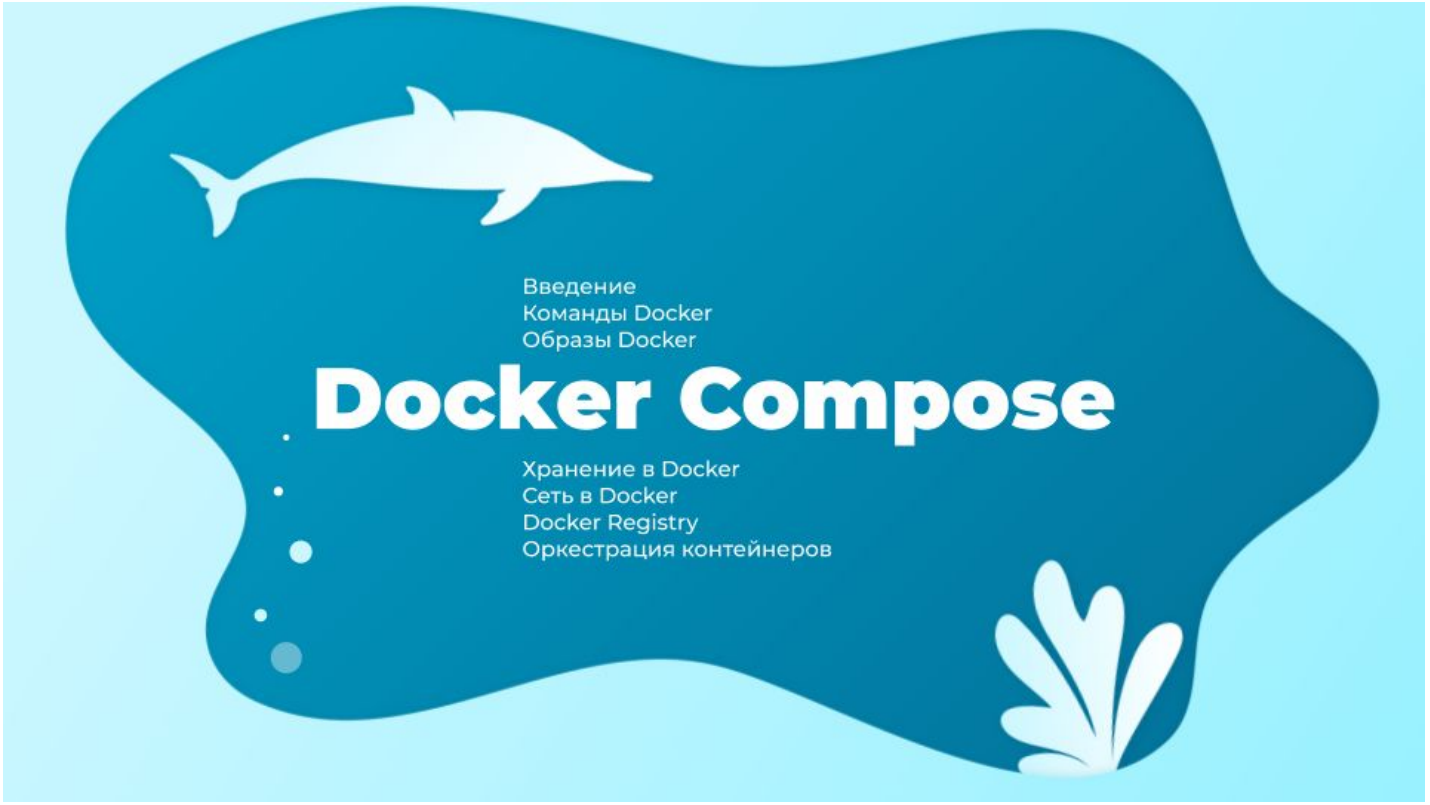
Запомни, чтобы это сработало всегда нужно указывать ENTRYPOINT и CMD в формате JSON.

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 праметр --entrypoint.

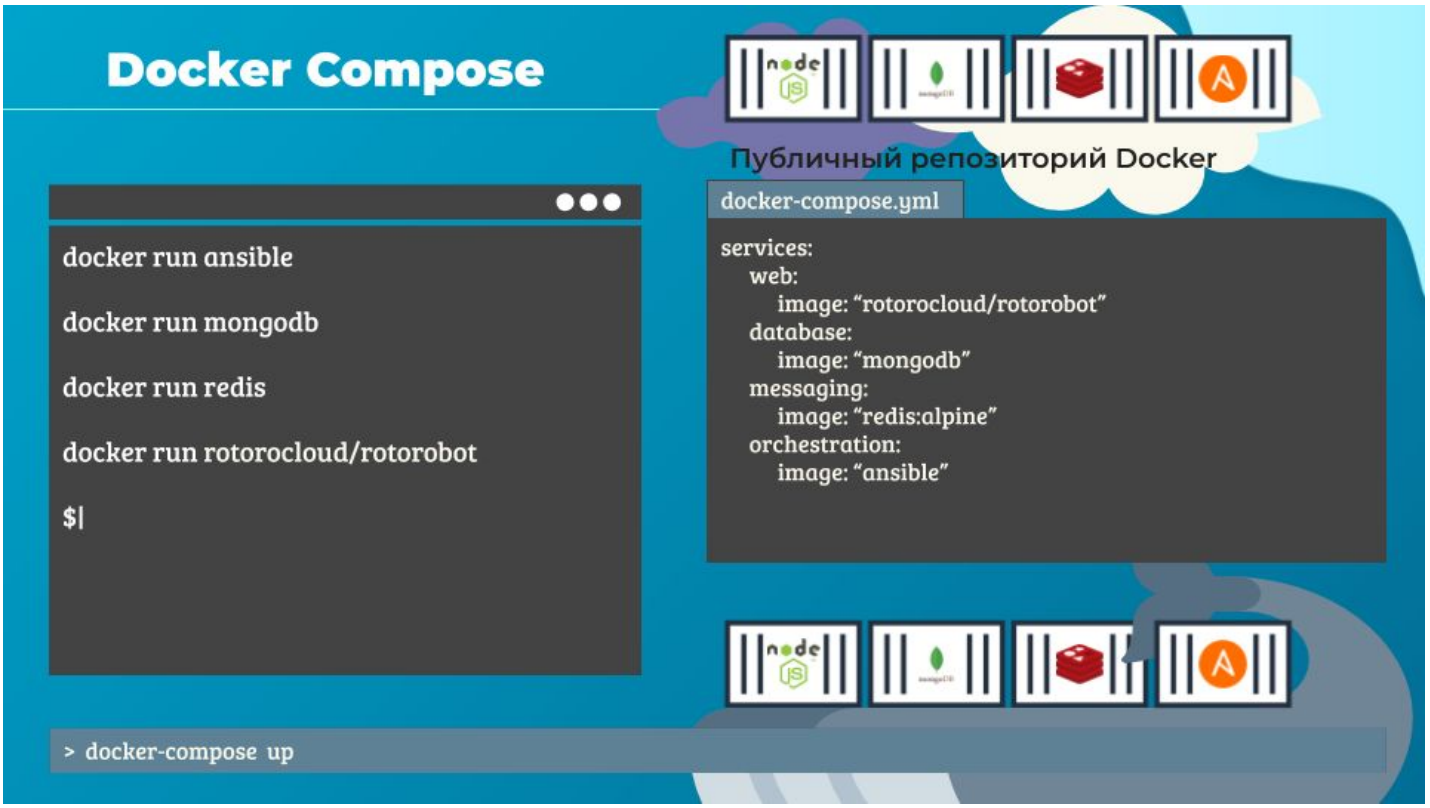
`docker run --entrypoint super-sleep ubuntu-sleeper 10`

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

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



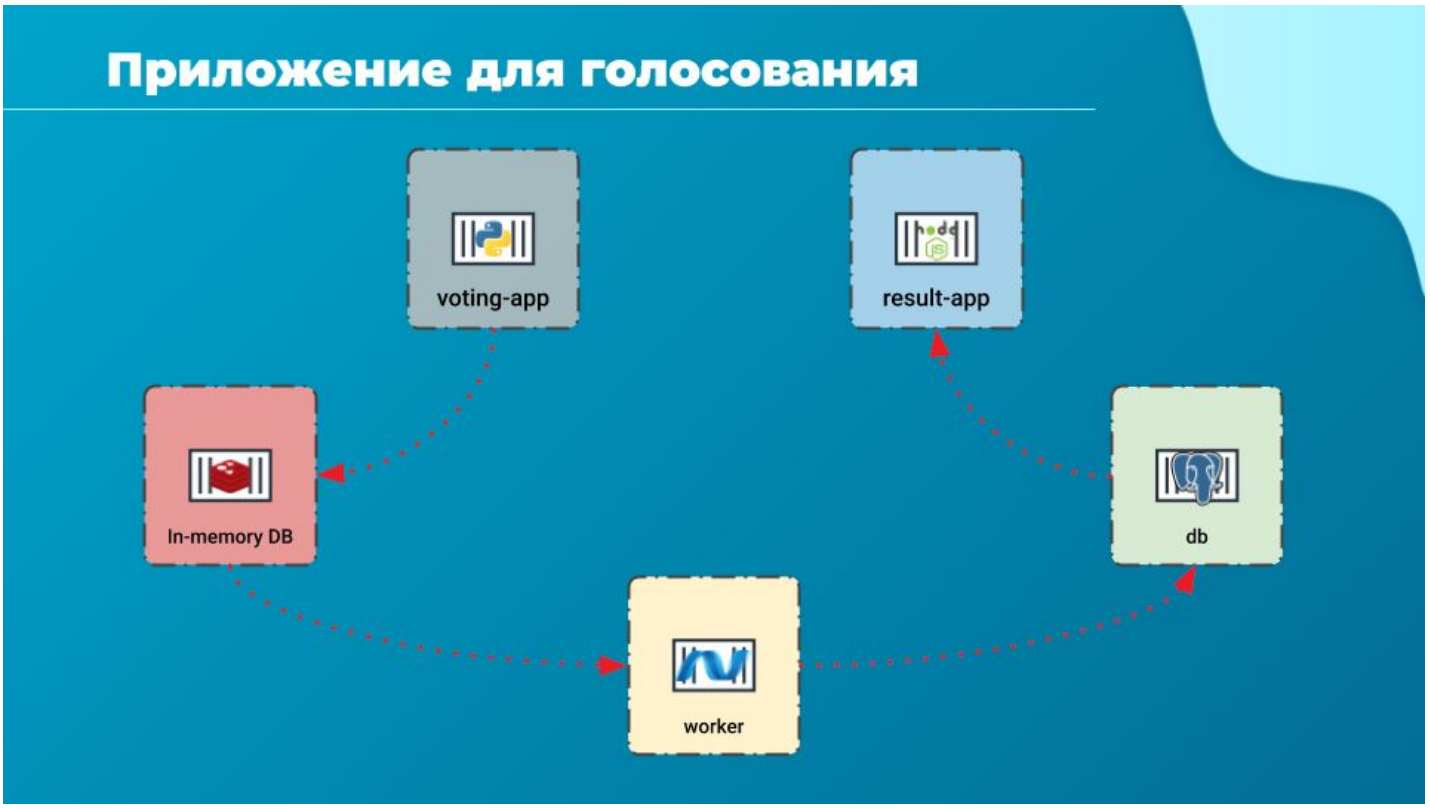
Привет и добро пожаловать на лекцию о Docker Compose. Дальше в лекции мы будем работать с конфигурациями в Yaml-файлах. Поэтому важно, чтобы ты чувствовал себя комфортно с yaml. Для новичков в yaml в секции приложений в конце курса есть материалы, которые помогут тебе быстро освоиться в этом формате представления данных.



Давай быстро резюмируем несколько вещей. Мы знаем, как построить образ, как отправить его на хранение и как запустить докер-контейнер с помощью команды `docker run`. Так, а если теперь нам нужно настроить сложное приложение с несколькими компонентами, как нам поступить?

Можно написать скрипт с несколькими командами `docker run`. Но лучший способ сделать это - использовать Docker Compose. Мы могли бы создать файл конфигурации в формате YAML под названием docker-compose.yml и положить туда все различные службы и прописать параметры, специфичные для их запуска. После этого, запустив команду `docker-compose up`, мы смогли бы одномоментно вызвать весь стек приложения.

Такой вариант проще реализовать, запускать и поддерживать, чем отдельные контейнеры с bash скриптами, поскольку все изменения всегда хранятся в одном файле конфигурации docker-compose.yaml. Но тут есть определенные ограничения, т.к. все это применимо только к запуску контейнеров на одном хосте Docker. Пока не будем беспокоится о моем yaml-файле.
Он вполне рабочий, но это слишком простое приложение, которое я собрал. Давай посмотрим на лучший пример.



У нас есть приложение от создателей докера, на котором хорошо отрабатывать особенности этой технологии. Итак, задача приложения собрать голоса пользователей, обработать их, сохранить и показать результат по требованию. Приложение должно быть шустрое, выдерживать большую пиковую нагрузку и быть устойчивым к сбоям.

Согласно этим требованиям разработчики создали приложение. Скорость будет достигаться за счет использования технологий кэширования, вроде redis, масштабирование достигается за счет контейнеризации, а устойчивость к сбоям за счет микросервисной архитектуры. Это тестовое приложение для голосования имеет под собой три логических яруса: фронтенд, бекенд и обработка данных.

Приложение для голосования - voting-app - разработано на Python, оно предоставляет веб-страницу с двумя опциями: Cats и Dogs. После того, как пользователь сделал выбор, voting-app передает эти данные для обработки в бэкэнд. Приложения voting-app и result-app несут в себе функции представления данных и взаимодействия с пользователем, другими словами фронтенд. Задача этих приложений устойчиво отработать под нагрузкой, дав нашим пользователям хороший опыт от общения с системой - user experience - UX.

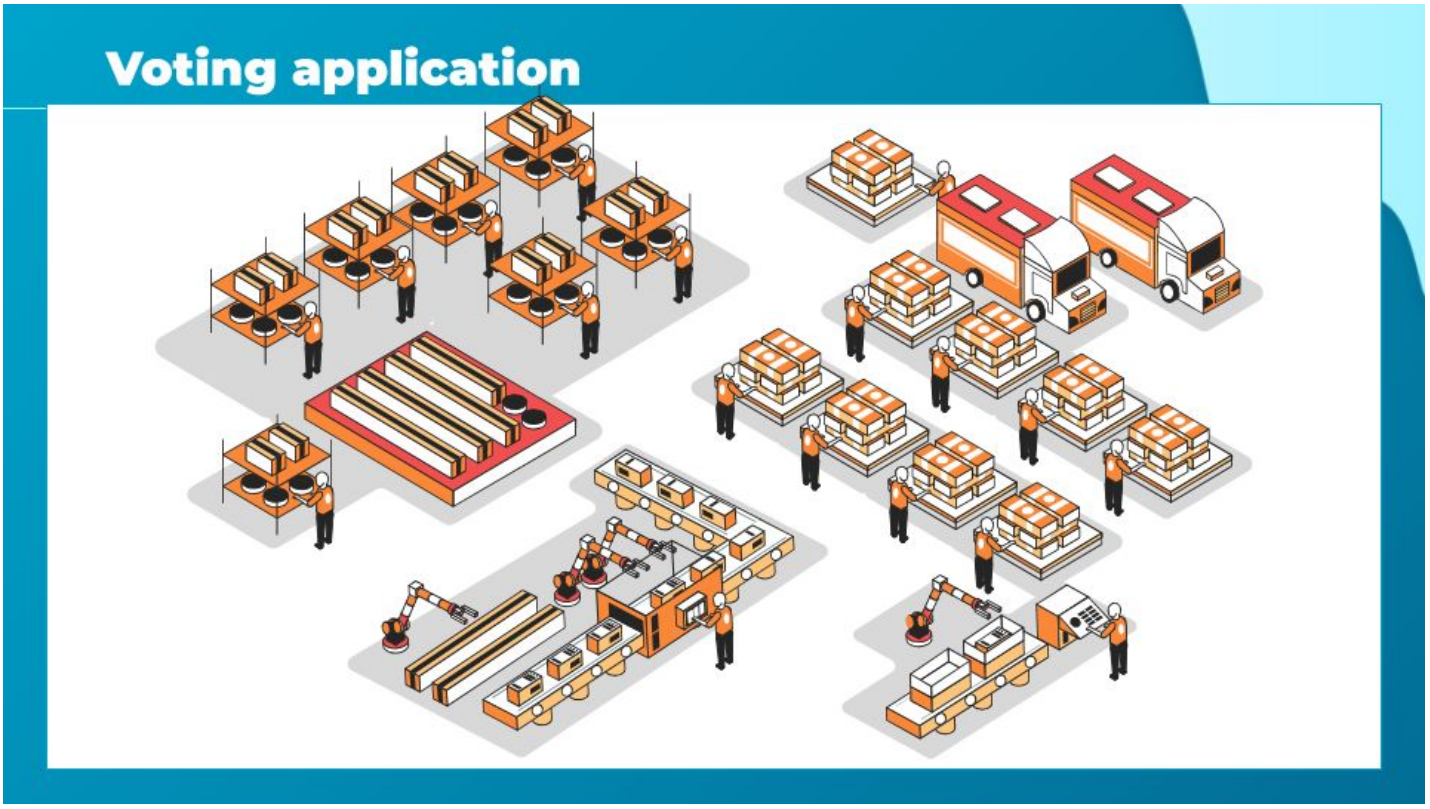
Если бы voting-app пыталось сразу записать данные в реляционную СУБД, вроде Postgres, мы бы быстро уперлись в ограничение пропускной способности БД. Пока база обрабатывала все запросы, наш портал example-vote.com был бы недоступным. Мы пойдем другим путем.

Для обеспечения быстрой работы используем временное хранилище данных redis. Благодаря тому, что его база находится в оперативной памяти, voting-app очень быстро выполнит запрос и вернется к своему привычному делу - показывать веб-страницу пользователю.

Итак, на первом этапе все данные стекаются в in-memory cache, реализованный в redis. Далее в игру вступает третий компонент нашей системы - worker.

Worker это .net приложение, его функция периодически забирать информацию из кэша, производить дополнительные действия по ее проверке и подготовке к хранению, а далее отправить этот "пакет" данных в Postgres на постоянное хранение.

Реляционная база данных хранит свою информацию в виде таблиц, из-за этого скорость записи у нее не так высока, как скорость чтения. Result-app демонстрирует результаты голосования, вычитывая их из БД с вполне с приличной скоростью, а если этого вдруг станет не хватать, то всегда можно будет подключить redis и к этому процессу.



Это все по архитектуре нашего simple voting application. Я бы сравнил это с фабрикой.

Кладовщики (python) имеют одну задачу - максимально быстро положить на склад (redis) все прибывающие материалы (запросы от юзеров). Воркер - это автоматизированная линия, которая постоянно производит свои товары из материалов (задания для БД из запросов). База данных Postgres - склад готовой упакованной продукции, которую удобно будет использовать потребителям. А служба доставки (NodeJS) - доставляет готовые результаты пользователям.

Как видишь решение наших разработчиков сочетает в себе несколько инструментов, платформ и технологий. На его примере мы посмотрим, как легко можно настроить стек разнородных компонентов в Docker.

На пару минут отложим в сторону все автоматизации вроде services от docker-compose и stacks от Docker swarm и посмотрим, как мы можем собрать этот стек приложений на едином окружении Docker, используя сначала обычные команды запуска.

Мы считаем, что все наши образы уже созданы и доступны в Docker-репозитории.

Давай начнем с уровня данных.
Для запуска используем команду `docker run`.
Начнем с redis.
Команда `docker run -d --name=redis redis`

Как мы знаем, параметр -d говорит Docker, что контейнер нужно запустить в фоновом режиме.
Также мы указываем имя контейнера redis. Имя контейнера важно, а почему, ты скоро поймешь.

Далее разворачиваем Postgres DB
Команда `docker run -d --name=db postgres:9.4`

-d значит тоже самое, что и в предыдущий раз - запуск в бекграунде. Имя для контейнера базы db.

Дальше очередь фронтенда. Запустим сборщик голосов командой
`docker run -d --name=vote -p 5000:80 voting-app`

Это приложение веб-сервер, слушает на http порту (#80), мы опубликуем его в хост-системе под номером 5000. Таким образом трафик полученный на порт 5000 нашей хост-системы будет передан на порт 80 контейнера vote, этим мы сделаем его доступным из браузера.

Приложение, показывающие результат result-app разворачиваем командой
`docker run -d --name=result -p 5001:80 result-app`

Здесь мы аналогично пробрасываем порты для доступности результатов из браузера.

В конце запустим worker
`docker run -d --name=worker worker`

Ок, вроде все идет неплохо. Контейнеры запустились. Но проблема в том, что обращение на 5000 порт из браузера - дает нам 500 ошибку.

Дело в том что мы запустили 5 контейнеров, а не микросервисное приложение. Приложением оно станет, когда сервисы начнут общаться друг с другом, т.е. у контейнеров будет возможность коммуницировать. Мы должны связать контейнеры и сказать каждой части приложения с кем им взаимодействовать. Применительно к voting-app, нужно явно сказать с каким экземпляром redis ему работать, т.к. redis может быть несколько. Тоже самое с worker, он должен понимать, какой Postgres ему нужен.

Как нам это сделать? С помощью links. Link это опция командной строки, позволяющая связать два контейнера вместе.

Например веб-сервис voting-app зависит от кэширующего сервиса redis. Когда веб-сервер запустится этот код будет искать redis на хосте "redis". Но контейнер vote не сможет разрешить это имя и понять, где находится его redis. Для осведомления voting-app о том, где находится redis мы используем опцию link при запуске контейнера-голосования.

Пишется так --link (имя контейнера где живет сервис):(имя по которому будет общаться с этим контейнером запускаемое приложение).

В нашем случае `docker run -d --name=vote -p 5000:80 --link redis:redis voting-app`.

Вот поэтому ранее я говорил, что имена контейнеров важны! Чтобы мы смогли связать требуемые компоненты.

Что на самом деле делает --link? Просто вносит статическую запись в /etc/hosts контейнера vote с привязкой имени redis к текущему внутреннему IP контейнера redis.

Таким же способом мы связываем контейнер result с контейнером db.

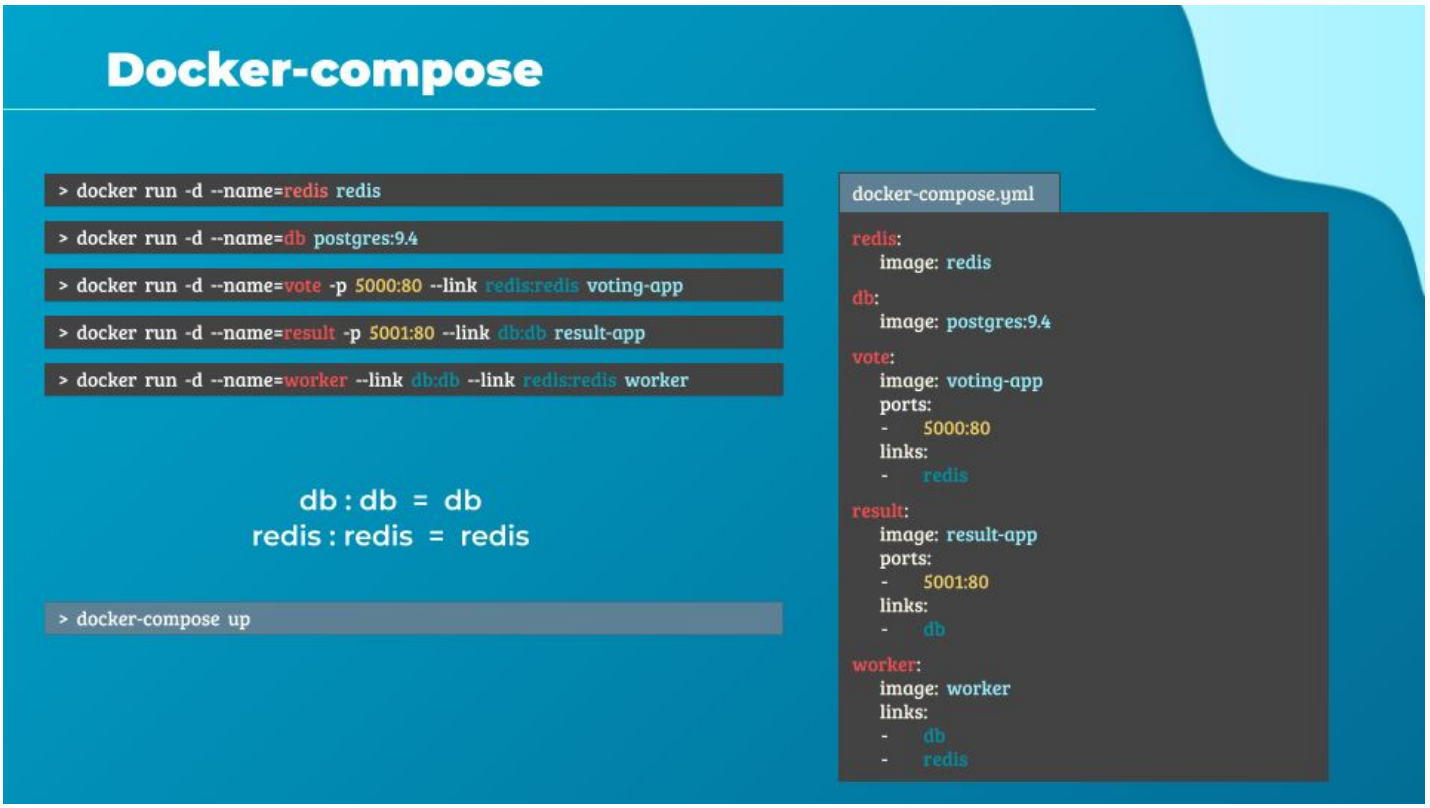
`docker run -d --name=result -p 5001:80 --link db:db result-app`.

Как видишь, в коде NodeJS-приложения название базы просто захардкожено. Это не лучшая практика, но для учебных целей можно.

Контейнер worker немного отличается от предыдущих, т.к. ему потребуется общаться сразу с двумя контейнерами: брать данные в redis и отправлять их в db.

Просто добавим два линка:
`docker run -d --name=worker --link db:db --link redis:redis worker`

На самом деле link устаревшая конструкция. На смену ей пришли более продвинутые концепции networks, но для понимания основ конструкция link нам подходит, в любом случае лишним не будет знать об этой возможности.



Итак, у нас есть проверенные и готовые к использованию команды Docker. Из него будет легко создать файлы для docker-compose. Начнем с dictionary имен контейнеров. Мы будем создавать его используя те же названия, что и в командах docker run.

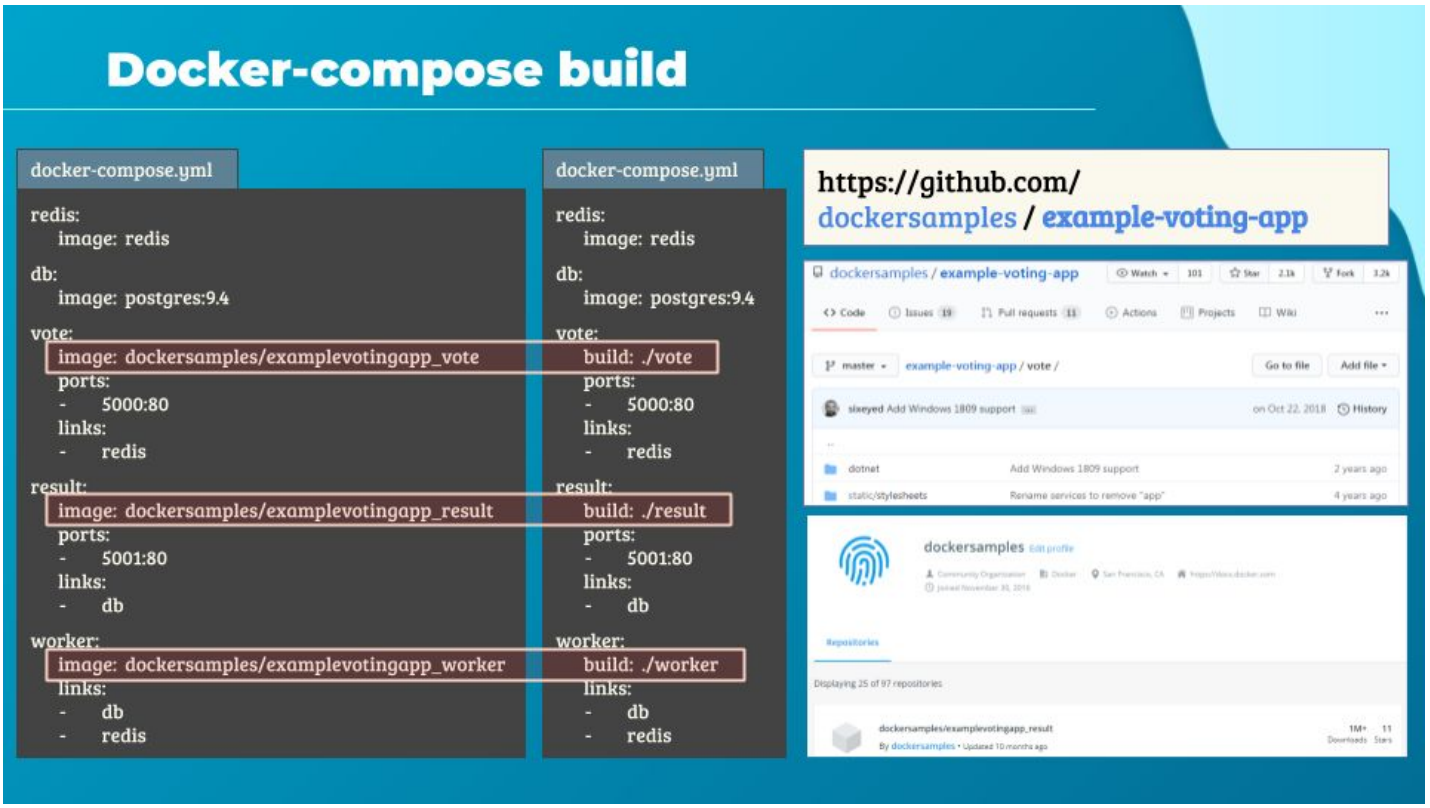
Теперь мы берем все имена микросервисов и создаем ключ для каждого из них. Затем под каждым элементом мы указываем, какой использовать образ. Ключ - это будущий контейнер, а значение - имя образа, которое будет в нем использоваться.

Также мы использовали дополнительные параметры в командах, а именно в двух командах публиковали порты. Для этих контейнеров мы укажем проброс соответствующих портов. Создаем свойство под названием ports и перечисляем все порты, которые мы хотим опубликовать под ним.

Наконец у нас остались links. Те контейнеры, которые нужно связать друг с другом следует явно обозначить. За это отвечает свойство links. Это свойство является массивом, таким же как и ports.

Заметь, если имя службы, к которой идет обращение в коде совпадает с именем контейнера, например redis или db, то можно указать это имя один раз без двоеточия. Это будет тоже самое, как если бы указали redis:redis или db:db.

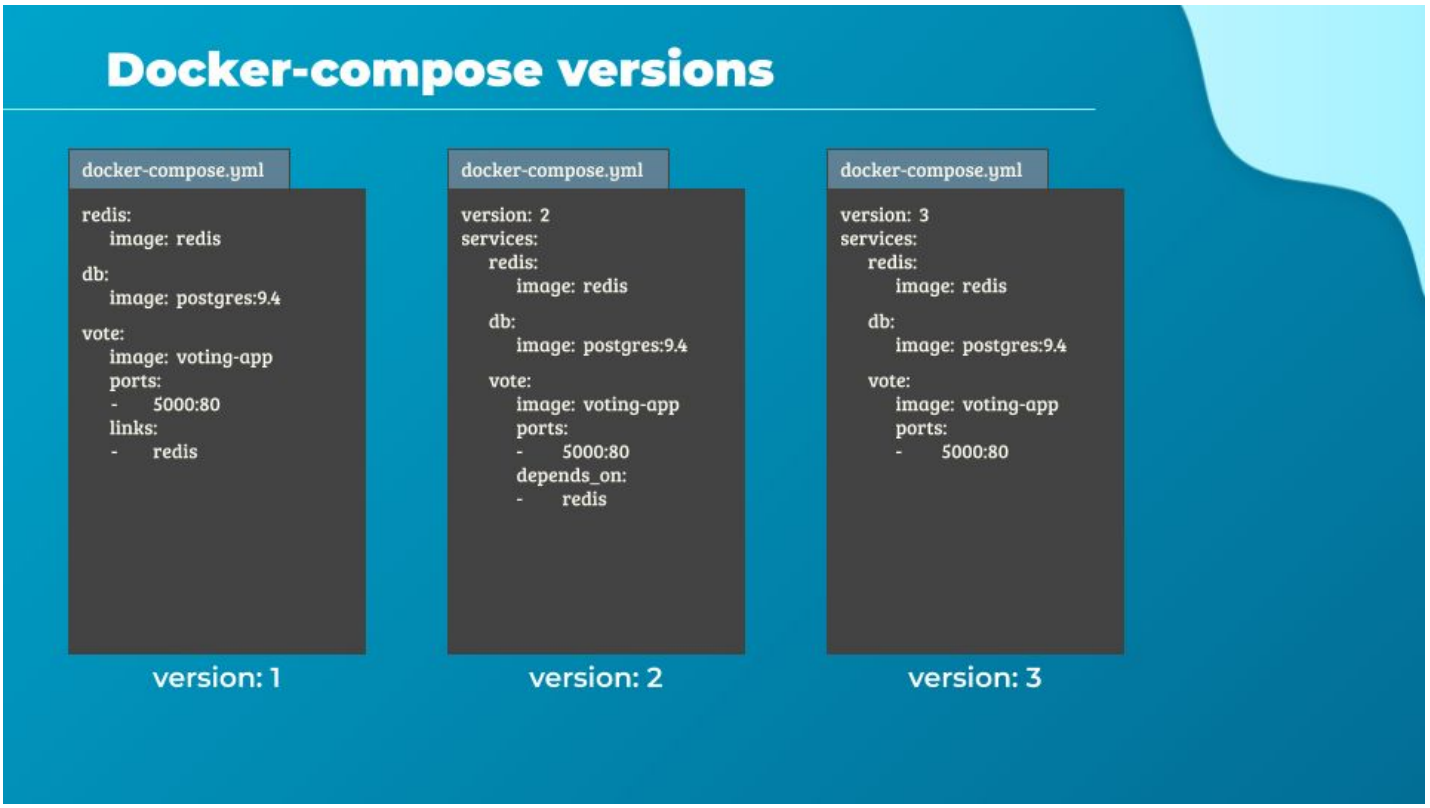
Мы закончили с нашим файлом конфигурации, и теперь поднять стек на самом деле очень легко. Делаем это командой `docker-compose up`.



В созданном нами файле docker-compose.yml мы исходили из того, что все образы уже собраны. Из пяти различных компонентов два известных нам образа redis и postgres уже доступны на DockerHub.

Это официальные образы от Redis и Postgres, но остальные три - наше собственное приложение. Совсем необязательно, чтобы они были уже созданы и записаны в какой-то докер-репозиторий. Если мы хотим проинструктировать Docker инициировать процесс сборки образа вместо скачивания уже созданного из репозитория, мы можем поменять параметр image на build, указав местоположение каталога, который содержит код приложения и Dockerfile с инструкциями по сборке образа.

В этом примере у приложения для голосования весь код лежит в папке с именем vote, также она содержит Dockerfile. Итак, когда мы запустим команду `docker-compose up`, она сначала соберет образ, даст ему временное имя, а затем будут использовать этот образы для запуска контейнера с использованием указанных в файле параметров. Аналогичным образом будут созданы две оставшиеся службы: образы будут собраны из соответствующих папок и запущены их контейнеры.



Теперь мы рассмотрим разные версии файла docker-compose. Это важно, потому что мы постоянно сталкиваемся с разными docker-compose файлами в разных форматах. Возникает вопрос, почему в разных местах они выглядят по-разному?

Docker-compose с течением времени развивался и сейчас поддерживает гораздо больше возможностей, чем вначале. Вот, например, урезанная версия файла docker-compose, которую мы использовали ранее.

Фактически это ранняя версия docker-compose, которая называется версия 1. У ней есть ряд ограничений, например, невозможно развернуть контейнеры в другой сети, отличной от мостовой сети по умолчанию. Также нет возможности указать, что запуск одного контейнера зависит от запуска другого. Например, наш контейнер базы данных должен появиться первым, и только после этого должно быть запущено приложение для голосования. Мы не можем указать это в версии 1 файла docker-compose.

Этот функционал стал поддерживаться только с версией 2 и выше. С версией 2 также немного изменился и формат файла. Мы больше не указываем информацию о стеке напрямую, как раньше. Все это стало инкапсулировано в раздел services. Поэтому нам нужно создать свойство с именем services в корневом уровне yml файла, а затем переместить все описания контейнеров в этот раздел служб. Чтобы мы по-прежнему могли использовать эту же команду `docker-compose up` для вызова своего стека приложений, нужно явно сообщить docker-compose, какую версию файла мы здесь используем. Мы свободно можем использовать версию 1 или версию 2 в зависимости от своих потребностей. Итак, чтобы все это работало, в первой строчке мы указываем, как docker-compose разбирать форматирование. Используется параметр version: и указание версии.

В этом случае цифра 2 после двоеточия сразу скажет docker-compose объединить контейнеры в единую мостовую сеть, а затем использовать ссылки для связи между контейнерами, как мы это делали явно в первой версии нашего файла.

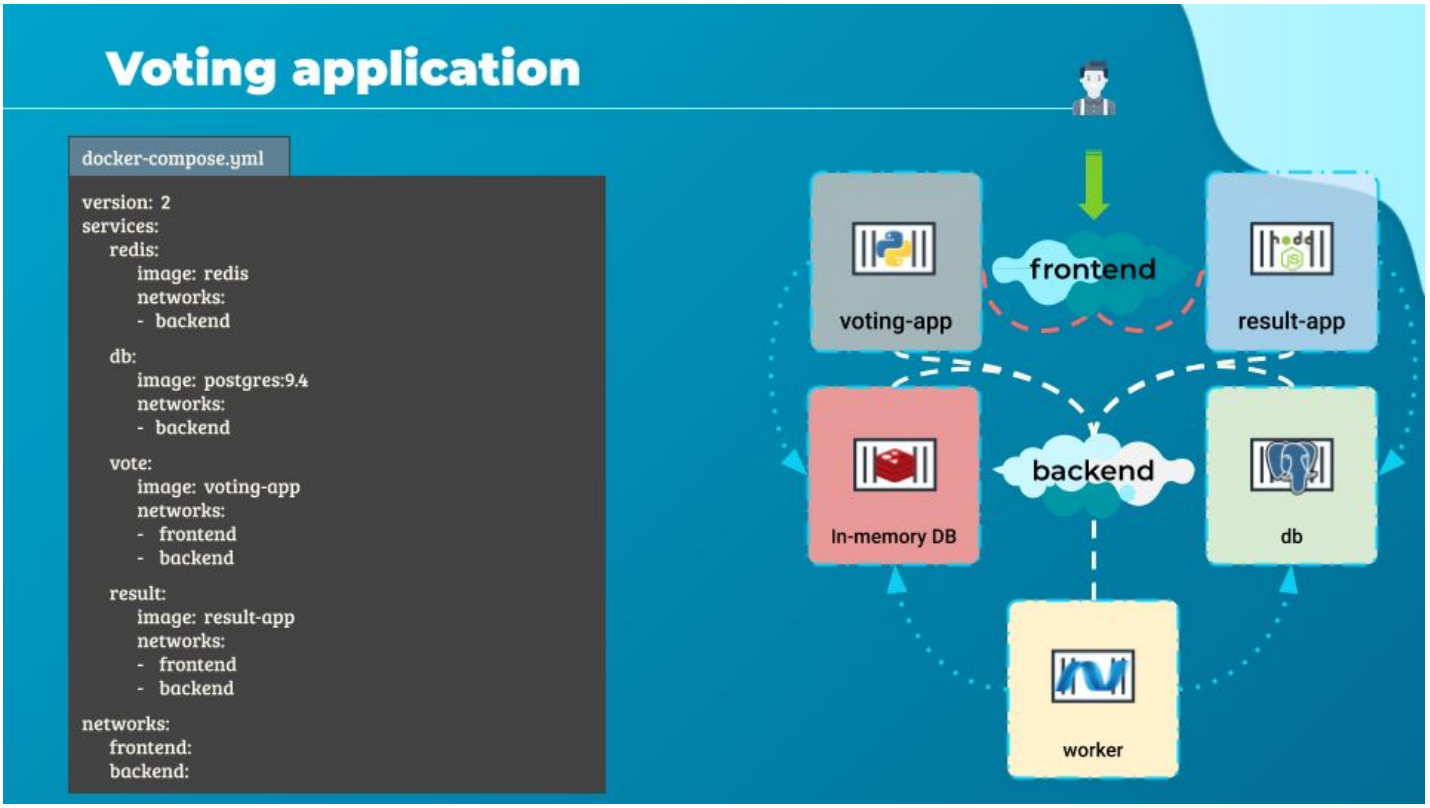
Это еще одно отличие первой версии от второй, что контейнеры сразу могут связываться друг с другом, используя имена, заданные под параметром services. Т.е. во второй версии docker-compose

параметр links больше не нужен. Мы можем просто избавиться от всех ссылок, которые были созданы, при конвертации docker-compose файла из версии 1 в версию 2.

И, наконец, версия 2 также вводит функцию зависимостей для запуска, если мы хотим указать порядок создания контейнеров. Например, веб-приложение для голосования зависит от службы redis. Таким образом, нам необходимо убедиться, что контейнер redis запускается первым, и только после этого запускается веб-приложение для голосования. Мы можем добавить свойство depends_on, в приложение для голосования и указать, что оно зависит от redis.

Теперь версия 3. На сегодняшний день последняя версия 3.8. Ее структура сходна с версией 2. Вверху описание версии, далее идет раздел services, в который мы также помещаете все свои контейнеры, как и в версии 2. Существенное отличие в 3 версии - ее поддержка оркестрации с помощью Docker swarm и специфичных возможностей для совместной работы нескольких хостов.

В ней некоторые параметры были удалены, и многие добавлены. Увидеть подробности ты можешь в разделе документации на сайте docker, поискав по слову compose. В следующих лекциях мы взглянем на версию 3 более подробно, когда обсудим Docker Stacks.



Сейчас давай поговорим о сетях в docker-compose касательно нашего приложения. До сих пор мы просто развертывали все контейнеры в мостовой сети по умолчанию. Хорошей практикой является разделить трафик из разных источников. Давай немного изменим архитектуру нашего развертывания.

Например, мы решили отделить трафик, генерируемый пользователями, от внутреннего трафика приложений. Для этого мы создаем внешнюю сеть, предназначенную для трафика от пользователей, и внутреннюю сеть, предназначенную для общения между приложениями.

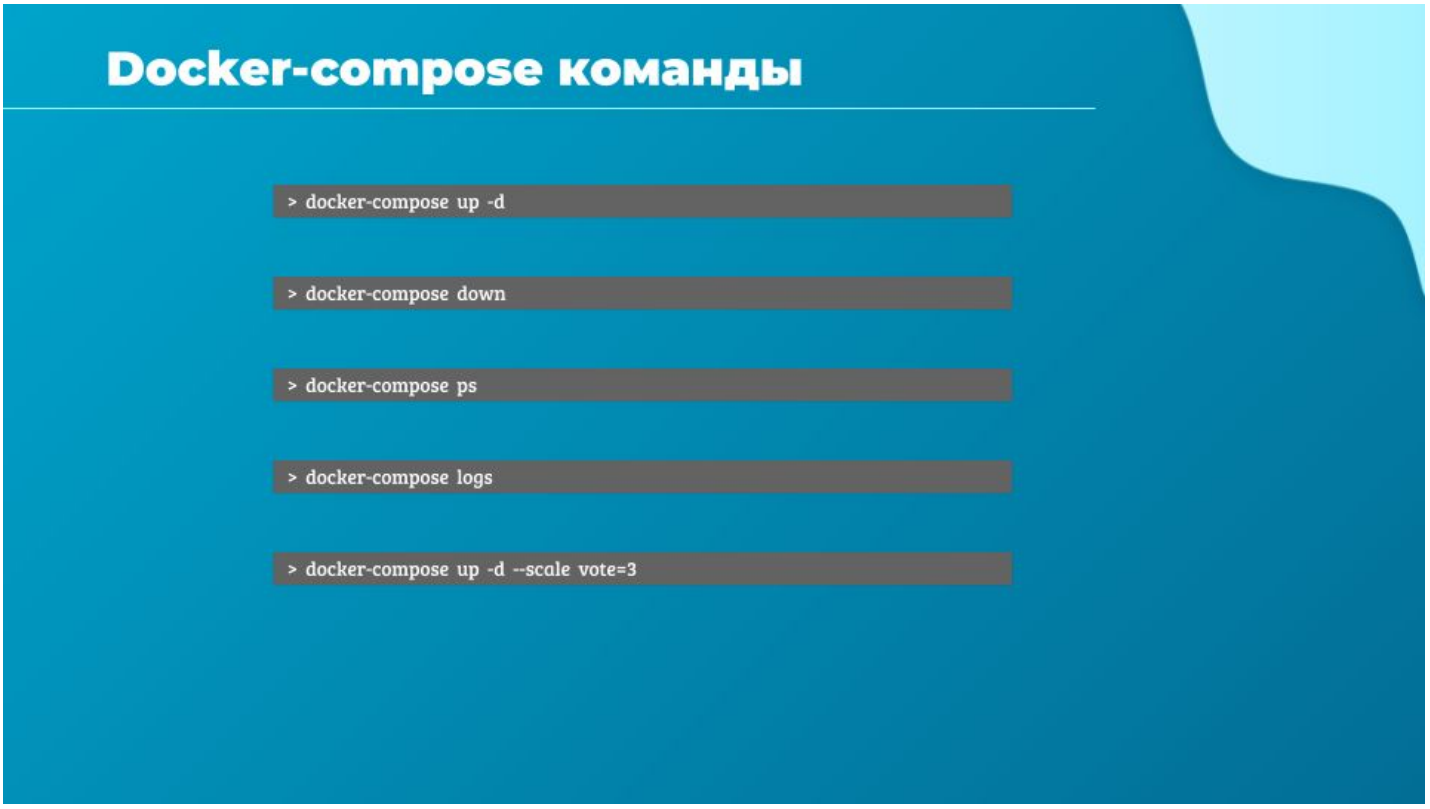
Фронтенд и бэкенд. Сетью фронтенда связываем приложения, ориентированные на пользователя, это приложение для голосования и для результатов. Бэкендом свяжем внутренние компоненты: redis, postgres и worker. Все это мы можем занести в наш файл docker-compose.yml

Обрати внимание, что я вырезал раздел портов для простоты, они все еще там, но просто не показаны на экране.

Первое, что нам нужно сделать, если мы будем использовать networks, - это определить какие именно сети мы собираемся использовать в этом развертывании. У нас есть две: внешняя и внутренняя сети. Поэтому создадим новое свойство, называемое networks, на уровне корня yaml документа, а не в services. В этом свойстве мы разместим нашу карту сетей. Мы планируем использовать эти сети во всех объявленных службах. Таким образом нам придется их явно объявить при помощи свойства networks уже внутри имени службы, к каким сетям должен быть подключен этот контейнер.

В данный момент мы указали, что result подключен к сети frontend и backend. потому что это приложение берет данные из внутренней сети и отдает их во внешнюю. Так же и для vote. К каким сетям будут подключены службы redis и dB?

Это только backend, т.к. эти приложения не взаимодействуют с пользователем. В отличие от интерфейсных приложений, таких как приложение для голосования и результатов, которые необходимо подключить как к фронтенду, так и к бэкенду. Мы также должны добавить раздел networks для воркера, чтобы его контейнер был добавлен во внутреннюю сеть, здесь просто не хватило на него места.



Теперь, когда мы разобрали эти файлы docker-compose переходим к практике, чтобы потренироваться в составлении подобных yaml файлов. Но перед этим я хочу показать тебе несколько основных команд Docker Compose.

`docker-compose up -d`
Разворачивает стек из файла docker-compose.yml текущей директории.

`docker-compose down`
Останавливает стек и удаляет все контейнеры и сети.

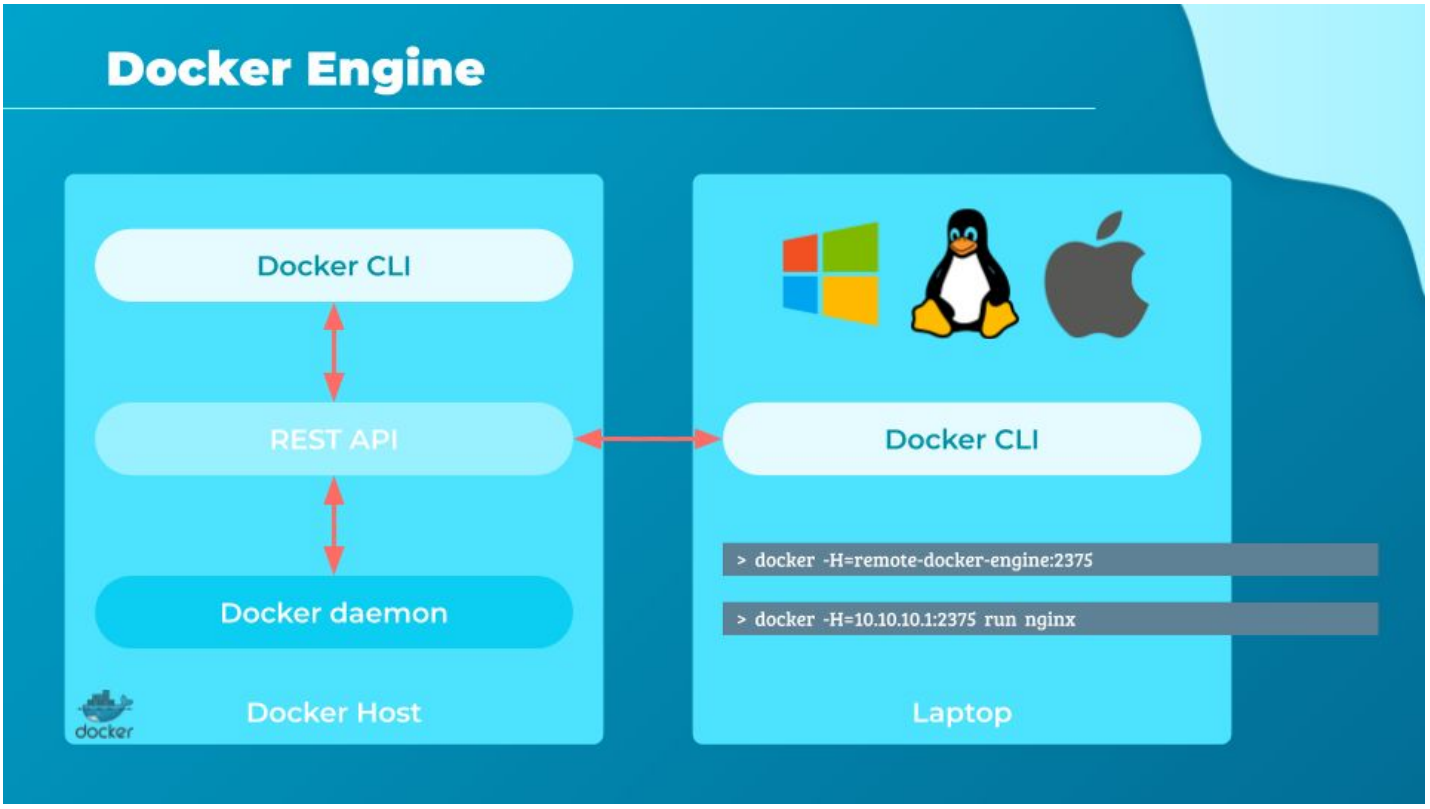
`docker-compose ps`
Покажет контейнеры, за которые Docker Compose несет ответственность.

`docker-compose logs`
Общий пул логов поднятого стека Docker Compose.

`docker-compose up --scale vote=3`
Управление количеством реплик определенной службы.

А в этой лекции все, я жду тебя на следующей.

Добро пожаловать на эту лекцию о движке Docker. В этой лекции мы более подробно рассмотрим архитектуру Docker, на чем основана магия изоляции контейнеров и какие технологии у него "под капотом".



Движок Docker (Docker engine), как мы узнали ранее, работает на хосте с установленным Docker. Это такой комбайн, который умеет многие вещи:

- **собирать образы**
- **запускать контейнеры**
- **транспортировать образы**
- **оркестрировать контейнеры (в случае swarm)**
- **обслуживать отдельные слои**

- **делать траблешутинг**

С одной стороны это удобно, с другой стороны универсальность сыграла с Docker в минус. Об этом мы поговорим еще в разделе окрестрации. В данный момент речь об архитектуре.

Итак, когда мы устанавливаем Docker на хост Linux, фактически мы устанавливаем три разных компонента:

- **Демон Docker**
- **REST API-сервер**
- **приложение командной строки докера.**

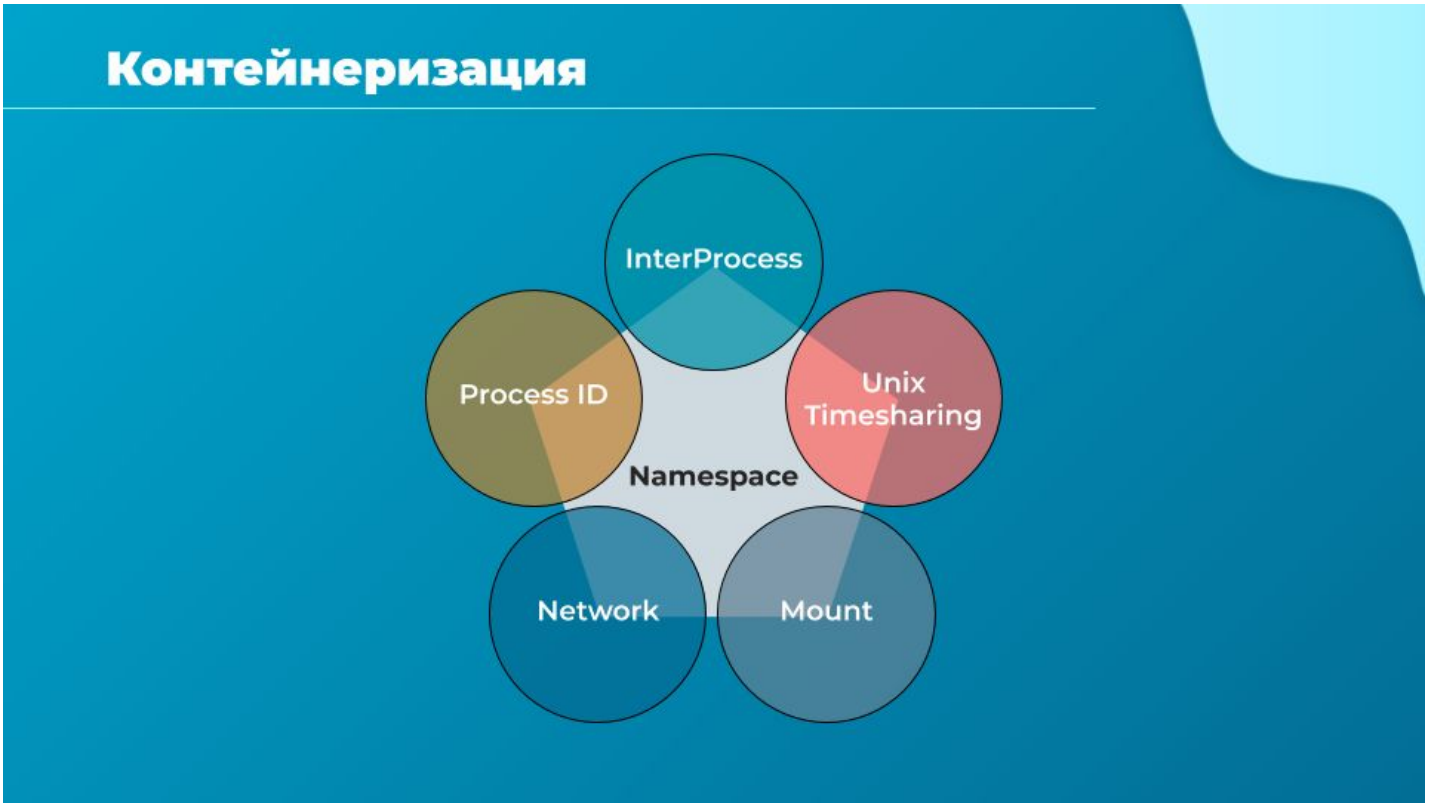
Docker-daemon - это фоновый процесс, который управляет объектами Docker, такими как образы, контейнеры, тома и сети.

API-сервер Docker поддерживает API, который программы могут использовать для общения с демоном и предоставления инструкций. Ты можешь создать свои собственные инструменты, используя этот REST API. Кстати, на этом принципе построены наши лабораторные работы.

Docker cli - это утилита командной строки, которую мы использовали до сих пор для выполнения таких действий, как запуск и остановка контейнеров, уничтожение образов и т. д. Она использует вызовы REST API для взаимодействия с демоном Docker.

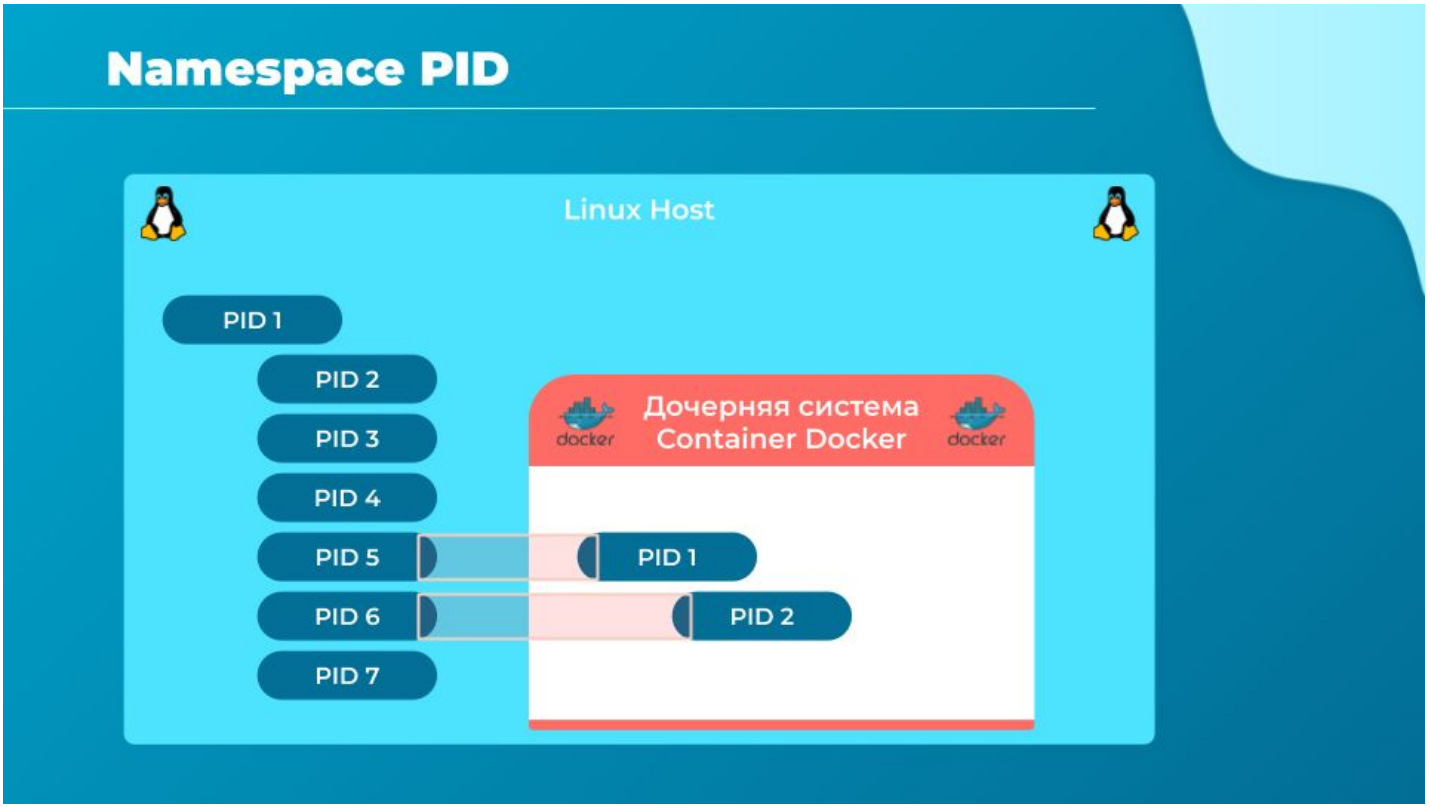
Следует отметить, что утилита командной строки Docker не обязательно должна находиться на одном хосте. Этим докер-хостом может быть другая система, например, я могу управлять серверным удаленным движком Docker, работающим на сервере в ДЦ со своего ноутбука. Чтобы использовать такой вызов к команде docker нужно добавить параметр -H, далее указать адрес удаленного хоста и его порт. Также можно проиниализировать переменную окружения DOCKER_HOST для постоянных вызовов к определенному хосту.

Как показано здесь, для запуска контейнера на основе Nginx на удаленном хосте Docker запустим команду: `docker -H 10.10.10.1:2375 run nginx`



Теперь давай попробуем разобраться, как именно в Docker контейнеризирует приложения чуть ниже уровнем, что называется "под капотом".

В случае Linux Docker использует пространство имен (namespaces linux) для изоляции идентификаторов процессов рабочего пространства, подключения к сети, системы разделения времени Unix. Процессы контейнера выполняются в их собственном пространстве имен, тем самым обеспечивая изоляцию с другими процессами. Если тебе не очень ясно, это нормально. Когда я впервые об этом услышал, я ничего не понял, поскольку раньше не встречался с namespaces.



Давай посмотрим на пару примеров. Сначала как работает изоляция процессов - изоляция в PID namespace.

Всякий раз, когда система Linux загружается, она запускается только с одного процесса. У него process identificator = 1. Так устроен Linux. Это корневой процесс, запускающий все остальные процессы, они как бы отпочковываются от него. После загрузки у нас бежит несколько процессов одновременно. Выполним команду `ps`, чтобы вывести список всех запущенных процессов. Процессы уникальны, и два процесса не могут иметь одинаковый идентификатор процесса.

Теперь, мы создали контейнер, он будет похож на дочернюю систему, размещенную в текущей системе. Дочерняя система должна думать, что она является независимой системой, что она сама по себе. Таким образом у нее должен быть свой собственный набор процессов, порожденных от корневого процесса №1, а также прочие атрибуты независимой системы.

Да, как видишь у нее присутствует процесс с ID 1, но мы знаем, что между контейнерами и базовым хостом нет жесткой изоляции. Таким образом, процессы, выполняемые внутри контейнера, на самом деле являются процессами, выполняемыми на базовом хосте. Но у двух процессов не может быть одного и того же PID. Для хоста все процессы должны быть уникальны. Но внутри контейнера все иначе. Он по другому нумерует процессы, он же считает себя независимым.

Нам нужно как-то урегулировать этот конфликт. Здесь вступает в игру PID namespace. Эта штука позволяет добавить к процессу помимо его настоящего ID еще и другой. Также она будет понимать, каким службам давать первичный PID, а кому "фейковый". С каждым процессом может быть связано несколько таких ID процесса.

В нашем случае, когда процессы запускаются в контейнере, это фактически просто еще один набор процессов в базовой системе Linux, и они получат следующий доступный идентификатор процесса в хостовой системе. Т.е. пять и шесть. Однако, они также получают другой идентификатор процесса начиная с PID 1 в пространстве имен контейнера, которое видно только внутри контейнера. Наш контейнер имеет собственное корневое дерево процессов и поэтому является независимой системой. Точнее считает себя таковой. Итак, как это связано с реальной системой?

Допустим я запустил на своем хосте с адресом 10.0.0.13 другую ОС в контейнере с адресом 172.17.0.2.

Теперь я зайду с двух терминалов на докерхост и в контейнер.
И там и там я выполню команду `ps aux`.

В контейнере я увижу два процесса:
- **с PID 1, это bash в котором я нахожусь**
- **с PID 10 это только что выполненная команда `ps aux`**

На хосте после целой простыни процессов, в конце я увижу интересные для меня:
- **с PID 2945 это bash, который работает в данный момент в контейнере**
- **с PID 3119 это `ps aux`, который только отработал в контейнере**
- **с PID 3129 это `ps aux`, который только отработал в на хосте**

Как видишь, контейнер думает, что корневой процесс это bash и у него PID = 1, но его реальный PID 2945.

Я бы сравнил жизнь контейнера с параллельной вселенной, в которой уютно живется процессам контейнера, они не видят остального мира, но докер-хост видит его процессы полностью.

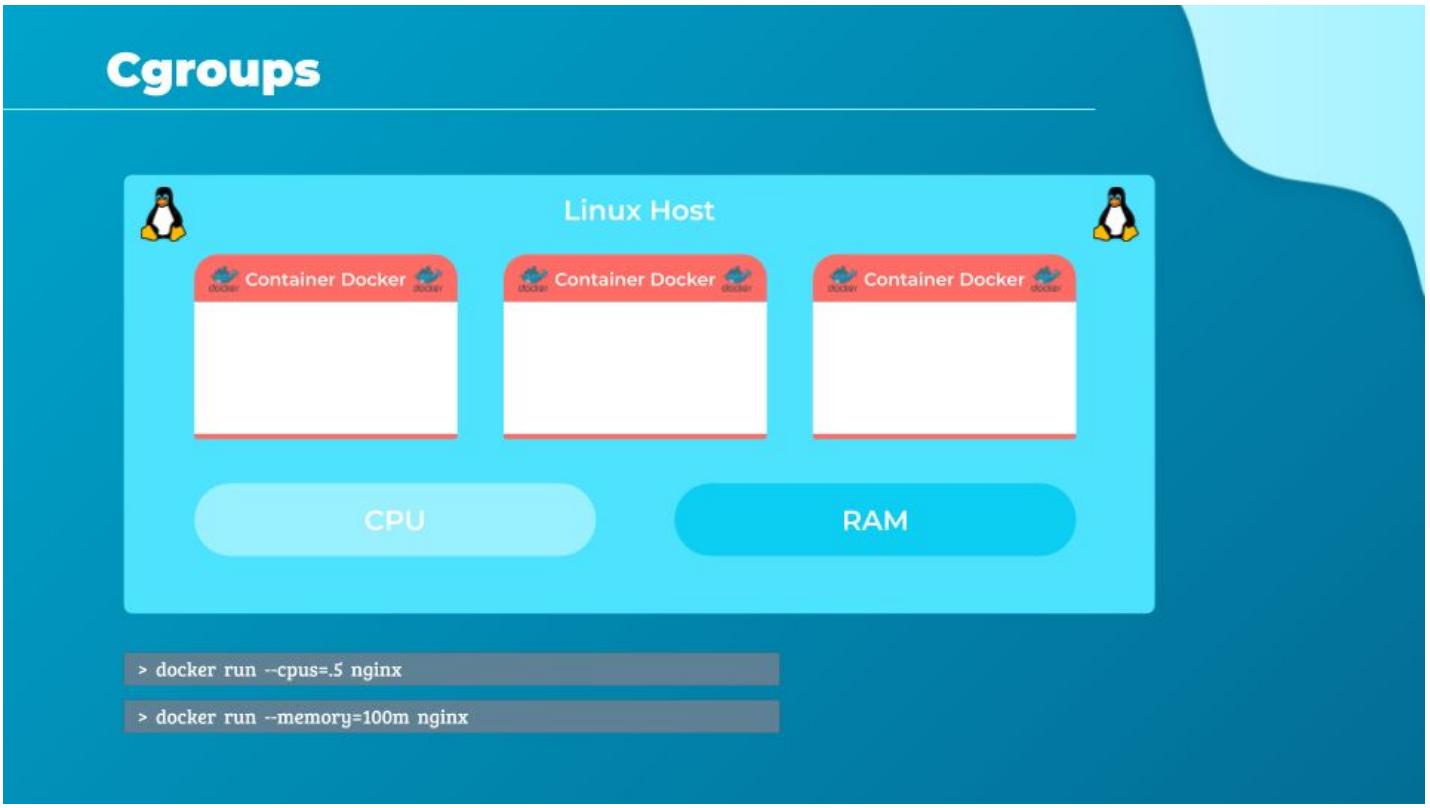
Теперь представим, что мы отключили изоляцию PID. Что бы произошло? Процессы в контейнере приобрели такие же PID номера, что и в хостовой системе. Работа приложения в контейнере конечно будет нарушена. Получается, контейнер стал не таким изолированным, мы как бы совместили его реальность с хостовой.

Давай зайдем еще дальше и отключим изоляцию UTS, которая отвечает за имя хоста. Как видишь мы еще больше совместили контейнер и низлежащую ОС.

Теперь отключим изоляцию MNT и NET. Т.о. контейнер не сможет иметь свой собственный IP и собственную файловую систему. Теперь он полностью совпадает с ОС, он больше не изолирован.

Как ты понимаешь, перегородки в этом общежитии контейнеров достаточно эфемерные, потому что и базовый докер-хост, и контейнеры совместно используют одни и те же системные ресурсы, такие как процессор и память.

Ок, я надеюсь стало более понятнее, почему контейнер это совсем не виртуальная машина.



Это подводит нас к вопросу сколько ресурсов будет выделено для хоста и контейнеров, а также как Docker управляет ресурсами и делится ими между контейнерами.

По умолчанию нет ограничений на то, сколько ресурсов может использовать контейнер, и, следовательно, контейнер может в конечном итоге использовать все ресурсы на базовом хосте. Но есть способ ограничить объем ЦП или памяти, который может использовать контейнер. Docker использует контрольные группы (cgroups), чтобы ограничить количество аппаратных ресурсов, выделяемых каждому контейнеру. Это можно сделать, установив параметр --cpus в команде запуска контейнера

Например, `docker run --cpus=.5 nginx` позволит Docker использовать одномоментно не более 50% вычислительной мощности хоста.

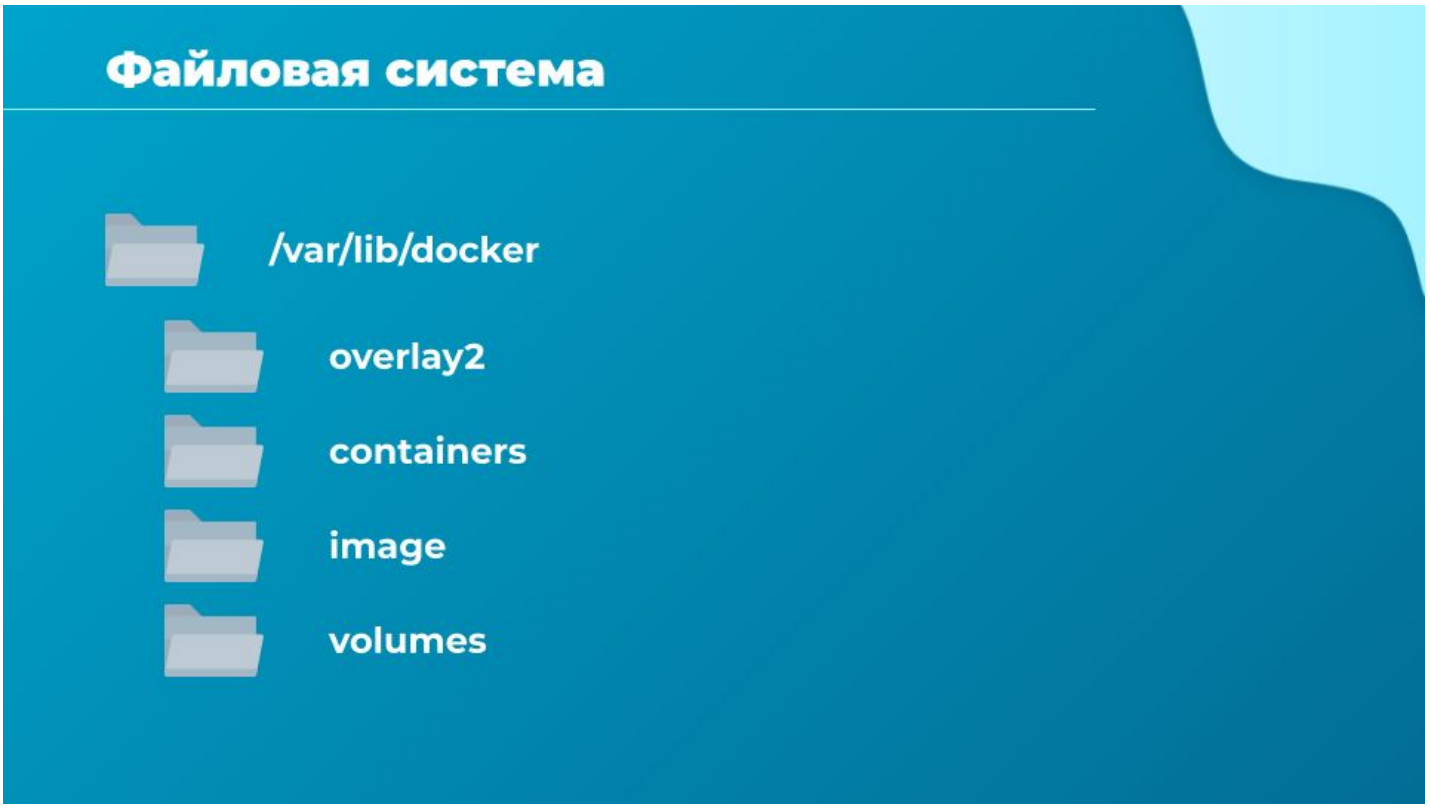
То же самое и с памятью: `docker run --memory=100m nginx`

Данная команда ограничит использование оперативной памяти контейнера лимитом в 100мб.

Я привел лишь два примера, возможности Docker в части управления ресурсами достаточно гибкие. Если хочешь узнать больше, поищи в документации докера по слову limits или "resource constraints".

Это все в этой лекции. Далее мы поговорим о других интересных темах, таких как хранилище и файловых системах в Docker. Жду тебя в следующей!

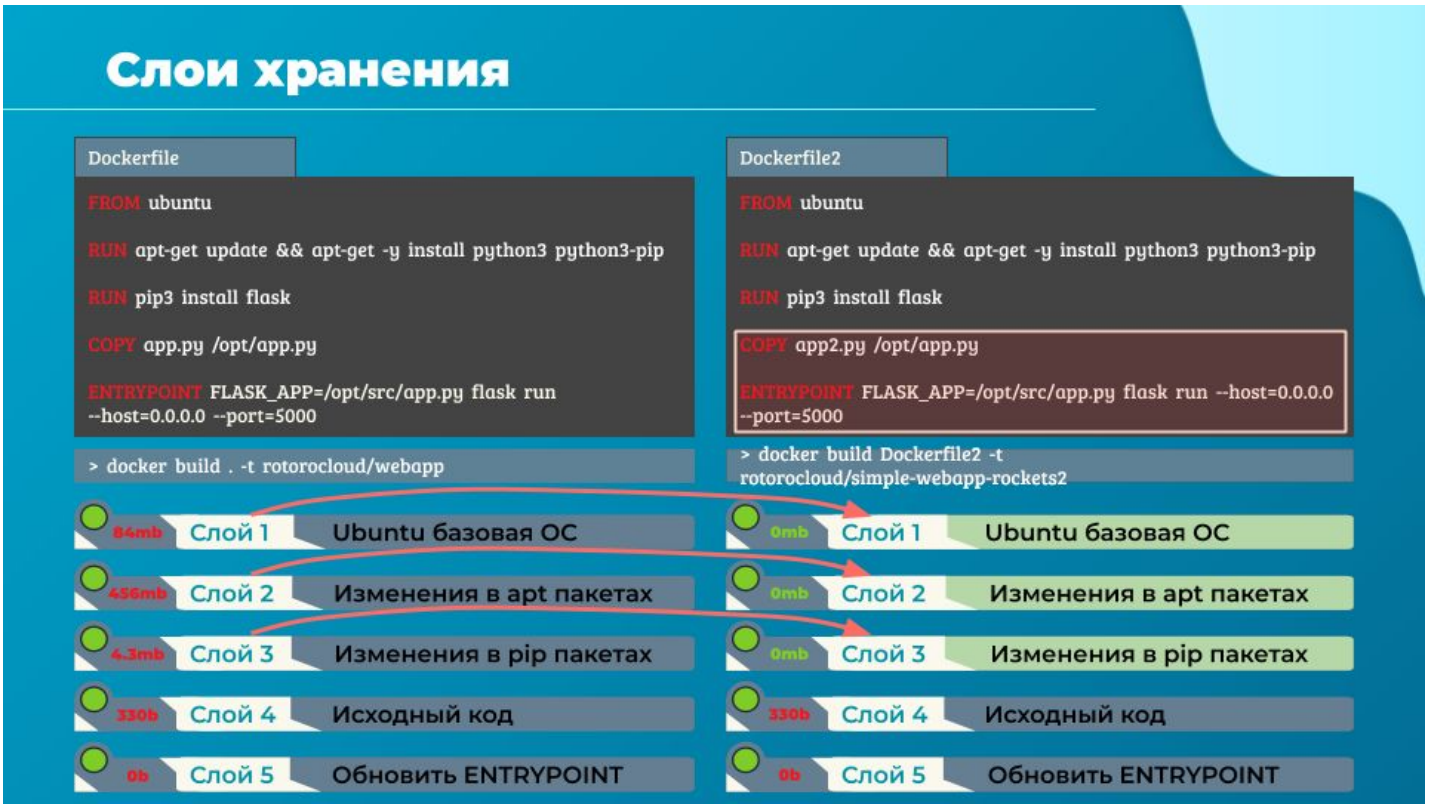
Привет и добро пожаловать на лекцию. В этой лекции мы изучаем продвинутые концепции Docker, а именно поговорим драйверах хранилища Docker и файловых системах. Мы посмотрим, где и как Docker хранит данные, и как он управляет файловыми системами контейнеров.



Давай начнем с того, как Docker хранит данные в локальной файловой системе.

Когда мы устанавливаем Docker в систему, он создает некоторую структуру папок в каталоге /var/lib/docker. Там находится несколько папок, с названиями overlay2, containers, image, volumes и т. д. Здесь Docker по умолчанию хранит все свои данные. Когда я говорю «данные», я имею в виду файлы, связанные с образами и контейнерами, запущенными на этом докер-хосте. Например, все файлы, относящиеся к контейнерам, хранятся в папке containers, а файлы, связанные с образами, хранятся в папке image. Любые тома, созданные для докер-контейнеров, создаются в папке volumes.

Что ж, пока не беспокойся об этом. Мы вернемся к этому чуть позже. А пока давай просто разберемся, где Docker хранит свои файлы и в каком формате. Итак, как именно Docker хранит файлы образа и контейнера?



Чтобы это хорошо понять, нужно вникнуть в многоуровневую или слоеную архитектуру Docker. Давай быстро вспомним то, что мы узнали об этом.

Когда Docker создает образы, он создает их в многоуровневой архитектуре. Каждая строка инструкции в Dockerfile создает новый слой в докер-образе с фиксацией изменений от предыдущего уровня.

Например, первый уровень - это базовая операционная система Ubuntu, за которой следует вторая инструкция, которая создает второй слой, который устанавливает все пакеты APT. Затем третья инструкция создает третий уровень, который занимается пакетами python, за которым следует четвертый уровень, который копирует исходный код. И, наконец, пятый слой, который обновляет точку входа образа.

Поскольку каждый слой сохраняет только изменения из предыдущего, это отражается и на размере. Если мы посмотрим на базовый размер образа, он имеет размер около 80 мегабайт. Пакеты apt, которые были установлены составляют около 450 МБ, а оставшиеся слои малы, чтобы понять преимущества этой многоуровневой архитектуры.

Теперь посмотрим второе приложение, в котором есть другой Dockerfile, но он очень похож на наше первое приложение. В нем используется тот же базовый образ, что и раньше, это ubuntu и оно использует те же зависимости Python и Flask, но имеет в себе другой исходный код для создания образа и также другую точку входа.

Когда я запускаю команду `docker build`, чтобы создать новый образ для этого приложения, первые три уровня обоих приложений будут одинаковы и Docker не будет создавать эти первые три уровня. Вместо этого он повторно применит одинаковые, уже созданные первые три слоя, которые он сделал для первого приложения. Их он возьмет из кэша, в который ложит все слои, когда-либо собранные на докер-хосте.

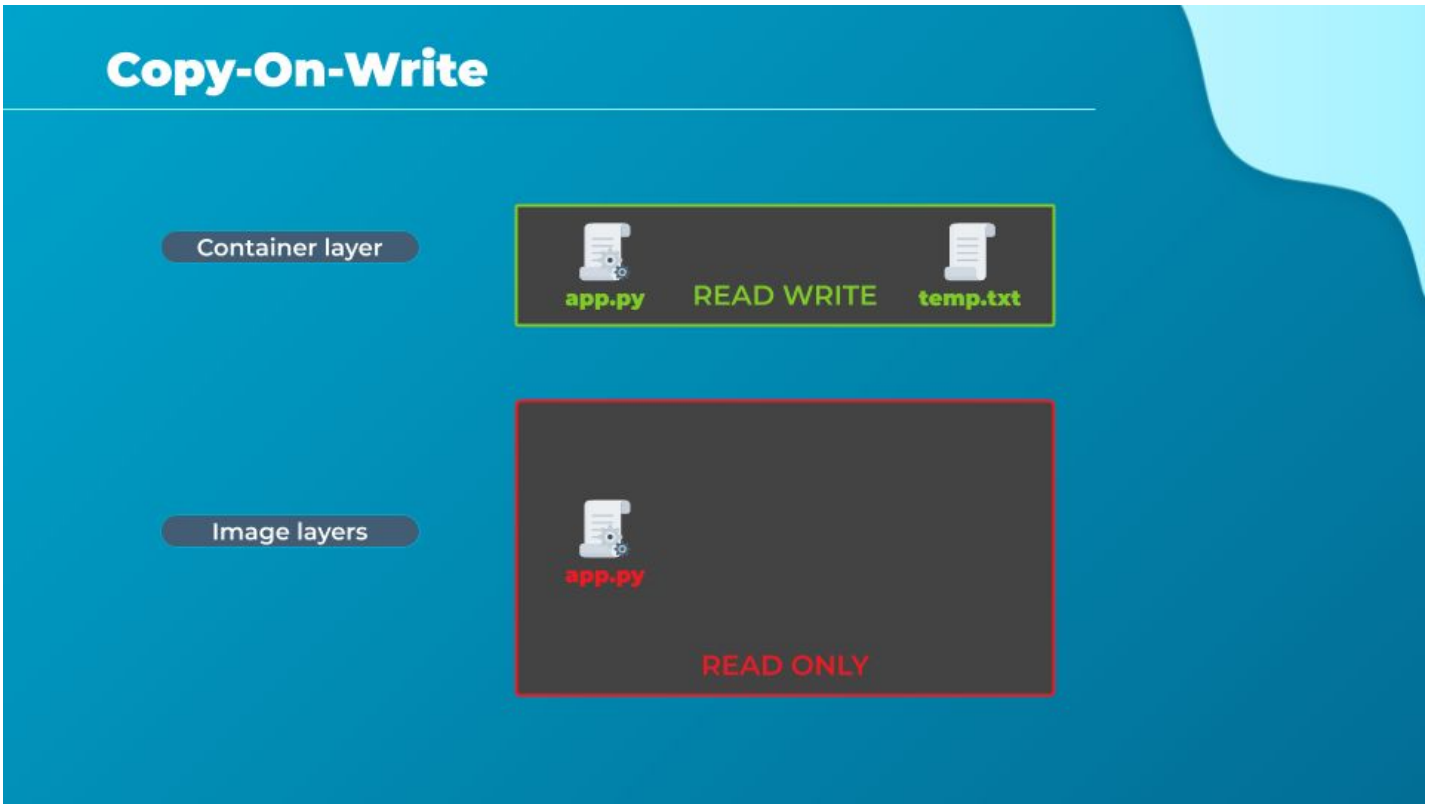
Далее, он создаст только последние два слоя с новым исходным кодом и новой точкой входа. Вновь собранные слои он также положит в кэш. Таким образом Docker создает образы быстрее и эффективно экономит дисковое пространство.

Это также применимо, если требуется обновить код своего приложения. Всякий раз, когда нужно обновить исходный код, например app.py, как в нашем случае, Docker просто повторно использует все предыдущие слои из кэша и быстро перестроит образ приложения, обновив слои с исходным кодом и следующие за ним. Таким образом, это сохранит нам много времени во время перестроек и обновлений образов.

Для наглядности, давай разместим слои снизу вверх, чтобы мы могли все это лучше понять.

Внизу у нас есть базовый уровень ubuntu, затем пакеты, затем зависимости, а затем исходный код приложения и точка входа. Все эти слои создаются, когда мы запускаем команду `docker build`. Формируется окончательный образ Docker, состоящий из этих слоев. После завершения создания образа мы не можем изменять содержимое этих слоев, они доступны только для чтения, а изменить их возможно только через запуск новой сборки.

Когда мы запускаем контейнер, основанный на этом образе при помощи команды docker run, Docker создает контейнер на основе этих слоев, а далее создает новый слой поверх слоев образа, с которого можно не только читать, но и в который можно писать. Слой с возможностью записи используется для хранения и изменения данных, созданных контейнером. Таких данных, как логи, временные файлы и прочие вещи, необходимые для работы приложения. В него также попадает любой файл, измененный пользователем в этом контейнере. Срок жизни этого слоя ограничен временем, пока жив контейнер. Когда контейнер разрушается, этот слой со всеми сохраненными в нем изменениями уничтожается.



Помни, что одни и те же слои образа делятся между всеми контейнерами, запущенными с использованием этого образа. Скажем, я зайду в созданный контейнер и создам там новый файл с именем temp.txt. Этот файл будет создан на уровне контейнера, который доступен для чтения и записи.

Как мы только что сказали, что файлы в слоях образа доступны только для чтения, что означает, что мы не можем ничего редактировать в этих слоях. Теперь давай возьмем пример кода нашего приложения, поскольку мы запекли наш код в образ. Код является частью одного их слоев образа, и, следовательно он будет доступен только для чтения после запуска контейнера. Что, если я хочу изменить исходный код, скажем для какого-то теста?

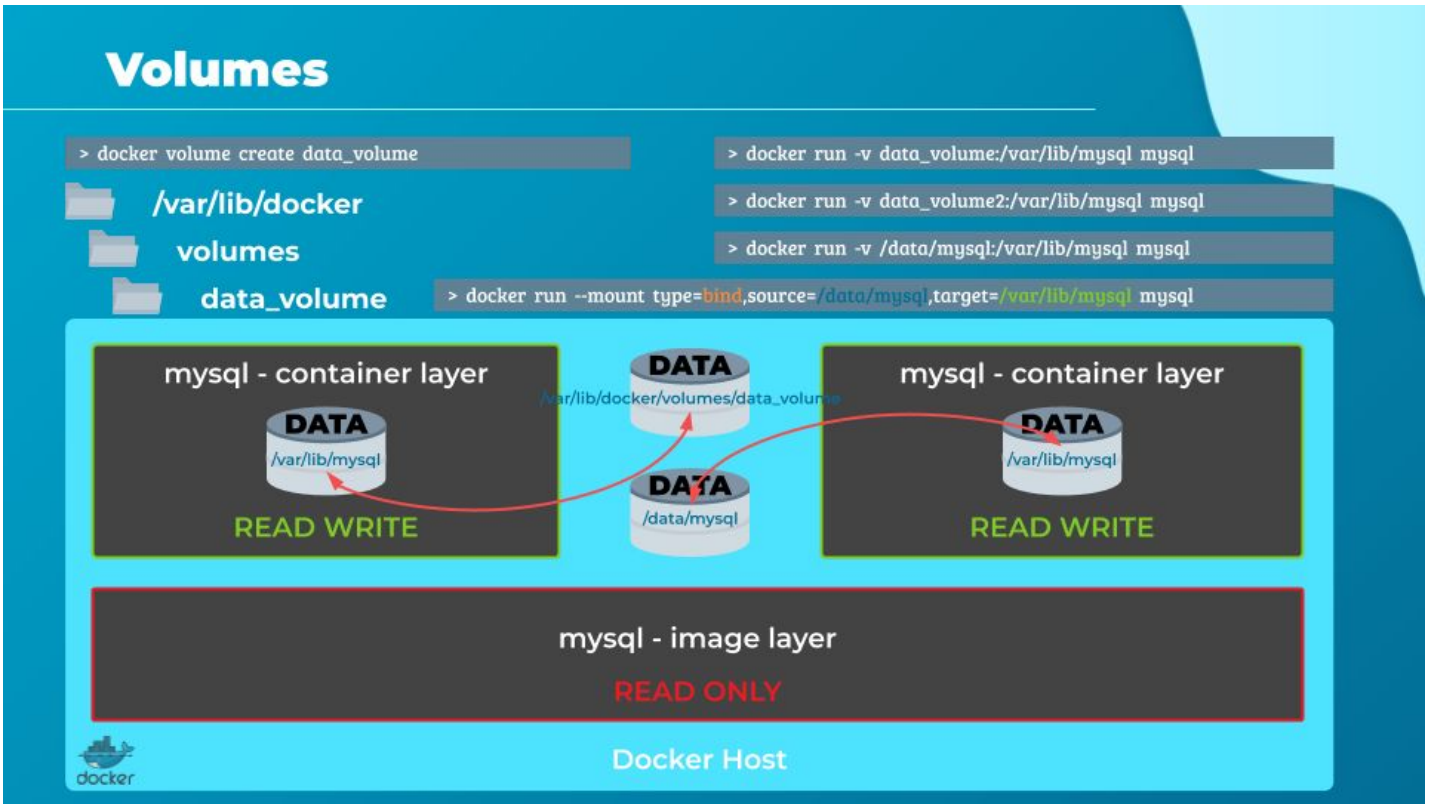
Вспоминаем, что один и тот же слой образа может использоваться несколькими контейнерами, созданными из этого образа. Значит ли это, что я не могу изменить этот файл внутри контейнера?

Нет, я все еще могу изменить этот файл, но прежде, чем я сохраню измененный файл, Docker автоматически создает копию файла на уровне чтения и записи, и затем я буду изменять и работать уже с другой версией файла, которая уже будет находится в другом слое - в слое контейнера, слое чтения и записи. Все будущие изменения будут внесены в эту копию файла на уровне чтения-записи. Это называется механизмом копирования при записи (copy-on-write).

Слои образа доступны только для чтения, это означает, что файлы в этих слоях не будут изменены на самом деле, образ будет оставаться неизменным все время, пока ты не перестроишь его с помощью команды `docker build`.

Итак, пока файлы не изменились, их не существует в слое контейнера. Они все где-то в слоях образа. Что происходит, когда мы избавляемся от контейнера?

Все данные, которые хранились на уровне контейнера, также удаляются. Изменения, которые мы внесли в app.py и новый временный файл, который мы создали, также будут удалены. А что, если мы хотим сохранить эти данные?



Например, мы работали с базой данных и хотели бы сохранить данные, созданные контейнером. Для этих целей мы можем добавить в контейнер постоянный том (volume).

Чтобы сделать это, сначала создадим том с помощью команды `docker volume create`.

Когда мы запускаем команду `docker volume create data_volume`, она создает папку с именем data_volume в каталоге /var/lib/docker/volumes. Затем, когда я запускаю докер-контейнер с помощью команды `docker run`, я могу смонтировать этот том внутрь контейнера для чтения и записи, используя опцию -v, как ты видишь на экране.

Я указал в `docker run -v` и имя моего вновь созданного тома, за которым идет двоеточие и местоположение внутри моего контейнера, которым является местоположением по умолчанию, где MySQL хранит данные, т.е. /var/lib/mysql. Далее указал имя образа.

Теперь Docker создаст новый контейнер и смонтирует созданный ранее том в папку /var/lib/mysql внутри контейнера, и все данные, записанные базой данных, фактически будут хранится в томе, созданном на докер-хосте. Даже если контейнер разрушится, данные остаются целыми. А что, если мы не запускали команду `docker volume create` для создания тома, прежде чем сделали `docker run`?

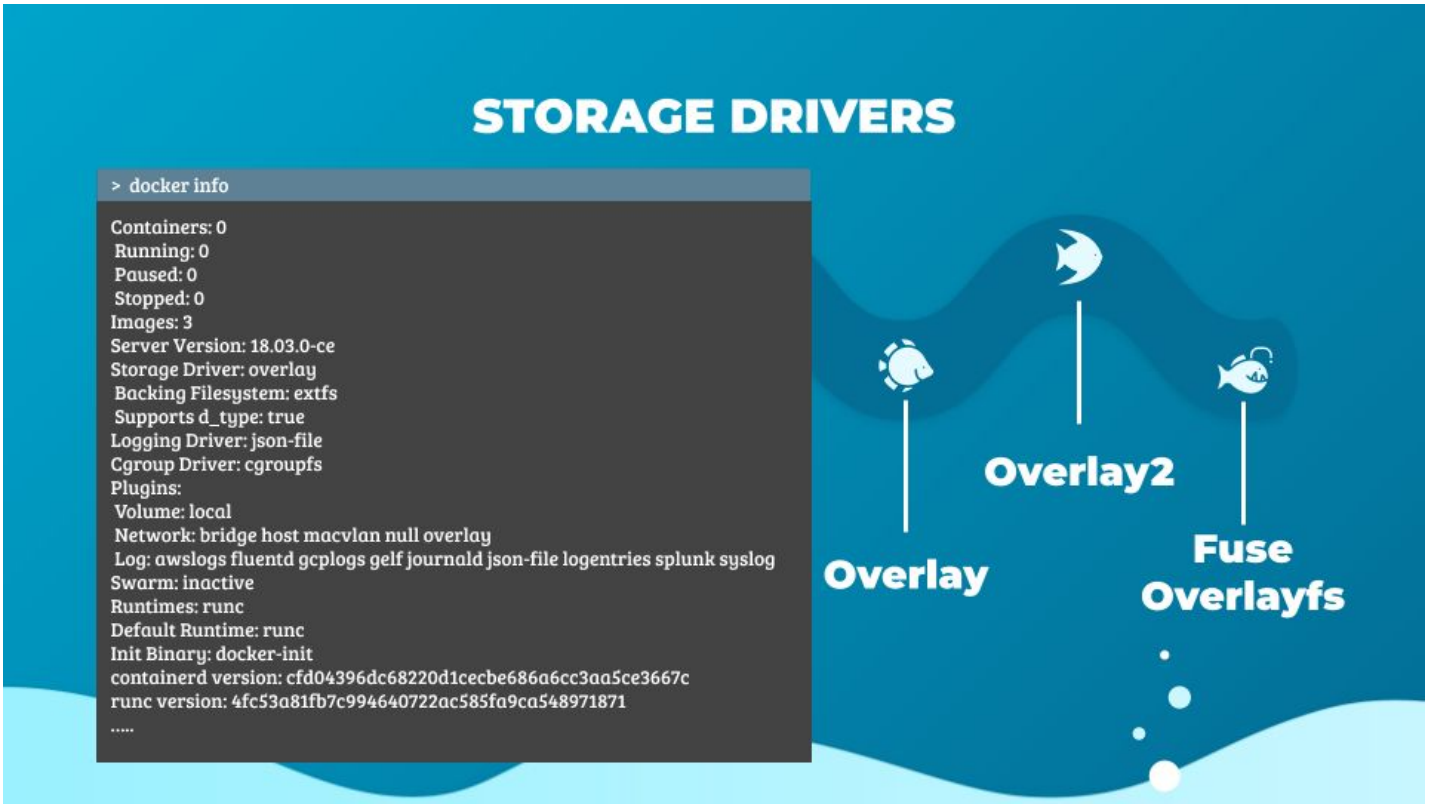
Например, я запускаю команду `docker run` для создания нового экземпляра контейнера mysql с параметром монтирования тома data_volume_2, но том я еще не создал. В таком случае Docker автоматически создаст том с именем data_volume_2 и подключит его к контейнеру. Мы можем найти все эти созданные тома, если просмотрим содержимое папки /var/lib/docker/volume. Это называется volume mounting, поскольку мы монтируем том, созданный Docker в папке /var/lib/docker/volume. Но что, если мы уже разместили данные в другом месте и не хотим их переносить в папку /var/lib/docker/volume?

Например, у нас есть директория в хранилище докер-хоста в размещении /data, и мы хотели бы хранить данные базы данных в качестве тома в этой папке, а не в директории томов docker по умолчанию. В этом случае мы также запустим контейнер с помощью команды `docker run -v`. Но в

этом случае мы предоставим полный путь в папку, которую хотим смонтировать. Это /data/mysql, и Docker создаст контейнер и подключит папку к контейнеру. Это называется bind mounting.

Таким образом, существует два типа монтирования: монтирование тома и монтирование с привязкой. Volume mounting монтирует том из каталога томов, а bind mounting монтирует каталог из любого места на докер-хосте.

И напоследок отмечу, что использование -v - это старый стиль. Новый способ - использовать опцию --mount. Это считается более предпочтительным способом, поскольку он более подробный, хотя на практике я его встречаю реже. В этом случае, мы должны указать каждый параметр в формате «ключ равно значение». Например, предыдущая команда может быть записана с параметром --mount, и далее указав опции source и target. Type этом случае - bind, источник - это местоположение на моем хосте, а цель - это местоположение в моем контейнере.



Ок, а кто несет ответственность за выполнение всех этих операций. Сохранение многоуровневой архитектуры. Создание слоя с возможностью записи, перемещение файлов между слоями для возможности копирования и записи и т. д.

Это storage drivers. Таким образом, Docker использует драйверы хранилища для обеспечения многоуровневой архитектуры. Некоторые из распространенных драйверов хранения: AUFS, BTRFS, ZFS, device-mapper, overlay и overlay 2, fuse overlayfs. Некоторые из них широко используются, некоторые уже устарели, но могут встретиться в каких-то легаси нагрузках.

Выбор драйвера хранилища зависит от используемой ОС. Например, у Ubuntu.
Драйвер хранилища по умолчанию для нее - это overlay2. Но до 18 версии Docker это был aufs, и он был недоступен в таких операционных системах, таких как Fedora или CentOS, и в них приходилось использовать device-mapper.

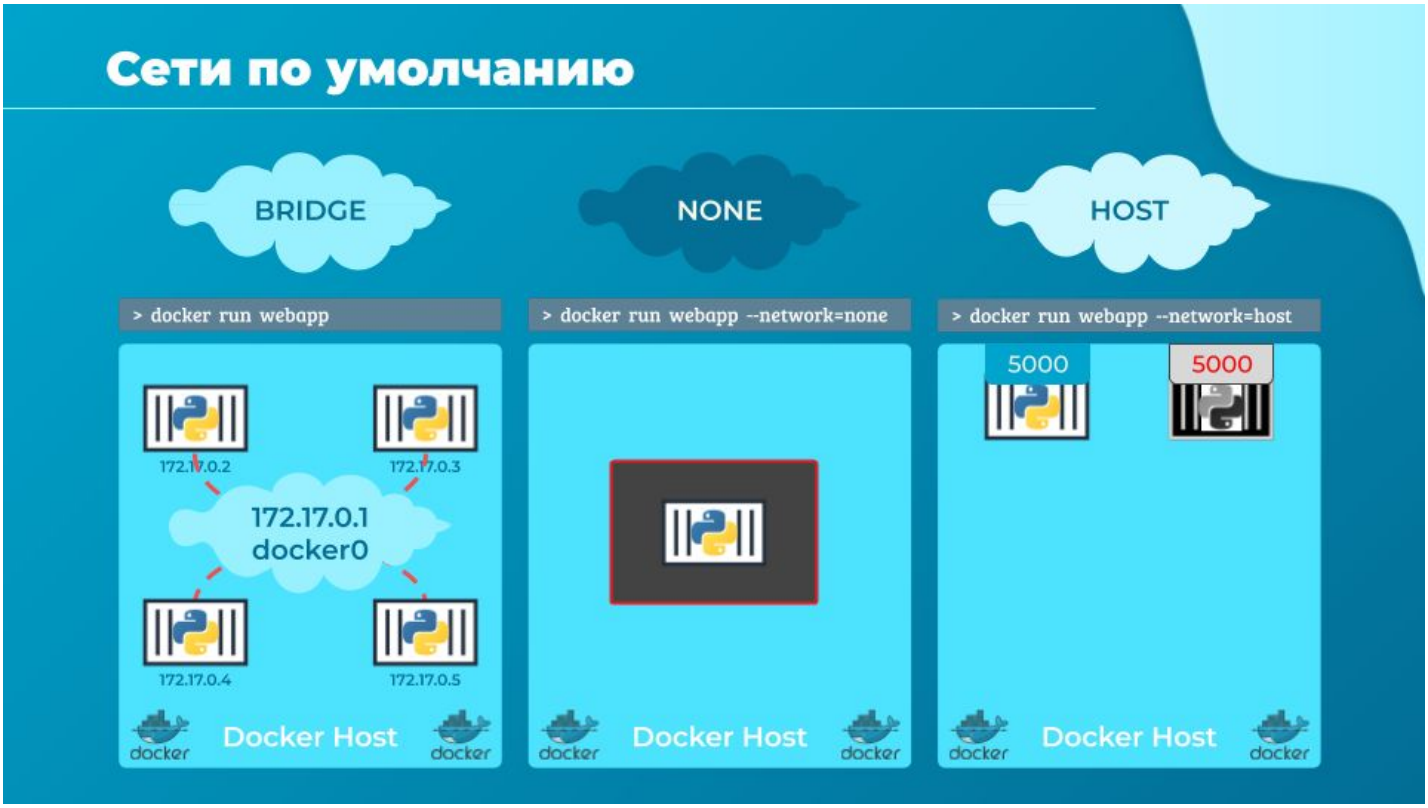
Docker сам постарается выбрать лучший драйвер хранилища, который доступен в конкретной операционной системе. Но это не всегда работает хорошо. В случае проблем нужно настраивать

руками, изучив соответствующие issues по конкретной ОС и ее ядру. Где-то до 18 года это было серьезной проблемой, с появлением overlay2 дела стали обстоять гораздо лучше.

Различные драйверы хранилища также обеспечивают разные характеристики производительности и стабильности, поэтому ты можешь выбрать тот, который соответствует потребностям твоего приложения и соответствии требования твоей организации. Если хочешь быть осведомленным о каком-либо из этих драйверов хранения, начни с документации Docker, а далее погрузись в десятки issues на гитхаб докера.

Это все из концепций хранения в Docker. Увидимся на следующей лекции.

Привет, добро пожаловать на лекцию, где мы посмотрим на сеть в докере.



Когда мы устанавливаем Docker, он автоматически создает три сети:

- **bridge**
- **none**
- **host**

Мостовая (bridge) сеть - это сеть по умолчанию, к которой подключается контейнер. Если мы хотим связать контейнер с любой другой сетью, то потребуется явно указать информацию о сети. Используй параметр --network командной строки, как я показал на экране. Мы сейчас рассмотрим каждую из этих сетей поближе.

Сеть Bridge - это частная внутренняя сеть, созданная Docker на хосте. Все контейнеры подключены к этой сети по умолчанию. Как правило диапазон этих внутренних IP-адресов 172.17, но это настраивается. Контейнеры получают доступ друг к другу, используя этот внутренний IP-адрес. Если требуется доступ к какому-то из этих контейнеров из внешнего мира, нужно сопоставить порты контейнеров с портами на докер-хосте.

Другим способом доступа к контейнерам извне, о котором я упомянул, является вариант прямого связывания контейнера с сетью хоста. Это убирает всю сетевую изоляцию между докер-хостом и докер-контейнером. Что будет означать, если мы запускаем веб-сервер на порту 5000 в веб-контейнере, он автоматически становится доступным извне на том же порту своего хоста, без необходимости какого-либо сопоставления портов.

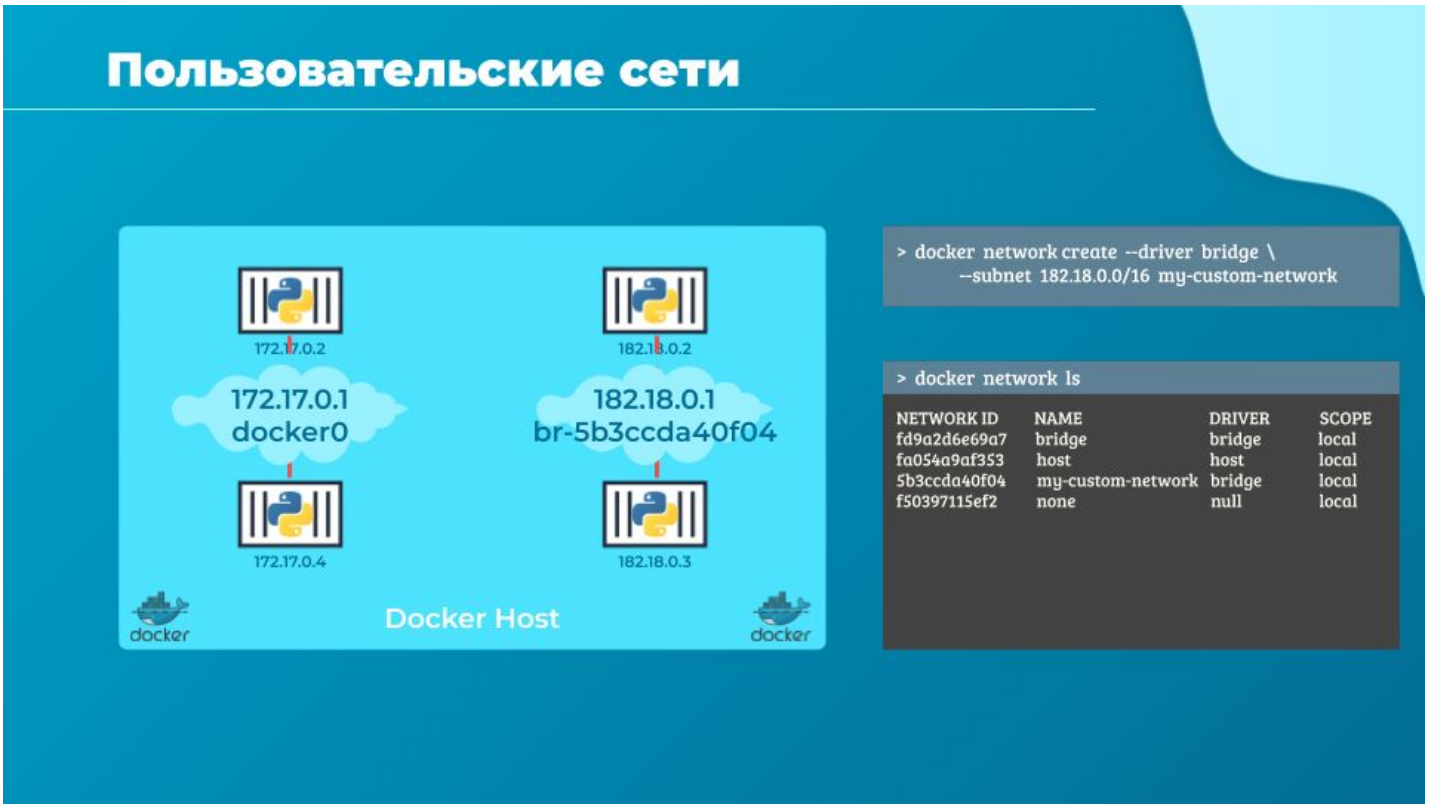
Здесь этот веб-контейнер будет использовать все сетевые ресурсы своего хоста. И это также будет означать, что, в отличие от прошлого примера, теперь мы не сможем запускать несколько контейнеров в одном хосте на одном и том же порту. Поскольку теперь нет ни внутренних и не внешних портов, порты являются общими. Службы самого хоста, разумеется, также занимают какие-то порты, и это нужно учитывать размещая контейнеры в сети типа host.

Теперь сеть none. Контейнеры не являются членами какой-либо сети и не имеют доступа к внешней сети или другим контейнерам, которые запущены в изолированных сетях.

На самом деле это еще не все. Для нагрузок, связанных с оркестрацией есть тип сети Overlay. Этот тип сети адаптирован для работы одной сети на нескольких узлах. Далее в Docker есть тип Macvlan, она будет интересна, если ты собираешься докеризировать нагрузки, которые раньше размещались в виртуальных машинах, эта штука полностью эмулирует физический хост в части сети. Также Docker позволяет с разной степенью успешности интегрировать кастомные сетевые плагины.

Эти сети узко специфичные, и о них мы будем говорить в продвинутом курсе Docker.

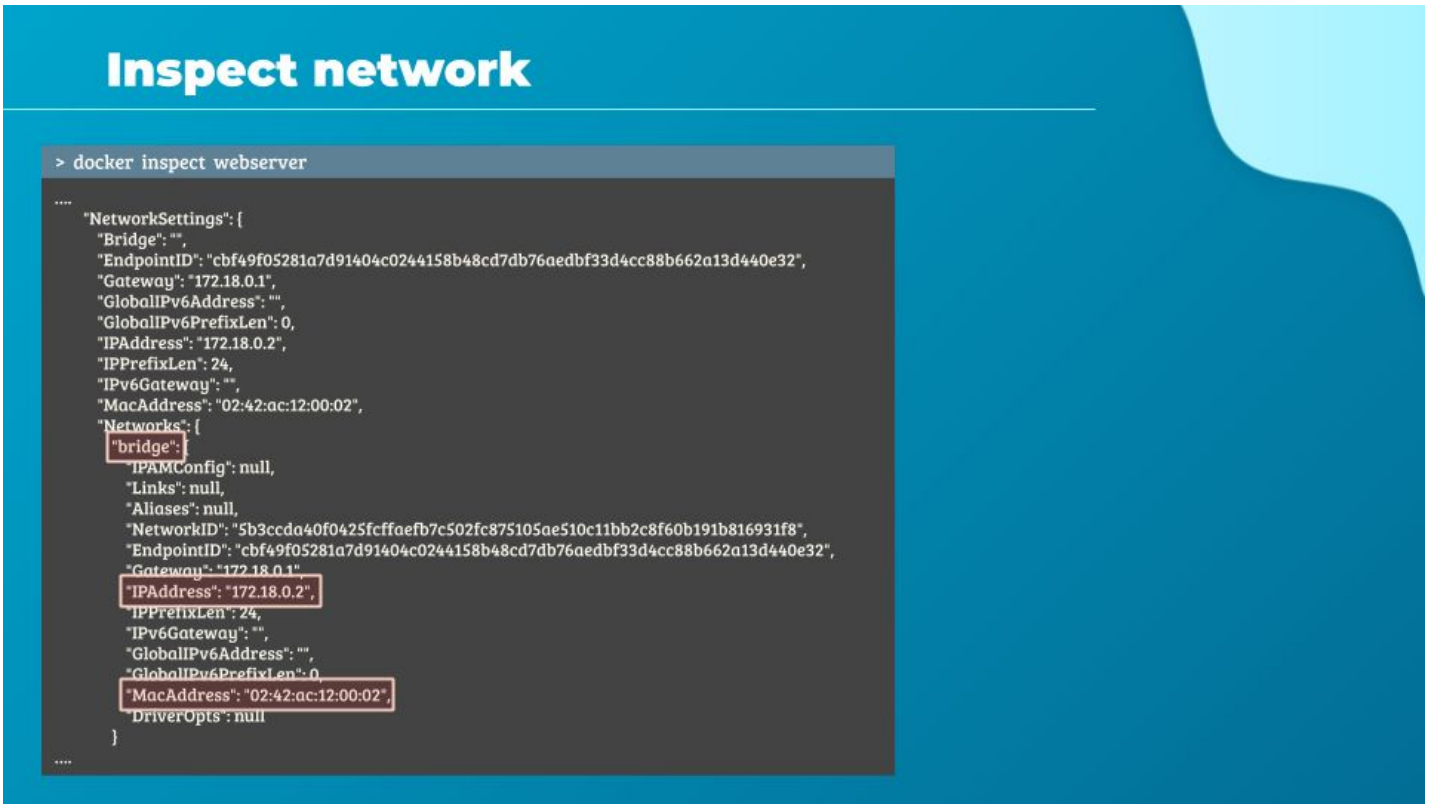
Итак, мы только что увидели мостовую сеть по умолчанию с идентификатором 172.17.0.1. Таким образом, все контейнеры, ассоциированные с этой сетью, смогут общаться друг с другом.



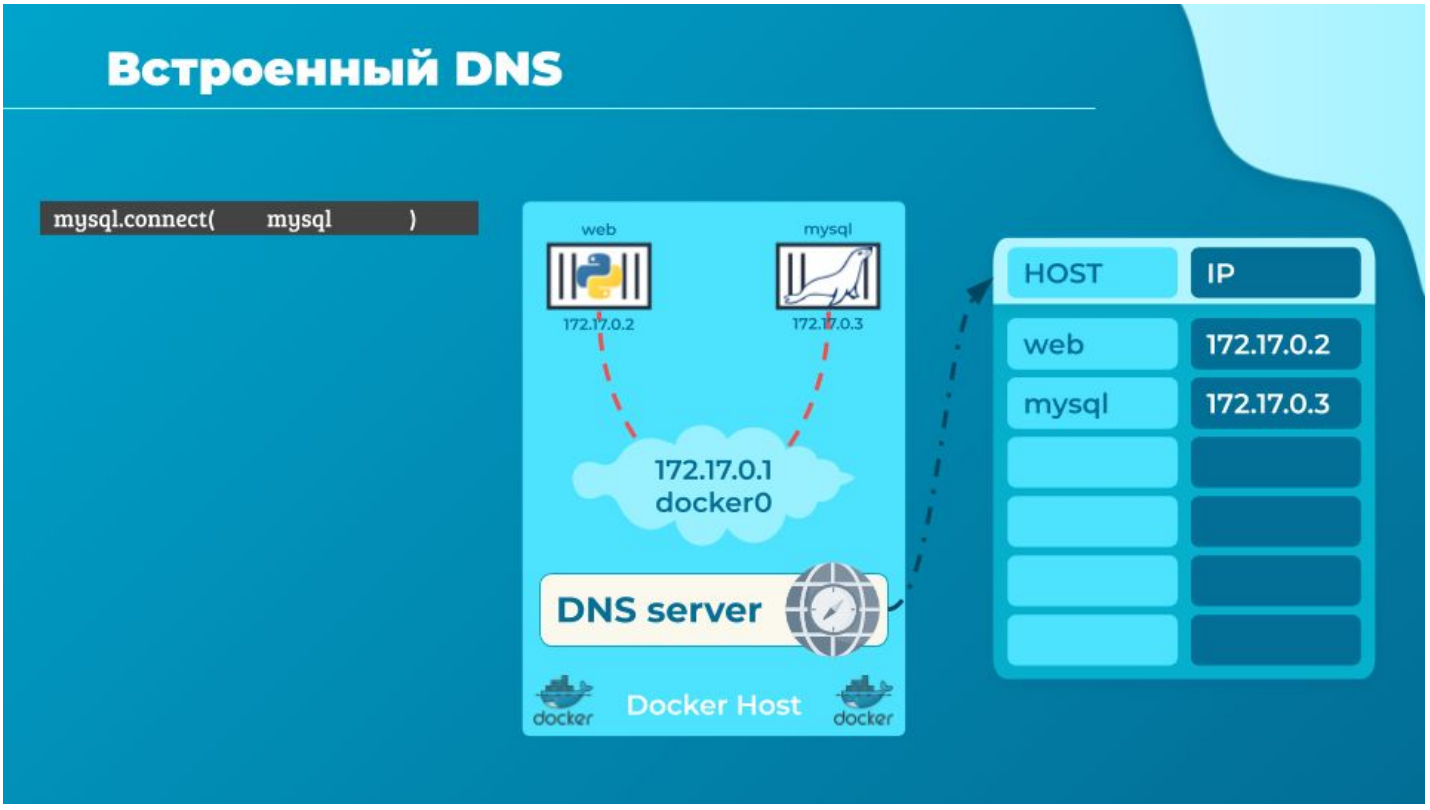
Но что, если мы хотим изолировать контейнеры внутри докер-узла, например, первые два контейнера во внутренней сети 172 и два вторых контейнера в другой внутренней сети, например 182?

По умолчанию Docker создает только одну внутреннюю мостовую сеть. Но мы можем сами создать свою собственную внутреннюю сеть с помощью команды `docker network create` и указать тип сети в параметре --driver, в нашей команде это bridge.

Мы указываем диапазон адресации новой сети в CIDR нотации после параметра --subnet. В этом случае 182.18.0.0/16 и за ним следует пользовательское имя этой новой изолированной сети. Теперь запустим команду `docker network ls`, чтобы вывести список всех сетей.



Для того, чтобы увидеть настройки сети и IP-адрес, назначенный существующему контейнеру, запустим команду `docker inspect` с названием контейнера или его ID. Там, в разделе NetworkSettings -> Networks мы можем увидеть тип сети, к которой контейнер привязан, его внутренний IP-адрес, Mac-адрес.



Как ты помнишь в bridge network другие контейнеры могут связываться друг с другом, используя свои имена. Например, здесь у меня есть веб-сервер и контейнер базы данных MySQL, работающие на одном хосте в одной сети. Как я могу реализовать доступ для моего веб-сервера к базе данных в контейнере?

Один из вариантов, который я мог бы сделать - это использовать внутренний IP-адрес, назначенный контейнеру MySQL, который в данном случае 172.17.0.3. Но это не очень правильно, поскольку не гарантируется, что контейнер получит тот же IP-адрес, когда система перезагрузится или контейнер по какой-то причине сломается и его место займет новый. Лучший способ сделать это - использовать имя контейнера.

Все контейнеры на докер-хосте могут находить адреса друг друга с помощью имени контейнера. Docker имеет встроенный DNS-сервер, который помогает контейнерам разрешать имена друг друга. Обрати внимание, что встроенный (embedded) DNS-сервер всегда работает по адресу 127.0.0.11. Вот так Docker реализует сети. А какие технологии стоят за этим?

Это изоляция network namespace. Т.е. для каждого контейнера используется свое пространство имен, которые позволяют разному трафику передаваться и обрабатываться по разным правилам, и делать это рядом друг с другом и незаметно друг для друга. Также используются виртуальные Ethernet адаптеры для соединения контейнеров вместе.

Вобщем-то это все, что в общих чертах я могу рассказать о сети. Сети тема обширная и непростая, в курсе для начинающих мы не будем глубоко погружаться в эти практики.

Подробнее об этом я планирую рассказать в своем следующем курсе по продвинутому Docker, который будет в 2021 году.

А на этом все в этой лекции о сетевых технологиях, делай практику и жду тебя в следующей!

Привет. В этой лекции мы посмотрим на Docker registry.

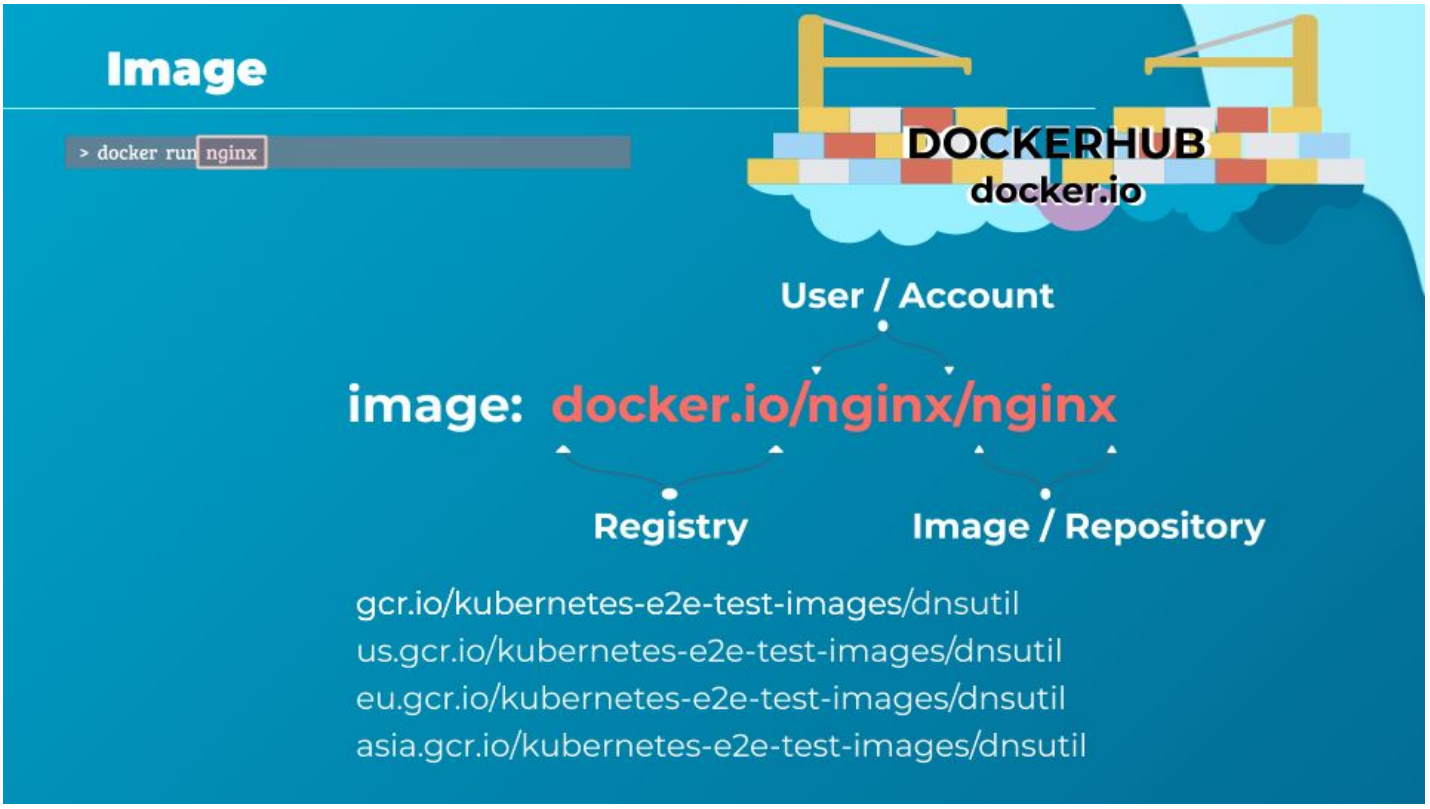
Реджистри - это сервер, который позволяет поддерживать репозитории докер-образов. Он позволяет хранить и распространять эти образы.

Иногда это выражение используют в более широком смысле, говоря о реджистри, как о том месте, где хранятся образы. Мы уже привыкли, что образы лежат на докерхабе, но это не всегда верно. В продакшене в основном используют частные докер-реджистри.

Для построения добротных DevSecOps-конвейеров используют приватные репозитории, в которых образы проходят необходимые этапы тестирования, секьюрити и комплаенс проверки. Также эти

реджистри используют при разработке для кэширования часто используемых образов (в свете введенных Docker ограничений это актуально) или для пуллинга предварительно подготовленных образов, когда в компании существует явный запрет на пользование 3rd party images. В основном из-за соображений безопасности.

Есть различные варианты размещения данных серверов: в облаке, он-премис для компании или локально для небольшой команды разработчиков. Есть менеджед решения, где вы получаете обслуживаемый репозиторий (например jfrog) или как платформу (например gcr.io).



Ок, давай посмотрим на простой контейнер nginx, который мы запустили в Docker следующей командой: `docker run nginx`.

Давай подробнее рассмотрим это название образа. Его имя - nginx, но что это за образ и откуда он взят? Это имя соответствует соглашению об именовании образов Docker, “nginx” здесь название образа или имя репозитория.

Когда мы сказали Docker nginx, то на самом деле это nginx/nginx. Первая часть обозначает пользователя или аккаунт. Поэтому, если мы не предоставили название аккаунта или имя репозитория, предполагается, что оно совпадает с указанным, которым в данном случае является nginx.

Имя пользователя обычно - это имя твоей учетной записи Docker Hub или, если это организация, то имя этой организации. Если ты используешь свою собственную учетную запись и создаешь свои собственные репозитории и образы в них, тебе нужно использовать аналогичный шаблон. А где эти образы хранятся и откуда они извлекаются?

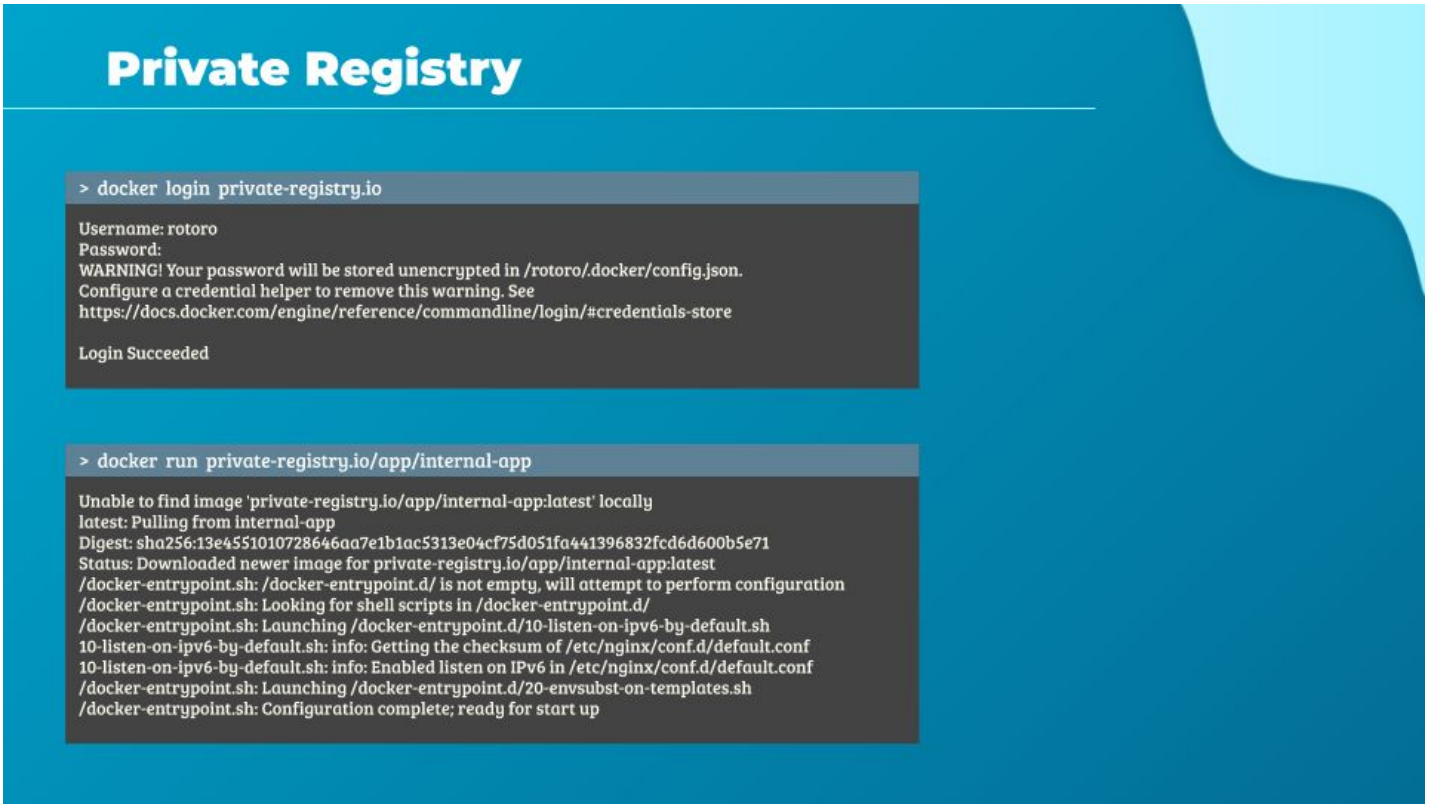
Поскольку мы не указали место, откуда эти образы должны быть извлечены, то предполагается, что он находится в докер-реджистри по умолчанию, а это как мы знаем Docker Hub. DNS-имя которого - Docker.io.

В реестре хранятся все образы. Каждый раз, когда мы создаем новый образ или обновляем существующий, мы помещаем его в реджистри, и каждый раз, когда кто-либо развертывает это приложение, оно извлекается из этого реестра. Есть также много других популярных реестров.

Например, реджистри от Google. Находится он по адресу GCR.io, и я знаю, что там хранится много образов связанных с Kubernetes, например их средства для тестирования кластера. Мы можем запросить эти e2e-тесты из глобальной версии их реджистри, или из версий с географической привязкой, для уменьшения latency при запросах из твоего региона.

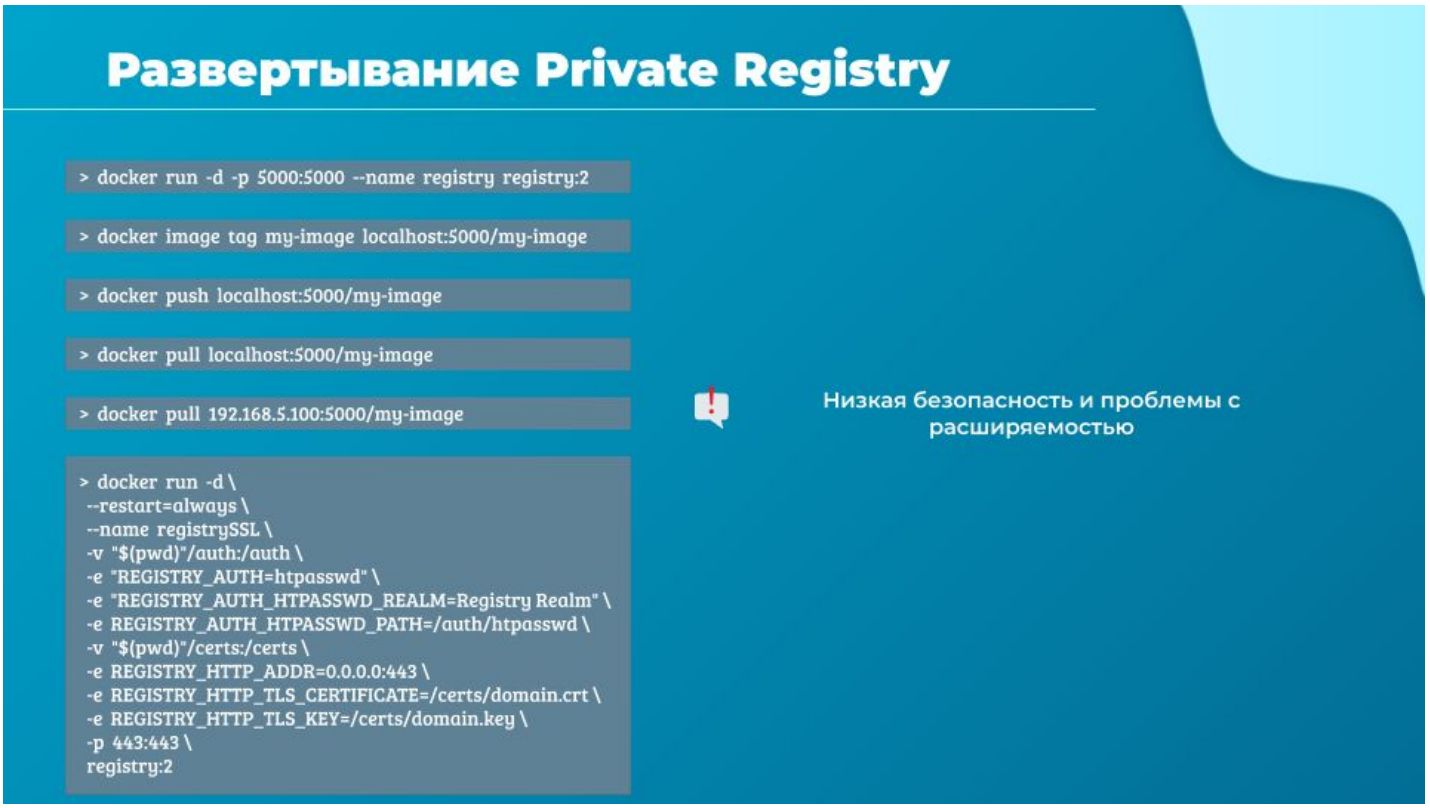
Все это общедоступные образы, которые может загрузить любой пользователь, и свободно использовать для собственных приложений. Но когда мы не хотим выносить это на публику, то размещение внутреннего частного реджистри может быть хорошим решением.

Многие поставщики облачных услуг, такие как AWS, Azure, GCP или Alibaba предоставляют частные реджестри, когда мы открываем у них учетную запись. В любом из решений, будь то реджистри Docker Hub, Google или свой внутренний частный реестр, мы можем сделать репозиторий частным, чтобы к нему можно было получить доступ только с помощью набора учетных данных.



С точки зрения Docker, для запуска контейнера с использованием образа из приватного реджистри сначала нужно выполнить аутентификацию в свой частный реестр с помощью команды `docker login` и ввода своих учетных данных. После этого данные для входа сохраняются на докер-хосте и при взаимодействии с этим реджистри будут использованы.

Для запуска приложения используй название частного реджистри как часть имени образа, как ты видишь на экране. Если ты попробуешь обратиться за образом в частный реджистри, не войдя в него, то получишь сообщение, что образ не может быть найден. Поэтому не забудь всегда входить в систему, прежде чем пушить или пуллить в частный реджистри.



Я сказал, что облачные провайдеры, такие как AWS или GCP, предоставляют возможность использования своих реджистри, когда мы создаем у них учетную запись. Но что, если мы запускаем свое приложение он-премис и не имеем доступа к частному реджистри? Как развернуть собственный частный реестр в своей организации?

ПО Docker registry это не часть поставки Docker, это само по себе другое приложение. И, конечно же, он доступен как докер-образ, его можно развернуть с docker hub. После запуска он предоставляет свои API-вызовы по умолчанию на порту 5000 нашего локального хоста. Как нам закачать туда свой собственный образ?

Используя команду `docker image tag`, чтобы пометить образ для частного реджистри, в который мы хотим это положить. В данном случае это localhost:5000/my-image. Поскольку реджистри работает на том же хосте, я обращаюсь к нему как localhost, далее использую двоеточие и порт 5000, за которым следует имя образа.

После этого я могу отправить этот образ в свой локальный частный реджистри с помощью команды `docker push` и нового имени образа. Чтобы забрать этот образ я должен указать полное имя своего образа в соответствии с соглашением docker, т.е. `docker pull localhost:5000/my-image`. Как ты понимаешь, сделать это я смогу, только с данного докер-хоста, в котором это имя будет разрешено правильно.

Другие машины из локальной сети смогут к нему обратиться с использованием IP адреса (или доменного имени) в качестве имени частного реджистри. Например 192.168.56.100:5000/my-image. Но это будет работать только в случае insecure реджистри и с костылями с сопоставлением имен. Я бы не советовал так делать.

Для того, чтобы локальный докер реджистри корректно работал, требуется удостоверить имя данного реджисти с помощью изданного SSL-сертификата. Это может быть сертификат удостоверяющего центра (в случае наличия домена) или можно настроить реджистри для взаимодействия с самоподписанным сертификатом (действительно для IP адреса или имени

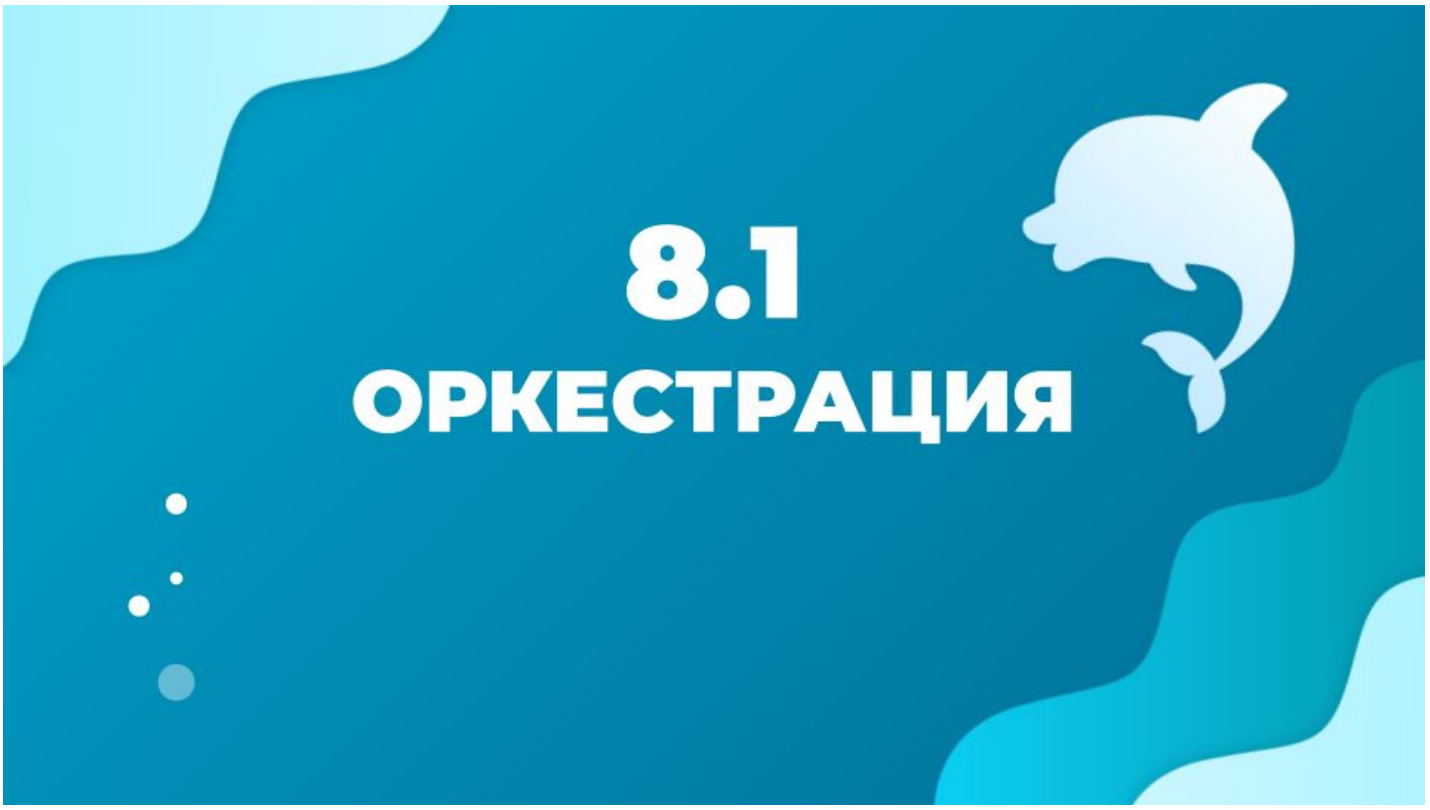
хоста). Но в этом случае следует позаботиться, чтобы все клиенты реджистри имели его сертификат в качестве корневого, иначе они не будут ему доверять.

Как я говорил есть еще вариант инсекьюрного реджистри, без без сертификата. Это практиковалось какое-то время назад, но в данный момент быстрее выпустить самоподписанный сертификат или настроить ротацию бесплатных с помощью certbot или подобным, нежели переконфигурирования докера и реджистри, чтобы все работало в обход проверки подлинности.

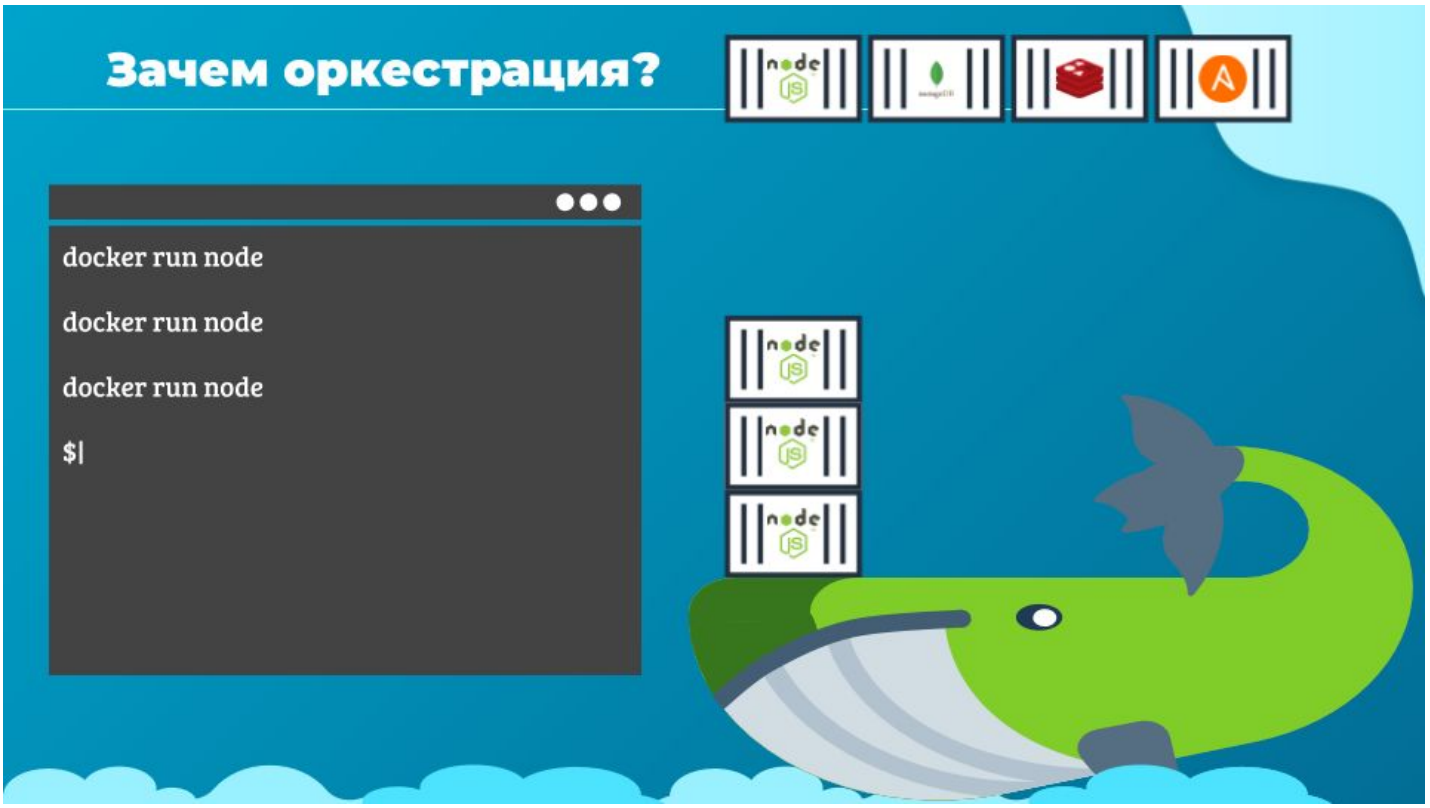
В докментации Docker разобрано несколько примеров запуска реджистри:

- **с отдельным контейнером**
- **с помощью docker-compose**
- **в инфраструктуре docker swarm.**

Вот и все для этой лекции. Смотрим демо о реджистри в Docker.



Привет и добро пожаловать на лекцию по оркестрации контейнеров. Здесь мы поговорим о том, как быть, если одного докер-хоста по каким-то причинам недостаточно для разворачивания проекта, с какими трудностями сталкиваются и как их преодолевают.



Ранее в курсе мы видели, как с запустить отдельный экземпляр приложения с помощью Docker. для этого мы использовали простую команду `docker run`. В этом случае мы запускаем приложение на базе node js с помощью команды docker run node. Но это всего один экземпляр на одном докер-хосте. Что будет, если количество пользователей возрастет настолько, что у запущенного инстанса не станет хватать ресурсов, чтобы справиться с нагрузкой?

Ты можешь развернуть дополнительный экземпляр своего приложения с помощью такой же команды `docker run` и повторить это несколько раз. Т.е. тебе придется самому следить за нагрузкой и производительностью приложения, соответственно реагировать запуская или останавливая дополнительные копии приложения. А если контейнер выйдет из строя, тебе нужно это обнаружить и снова запустить команду docker run, чтобы развернуть другой экземпляр приложения.

А что с докер-хостом? Надо знать как он себя чувствует, мониторить его ресурсы. Что если хост упадет и станет недоступным? В этом случае все наше приложение перестанет отвечать пользователям.

Если мы начнем решать все эти проблемы самостоятельно, то потребуется минимум отдельный инженер, который будет заниматься мониторингом состояний производительности и здоровья контейнеров и хоста и, при необходимости, исправлять ситуацию. Если твое приложение действительно большое, скажем от тысячи контейнеров, такой подход совершенно неприменим.

Разумеется можно начать самому автоматизировать эти процессы, написав какие-то скрипты. И это будет как-то работать. Проблемы начнуться очень скоро, когда:

- **поменяются сотрудники - новых придется переучивать,**
- **ротируются разработчики, у других может быть свой взгляд на автоматизацию,**
- **постоянно появляются новые технологии, которые требуют новых интеграций,**
- **появятся вызовы безопасности, которые вскроют все проблемы непродуманной архитектуры или каких-то временных компромиссов, сделанных для упрощения в прошлом.**

Решения по оркестрации контейнеров помогут нам в этом случае. Эти решения содержат в себе набор подходов, технологий и инструментов, которые обслуживают контейнерные нагрузки в продакшене. Как правило, подобное решение реализует в себе несколько докер-хостов, чтобы в случае неисправности одного из них работа не останавливалась, и пользователь не замечал простоя.

В случае проблем, система оркестрации сама переключит пользователя на другой докер-хост и попробует вернуть в строй неисправный. Также эти решения для оркестрации позволяет легко развернуть сотни или тысячи экземпляров приложения с помощью одной команды.

Эта команда используется в Docker swarm. Мы ее немного рассмотрим в следующей лекции.

Некоторые решения для оркестрации могут помочь нам автоматически увеличить количество экземпляров при увеличении количества пользователей, и уменьшить количество экземпляров при уменьшении спроса. Часть решений могут даже помочь нам в автоматическом добавлении дополнительных хостов для поддержки пользовательской нагрузки, а не только в кластеризации и масштабировании количества контейнеров.

Также они обеспечивают поддержку специальной сети между этими контейнерами на разных хостах, и обслуживают балансировку нагрузки пользовательских запросов между разными докер-хостами. Из этих решений можно осуществлять управление конфигурацией и безопасностью в кластере и обеспечивать совместное использование хранилища между хостами.

Сегодня доступно несколько систем для оркестрации контейнеров. У Docker это решение Docker Swarm, Kubernetes от Google и MESOS от Apache.

По сравнению с другими, Docker Swarm очень легко настроить и начать пользоваться, это хорошее решение для начала. Но с другой стороны в нем не хватает многих нужных функций, которые требуются для сложных приложений.

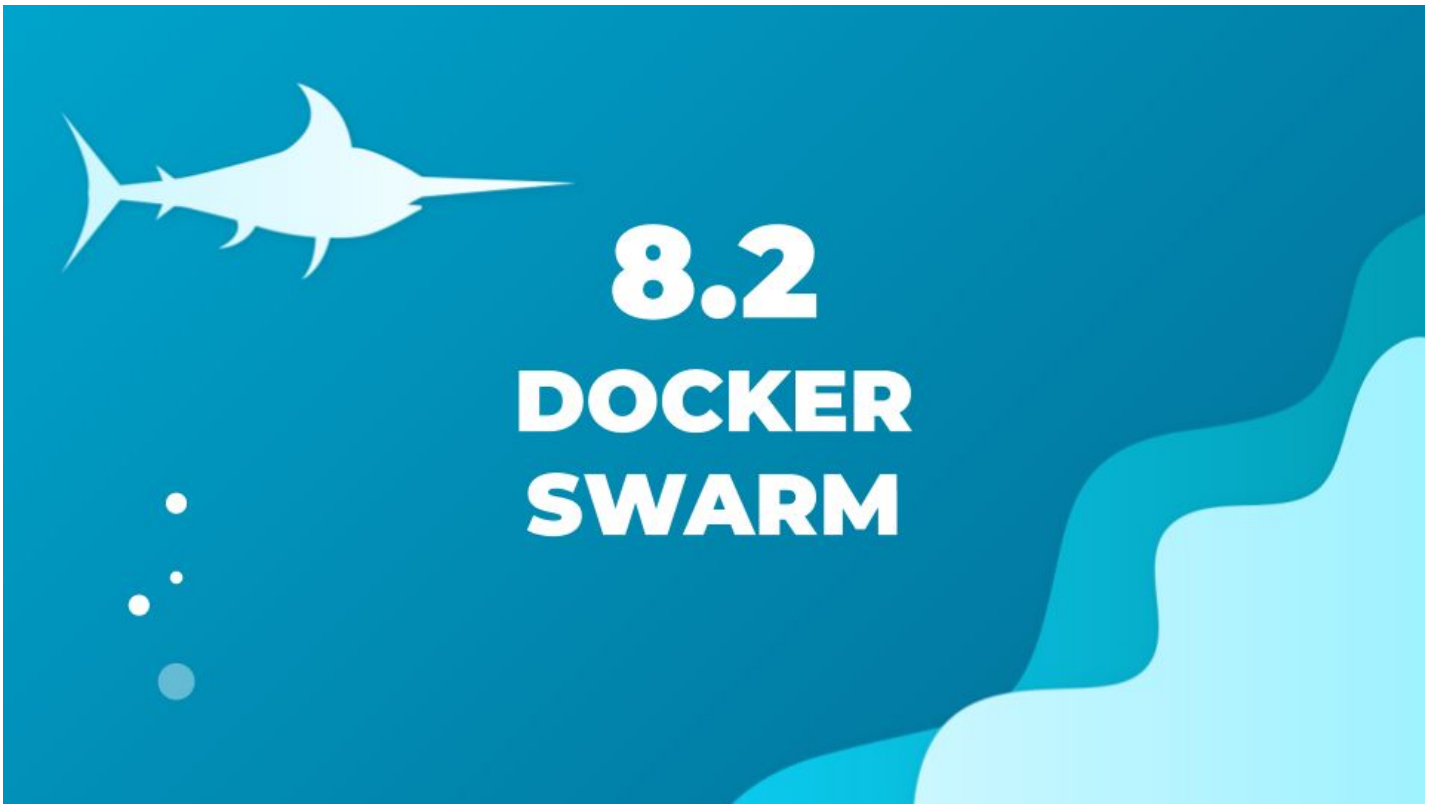
В случае с MESOS его намного сложнее установить и настроить, но он предлагает много продвинутого функционала.

Kubernetes из этих трех самый популярный:

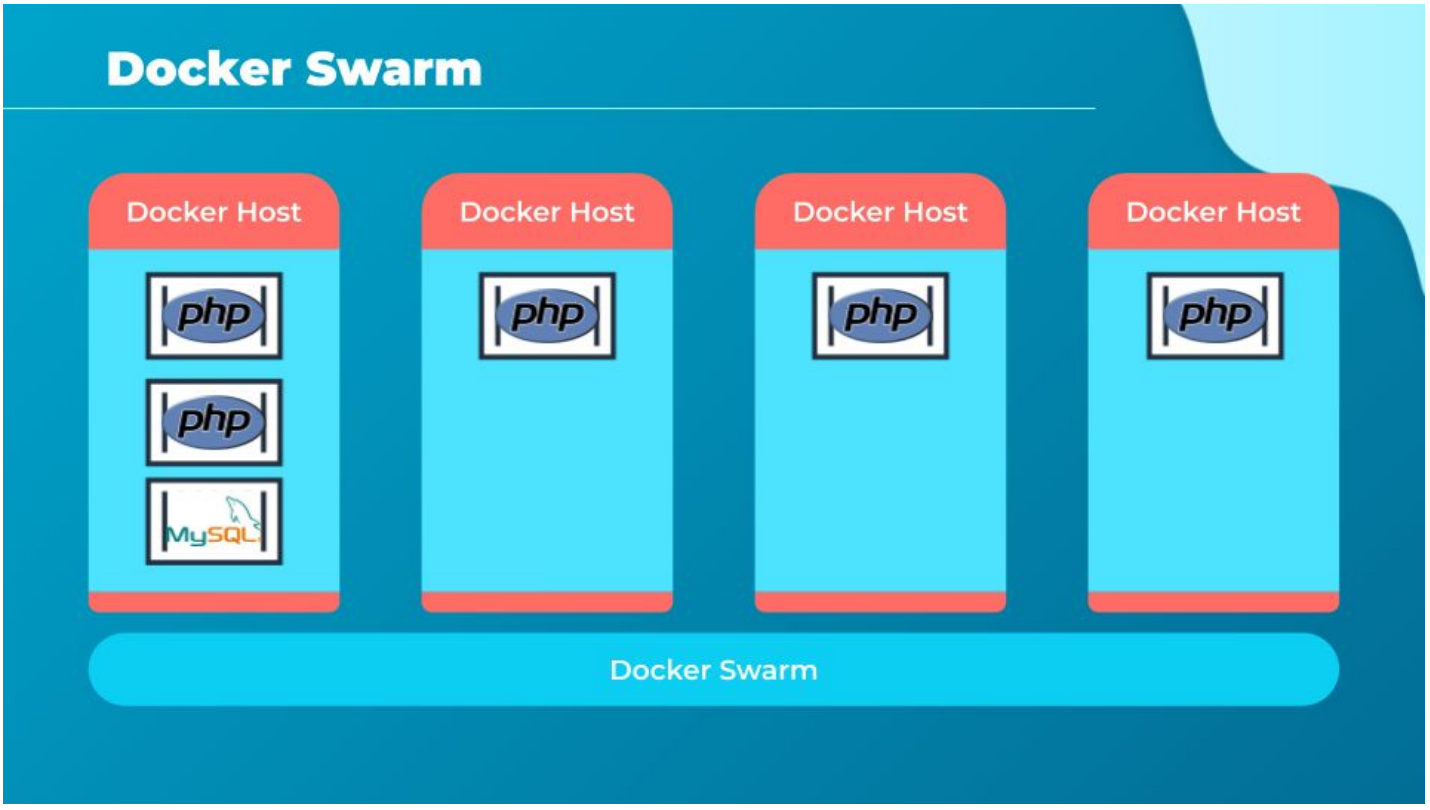
- **он не такой сложный в настройке, с ним можно быстро начать работу.**
- **он дает большое количество вариантов развертывания и поддерживает сложные архитектуры**

Kubernetes в данный момент поддерживается всеми клауд-вендорами, такими как GCP, Azure, AWS, Alibaba, а проект Kubernetes один из популярнейших на гитхаб.

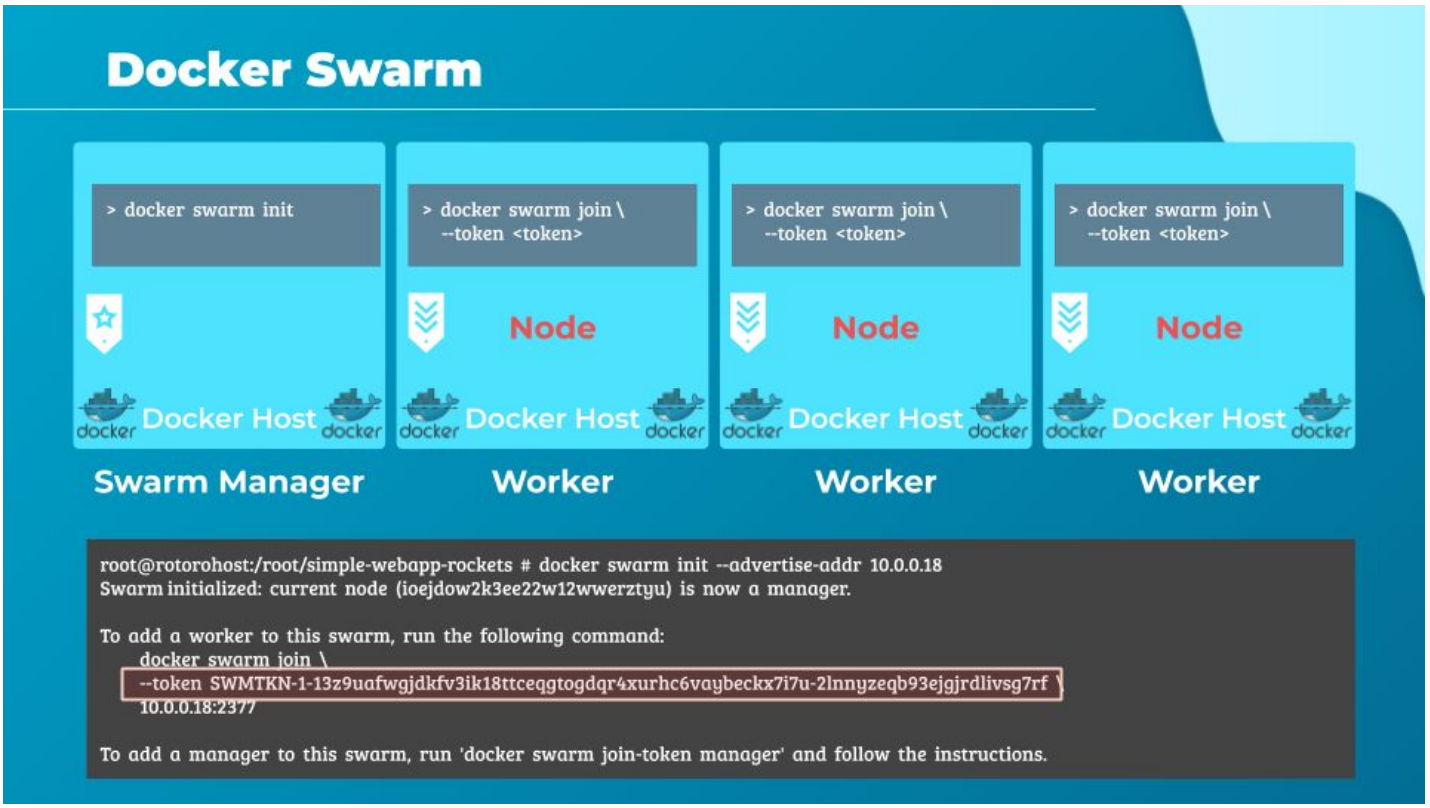
Это все в этой лекции, в следующих лекциях мы отдельно поговорим о Docker Swarm и Kubernetes.



Привет, в этой небольшой лекции я немного познакомлю тебя с Docker Swarm. Docker Swarm включает в себя много концепции и потребует своего собственного курса. Здесь я покажу основные детали, как говориться "в крупную клетку". После этого ты будешь понимать о чем речь, и, когда тебе понадобиться эта технология будешь знать, куда смотреть.



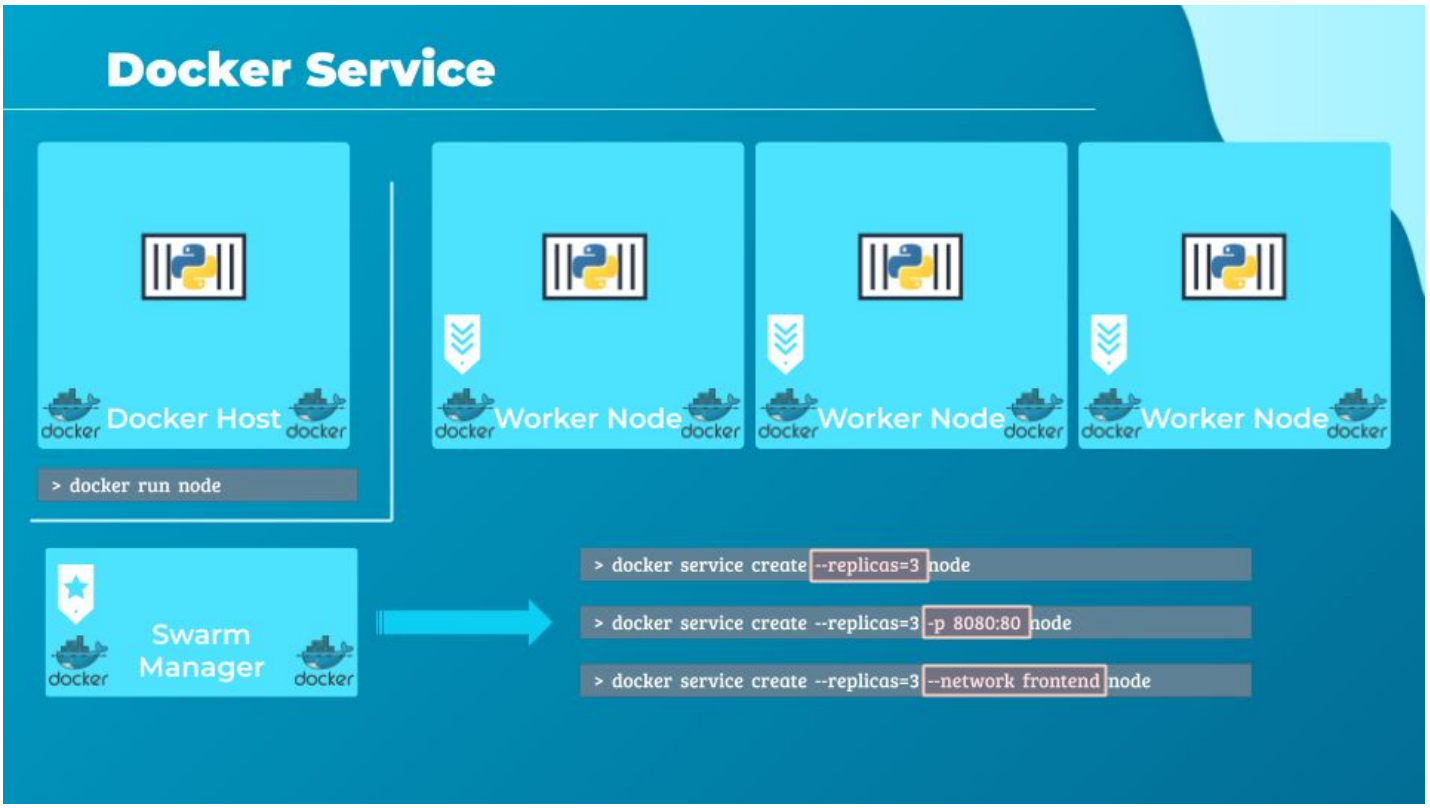
С Docker Swarm ты можешь собрать несколько докер-хостов в единый кластер, который будет согласованно работать над твоими задачами. Кластер займется распределением служб или экземпляров приложений на отдельные хосты, тем самым обеспечит высокую доступность и балансировку нагрузки между различными системами и оборудованием.



Для начала тебе потребуется одна или несколько машин с установленным на них Docker. После этого нужно определиться, какой из этих хостов будет использоваться в роли менеджера или мастера. Остальные хосты станут слейвами или воркерами.

Затем запустим команду `docker swarm init` на менеджере для инициализации менеджера swarm.

В выводе команды будет представлена команда, которую нужно запустить на воркерах, для того, чтобы рабочие экземпляры swarm смогли присоединиться к менеджеру в кластере. После присоединения к swarm рабочие хосты станут называться нодами, и теперь мы готовы создавать службы и развертывать их в кластере swarm.



Давай рассмотрим некоторые подробности. Как ты помнишь ранее, чтобы запустить экземпляр моего веб-сервера мне требовалось запустить команду `docker run` и указать имя образа, который я хотел запустить. Это создаст новый контейнер с экземпляром моего приложения и в нем мой сервер начнет обслуживать клиентов.

Мы уже узнали, как создать кластер swarm. А как использовать кластер для запуска нескольких экземпляров моего веб-сервера?

Один из способов добиться этого - запустить команду `docker run` на каждой воркер ноде кластера. В случае пары узлов это звучит приемлемо, но если нод сотни?

Т.е. мне придется придется входить на каждую из сотни нод и запускать эту команду, для этого мне нужно будет где-то держать все учетные записи или ключи для входа. Также мне придется придумать решение для балансировки нагрузки, а для этого мне потребуется также решить вопрос с мониторингом состояния узла в части его загруженности. И еще мне потребуется проверять, как чувствует себя контейнер, и живо ли мое приложение в нем, а там где произойдет сбой, мне нужно будет перезапустить контейнер самостоятельно.

Такая миссия очень быстро становится невыполнимой. Процессы можно улучшить самодельной автоматизацией, но ее приходится поддерживать и со временем это превращается в уродливого монстра. И именно здесь оркестрация Docker swarm решит эти задачи за нас.

Пока мы только создали этот кластер, но не видели оркестрацию в действии. Ключевым компонентом оркестрации swarm является service. Service в Docker - это один или несколько экземпляров одного приложения или службы, которые работают на нодах в кластере.

Например, я создам Docker service чтобы запустить несколько экземпляров моего веб-сервера на нескольких нодах в моем swarm кластере. Для этого запущу команду `docker service create` на менеджере swarm. Я укажу имя моего образа в этой команде. А использовав параметр --replicas задам количество экземпляров этого приложения, которые мне нужно развернуть в моем кластере.

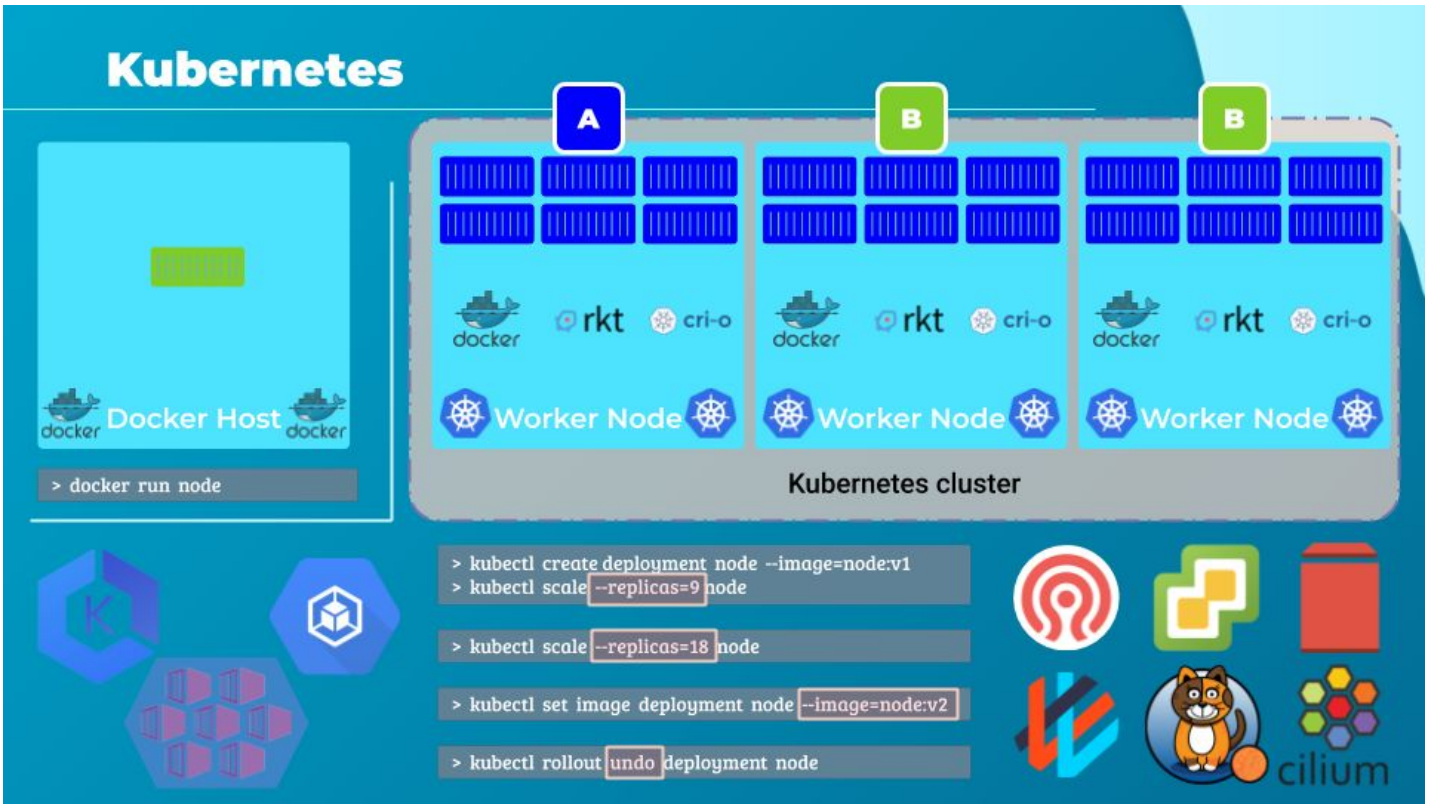
Поскольку я указал три реплики, я получу три экземпляра моего веб-сервера, распределенных по разным рабочим узлам.

Помни, что команда `docker service` должна запускаться на ноде-менеджере, а не на рабочем узле. Команда `docker service create` аналогична команде `docker run` в случае отдельного докер-хоста. То же самое и в отношении уже знакомых параметров, таких как -e (переменные окружения), параметра -p для публикации портов, опции --network для подключения контейнера к сети и т. д.

Что ж, это было введение в Docker Swarm на высоком уровне. Тут есть еще много чего, что стоит узнать, например, настройка нескольких менеджеров, оверлейные сети и прочее. Как я и говорил, нюансы потребуют отдельного курса. Это все в этой лекции. Увидимся в следующей!

Привет, в этой лекции мы делаем краткое введение в базовые концепции Kubernetes. Kubernetes требует своего собственного курса, даже не одного, а целых четырех. Сейчас мы постараемся кратко познакомиться с ним, на том уровне, чтобы не пугаться при беседе слова Kubernetes.

Итак, с помощью Docker мы смогли запустить один экземпляр приложения, введя в докер-cli команду `docker run`, которая запустила контейнер. И теперь наша жизнь изменилась, ведь прежде развертывание приложений никогда не было таким простым.



С Kubernetes, использующим Kubernetes-cli, известный как утилита управления kube control, мы можем запустить нужное нам количество экземпляров одного и того же приложения с помощью всего одной команды.

Kubernetes может легко отмасштабировать количество реплик приложения до нужных нам значений с помощью другой команды.

Kubernetes можно даже настроить на автоматическое выполнение этой операции, чтобы экземпляры и сама инфраструктура могли масштабироваться вверх и вниз, в зависимости от пользовательской нагрузки.

Kubernetes может обновлять тысячи экземпляров приложения в режиме последовательного обновления по одному, группами или все разом с помощью всего одной команды.

Если что-то пойдет не так, он может откатить все эти развертывания с помощью одной команды и вернуться на предыдущую рабочую версию.

Kubernetes может помочь нам устроить A-B тесты для новых фич нашего приложения, обновив только нужный процент экземпляров, благодаря возможности указания количества реплик для каждой версии.

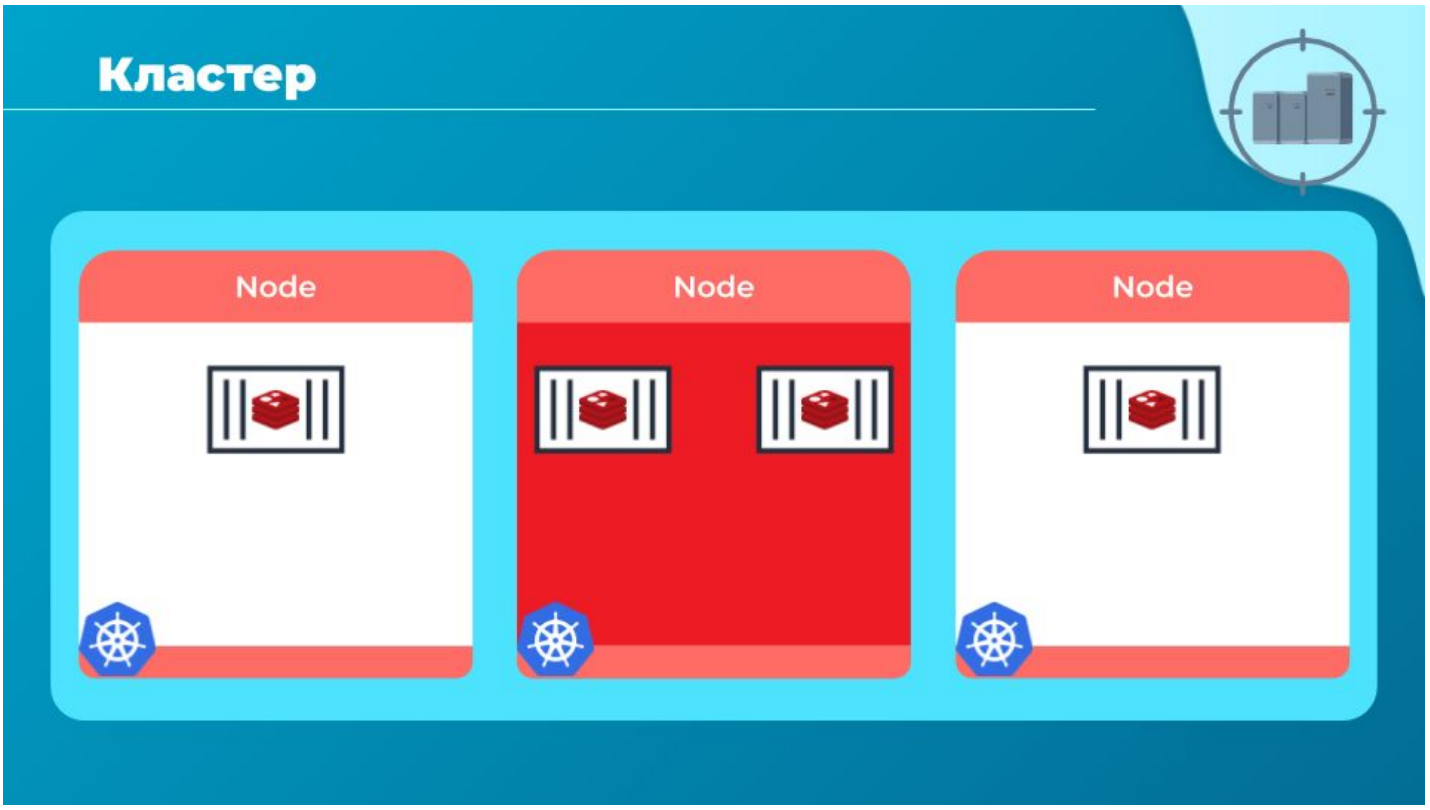
Открытая архитектура Kubernetes обеспечивает поддержку множества различных сетей и хранилищ. Любая торговая марка сетей или хранилища, о которой ты можешь подумать, имеет плагин для Kubernetes. Kubernetes поддерживает различные механизмы аутентификации и авторизации. Все основные поставщики облачных сервисов имеют встроенную поддержку Kubernetes.

Итак, какова связь между Docker и Kubernetes? Kubernetes использует докер-хосты для размещения приложений в виде контейнеров Docker. Но это не обязательно должен быть Docker все время, Kubernetes поддерживает в качестве альтернативы Dockers, например, Rkt или Cri-o. Но по факту на rkt редко где встретишь, и этот проект более не развивается. Так что в реальном мире

это или Docker, или Cri-o. И последний завоевывает позиции. Т.к. Docker многофункциональный инструмент, он не очень хорошо вписывается в роль только среды для запуска контейнеров, которую в нем видит Kubernetes. Cri-o, в свою очередь, имеет более оптимизированный и легковесный инструмент для запуска контейнеров и продвинутые возможностями для их траблешутинга.

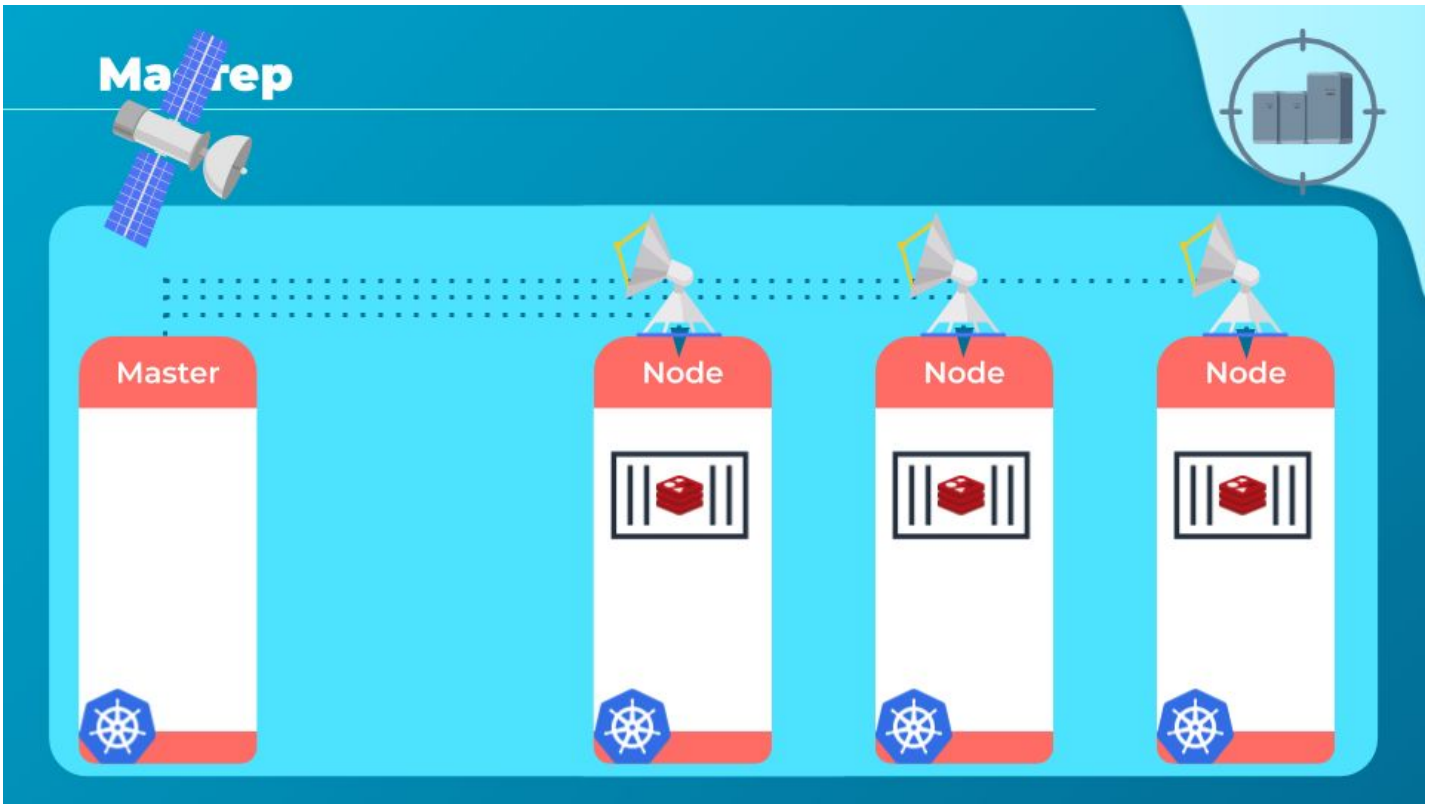
В момент написания статьи уже объявлено, что Kubernetes перестанет поддерживать Docker в качестве container runtime. Это не значит, что эра докер-контейнеров прошла, они как раз останутся, но похоже, что самого докера в Kubernetes со временем не останется.

Давай вернемся к основной теме и кратко рассмотрим архитектуру Kubernetes.
Кластер представляет из себя набор узлов - nodes.



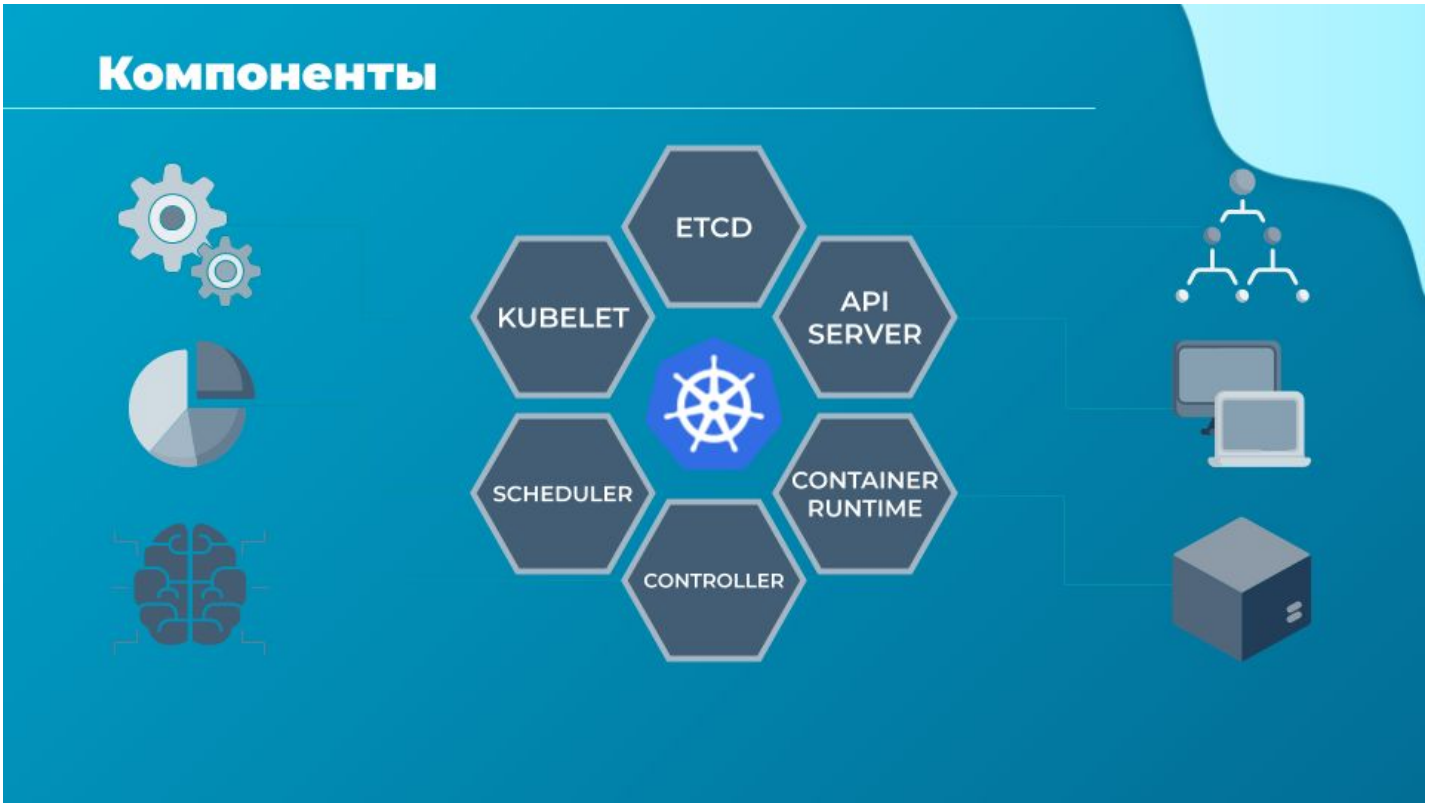
Нода это машина, физическая или виртуальная, на которой установлен Kubernetes. Воркер нода, это узел, на котором Kubernetes будет запускать контейнеры с полезной нагрузкой. Раньше они назывались миньонами. Услышав этот термин имей в виду, что это синонимы. Что случится, если нода выйдет из строя?

Очевидно, что наше приложение упадет. Выходит, нам нужно иметь больше одной ноды. Кластер - это набор сгруппированных вместе узлов. Таким образом, даже если один узел падает, наше приложение по-прежнему доступно для пользователей. Более того, наличие нескольких нод также помогает в распределении нагрузки.



Ок, теперь у нас есть кластер, но кто будет отвечать за его управление? Где будет храниться информация о членах кластера? Как нам узнать о событиях, происходящих на нодах? А когда нода выйдет из строя, как перенести рабочую нагрузку с упавшей ноды на исправную?

Вот здесь появляется мастер, или как теперь его политкорректно называют controlplane. Мастер это еще одна нода, член кластера Kubernetes, сконфигурированная как мастер. У нее особые функции: наблюдать за состоянием других нод и быть ответственной за оркестрацию контейнеров на воркер-нодах.



Когда ты устанавливаешь Kubernetes в систему, ты фактически устанавливаешь следующие компоненты:

- **API сервер**
- **Контейнер рантайм**
- **контроллеры и скедулеры**
- **службу kubelet**
- **службу ETCD**

API сервер выступает в качестве фронтенда, единого интерфейса для Kubernetes. Пользователи, устройства управления, интерфейсы командной строки, все общаются с API сервером для взаимодействия с кластером.

Среда выполнения контейнеров или контейнер-рантайм - это базовое программное обеспечение, которое используется для запуска контейнеров. В нашем случае это Docker, но есть и другие варианты.

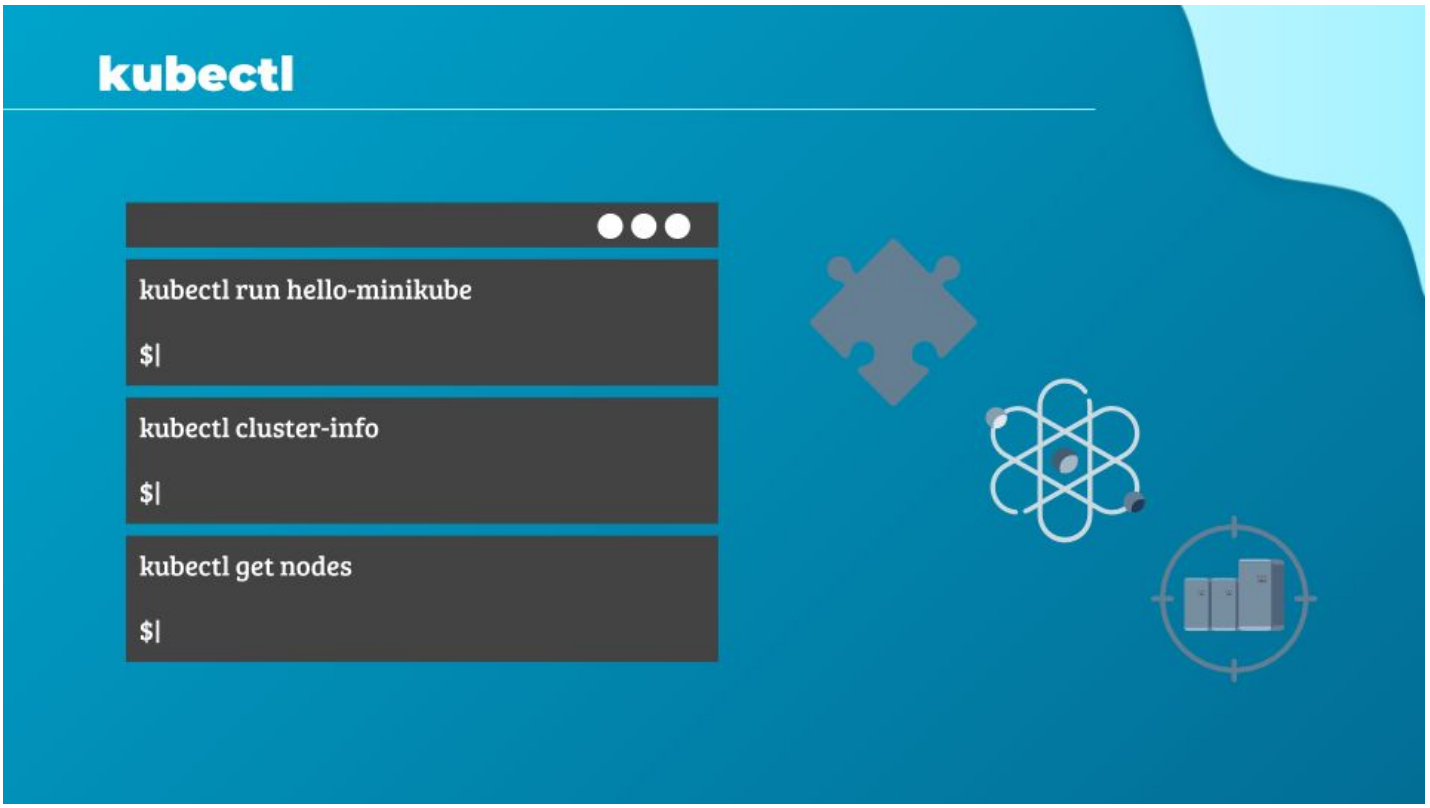
Контроллеры - это мозг оркестрации. Они смотрят за состоянием нод, контейнеров, эндпоинтов и ответственны за реакцию на события на нодах. Контроллеры принимают решения о создании новых контейнеров.

Скедулер ответственен за распределение работ или контейнеров между нодами. Он ожидает, когда в системе появится новый контейнер, чтобы назначить его выполнение на ноду.

Далее kubelet - это агент, который работает на каждом узле кластера. Агент отвечает за то, чтобы контейнеры работали на узлах должным образом.

И, наконец, ETCD - хранилище "ключ-значение". ETCD - это распределенная, надежная база данных, используемая Kubernetes для хранения всей информации нужной для управления кластером.

И в конце нам нужно немного узнать об утилите командной строки kubectl или kube control, как ее еще называют. Инструмент kubectl используется для развертывания и управления приложениями в кластере Kubernetes, для получения информации о кластере, получения состояния других нод в кластере и управления другими вещами.



Команда kubectl run используется для развертывания приложения в кластере.

Команда kubectl cluster-info используется для просмотра информации о кластере.

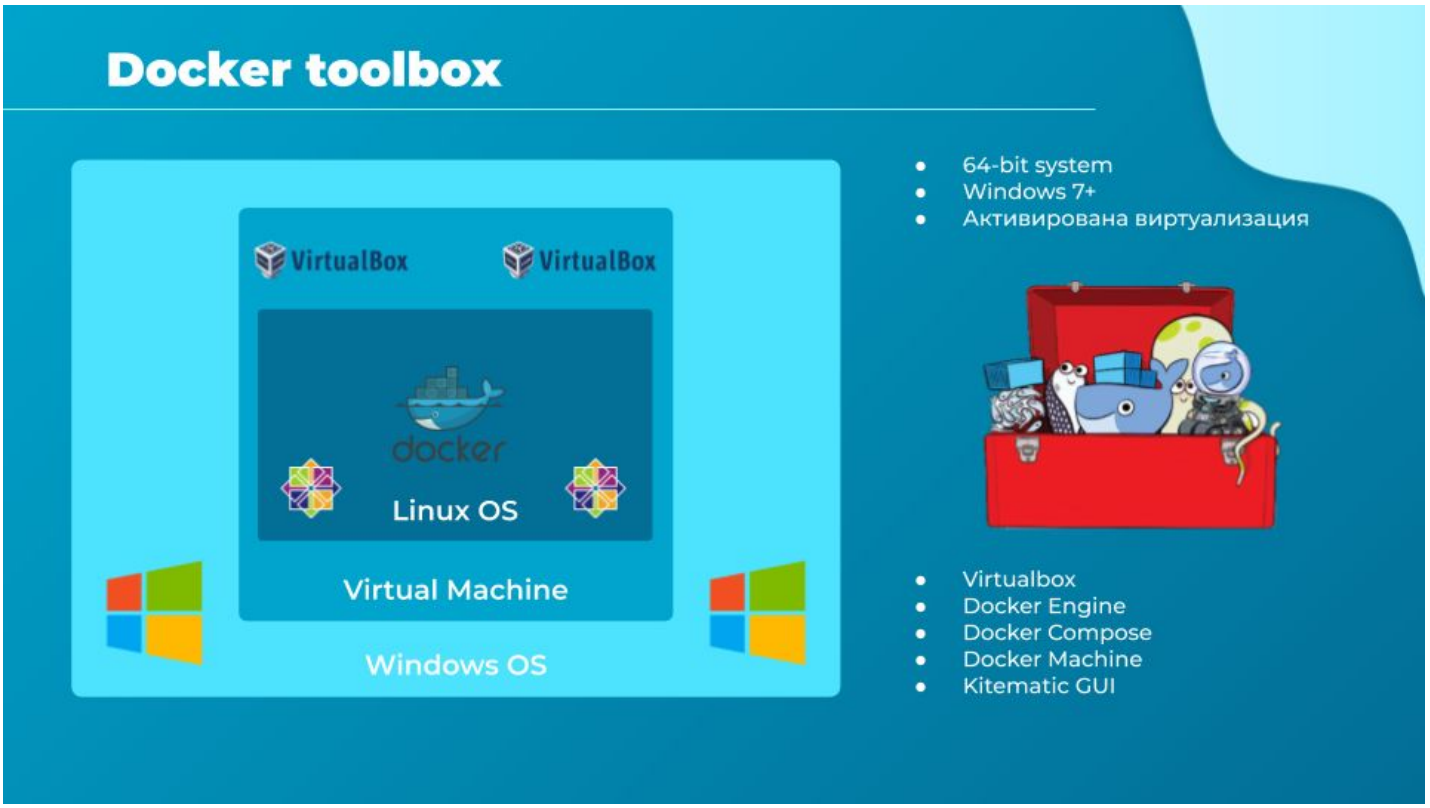
Команда kubectl get nodes используется для вывода списка всех узлов в кластере.

Как я говорил, это мощный инструмент и одной командой можно внести изменения в тысячи копий приложения на сотнях узлов.

Это все, что нам нужно знать из команд на данный момент по поводу понимания самой сути Kubernetes и его архитектуры. Если хочешь узнать больше, присоединяйся к моему курсу "Kubernetes для самых маленьких", а в 21 году я планирую выпустить курс по сертификации Kubernetes, этот курс уже экспертного уровня. Это все в этой лекции, жду тебя в заключении!

Добро пожаловать в приложение, здесь мы поговорим о Docker в Windows. Ранее в этом курсе мы узнали, что контейнеры совместно используют ядро операционной системы. Из этого следует, что у нас не может быть контейнера Windows, работающего на хосте Linux, и наоборот. Мы должны помнить об этом, начиная эту лекцию, так как это очень важная концепция. Как правило у большинства новичков возникают проблемы с этим.

Итак, какие варианты доступны для Docker в Windows? Это два решения. Первое - это Docker с использованием Docker Toolbox, а второе - вариант Docker Desktop for Windows.
Сейчас мы рассмотрим каждый из них.



Давай посмотрим на первый вариант Docker Toolbox. Это была одна из первых попыток реализовать Docker в Windows.

Представь, что у тебя есть ноутбук с Windows и нет доступа к какой-либо системе Linux, а ты хочешь попробовать Docker. Также у тебя нет доступа к системе Linux по сети или в облаке. Что бы ты сделал?

То, что я бы сделал, - это установил программное обеспечение виртуализации в моей системе Windows, такое как Oracle Vitualbox или Vmware workstation, и развернул на нем виртуальную машину Linux. Такую как Unbuntu или Debian, затем установил Docker на виртуальную машину Linux, и после этого смог бы поиграть с ним. На самом деле это не имеет ничего общего с Windows. Я бы не смог создавать образы Docker для Windows или запускать контейнеры Docker с Windows-приложениями. Ок, и очевидно, что я не смогу запустить контейнеры Linux непосредственно в Windows, т.к. я просто работаю с Docker на виртуальной машине Linux на хосте Windows.

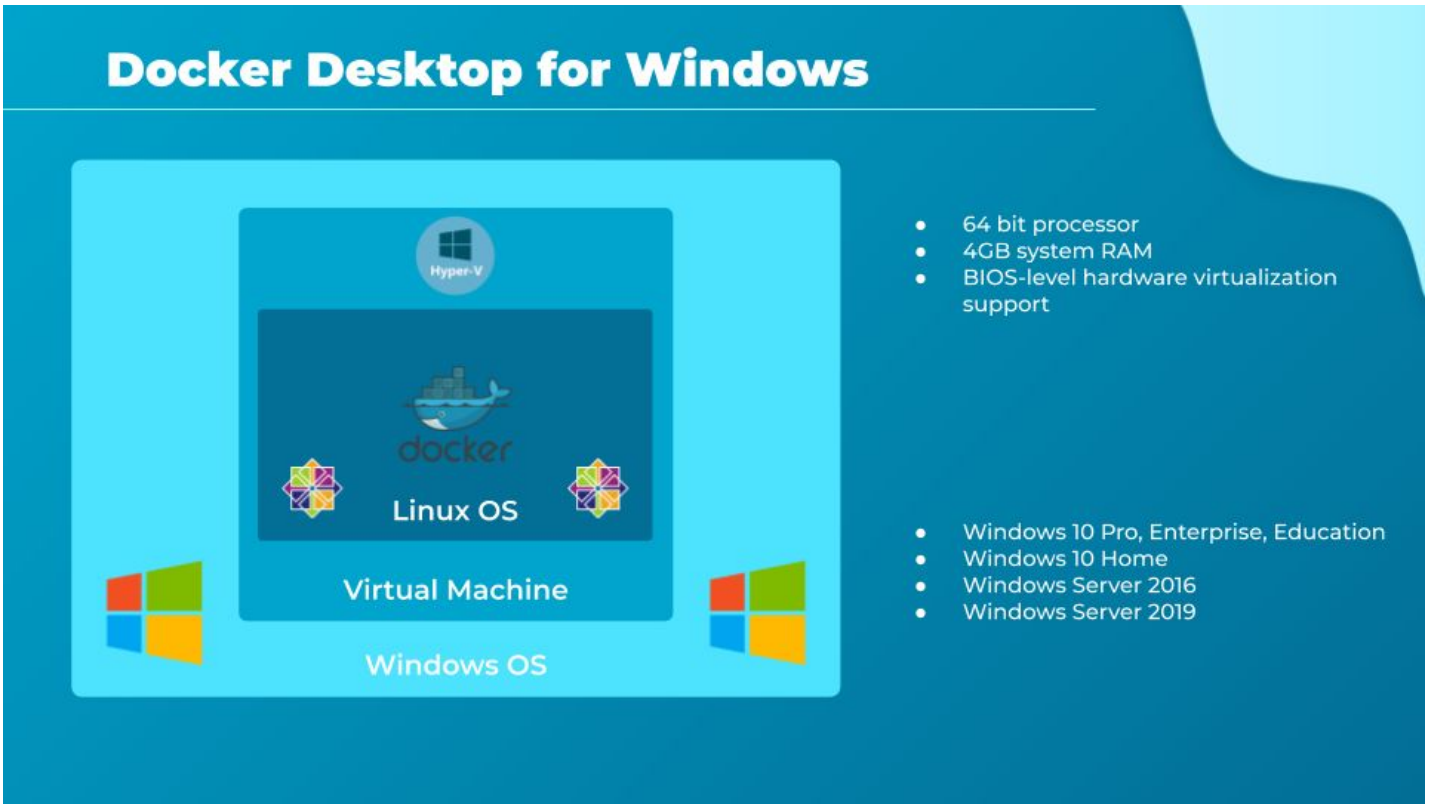
Docker, однако предоставляет нам набор инструментов для упрощения этого процесса, который называется Docker Toolbox. Docker Toolbox содержит в себе утилиты, такие как Oracle VirtualBox, Docker Engine, Docker Compose, Docker Machine и пользовательский интерфейс под названием Katematic.

Это поможет тебе быстро начать работу, просто загрузив и запустив исполняемый файл установщика Docker Toolbox, котрый проинсталлирует легковесную виртуальную машину, в которой уже настроен Docker. И теперь все будет готово, чтобы начать работу с Docker, и это действительно занимает немного времени.

Теперь о требованиях. Нужно убедиться, что твоя операционная система с разрядностью 64-бита, это Windows 7 или выше, и что в системе включена виртуализация.

Из минусов этого подхода - высокий overhead из-за виртуальной машины, а также возможность работать только с Linux-приложениями.

Также имей в виду, что Docker Toolbox - это устаревшее решение и используется для систем Windows, которые не соответствуют требованиям для нового варианта Docker для Windows.

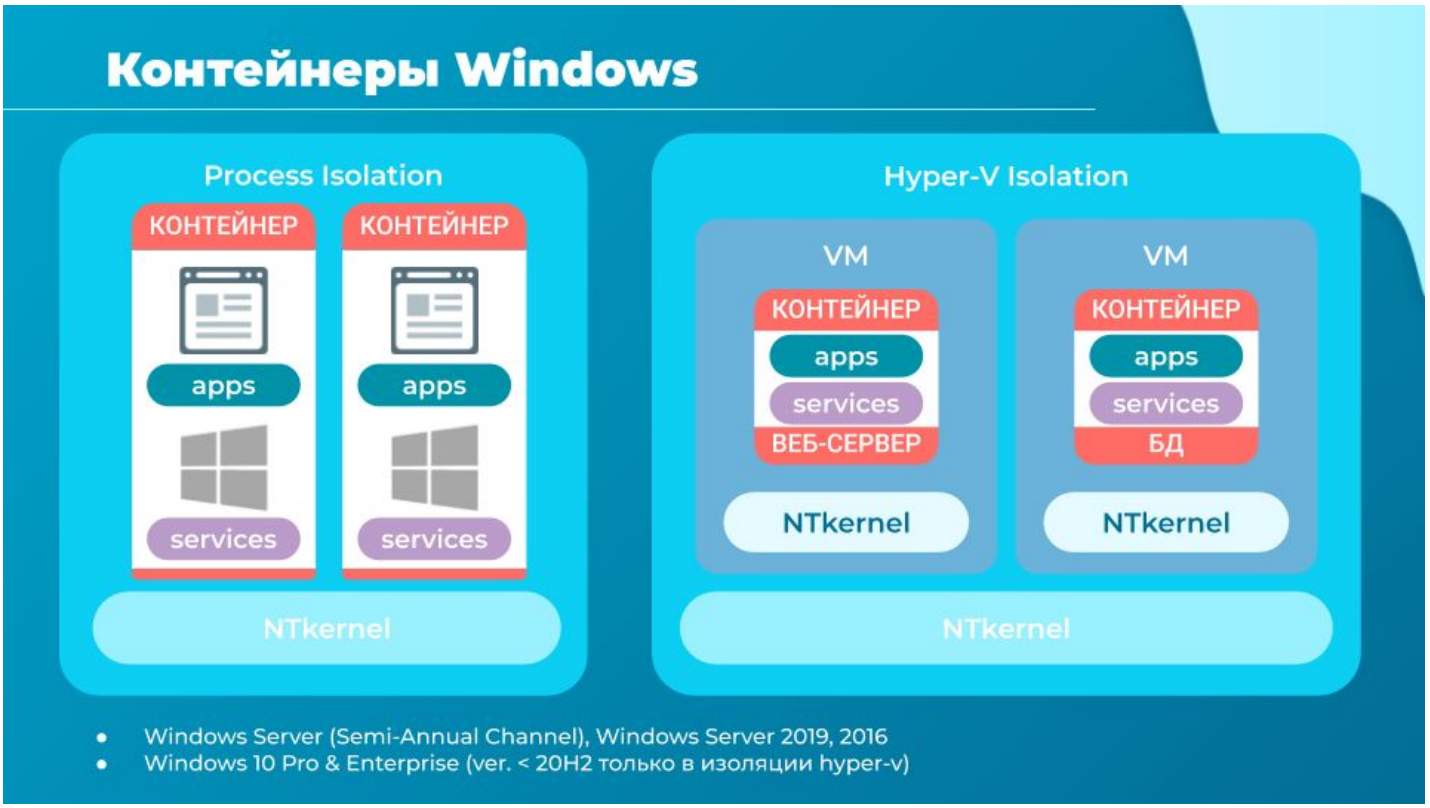


Второй вариант - это более новое решение, называется Docker Desktop for Windows. Для Windows в предыдущем варианте мы видели, что у нас был установлен Oracle VirtualBox в Windows, затем ОС Linux, а затем в Docker в этом Linux. С Docker Desktop мы по сути извлекаем VirtualBox и используем встроенную технологию виртуализации, доступную в Windows, под названием Microsoft Hyper-V.

В процессе установки Docker для Windows по-прежнему будет автоматически создаваться система Linux внутри, но на этот раз он будет создан на Microsoft Hyper-V вместо Oracle VirtualBox, и Docker будет запущен уже с зависимостью от Hyper-V. Эта возможность поддерживается только для Windows 10 и Windows Server 2016 и старше, поскольку эти операционные системы по умолчанию поддерживают Hyper-V.

Теперь самый важный момент, который мы обсуждали с поддержкой Docker для Windows.
Все сказанное ранее доступно строго для контейнеров Linux. Это приложения Linux и они упакованы в образы Linux Docker. Мы еще не говорили о приложениях Windows, образах Windows или контейнерах Windows. Контейнеризация Windows-приложений тоже возможна.

Ок, подытожим, оба варианта, которые мы только что обсудили. Они оба могут запустить контейнер Linux на хосте Windows. Первый для legacy, второй более производительный.



С Windows Server 2016 появилась возможность поддержки контейнеров Windows. Теперь мы можем упаковывать приложения Windows в докер-контейнеры Windows и запускать их на докер-хосте под Windows с использованием Docker Desktop.

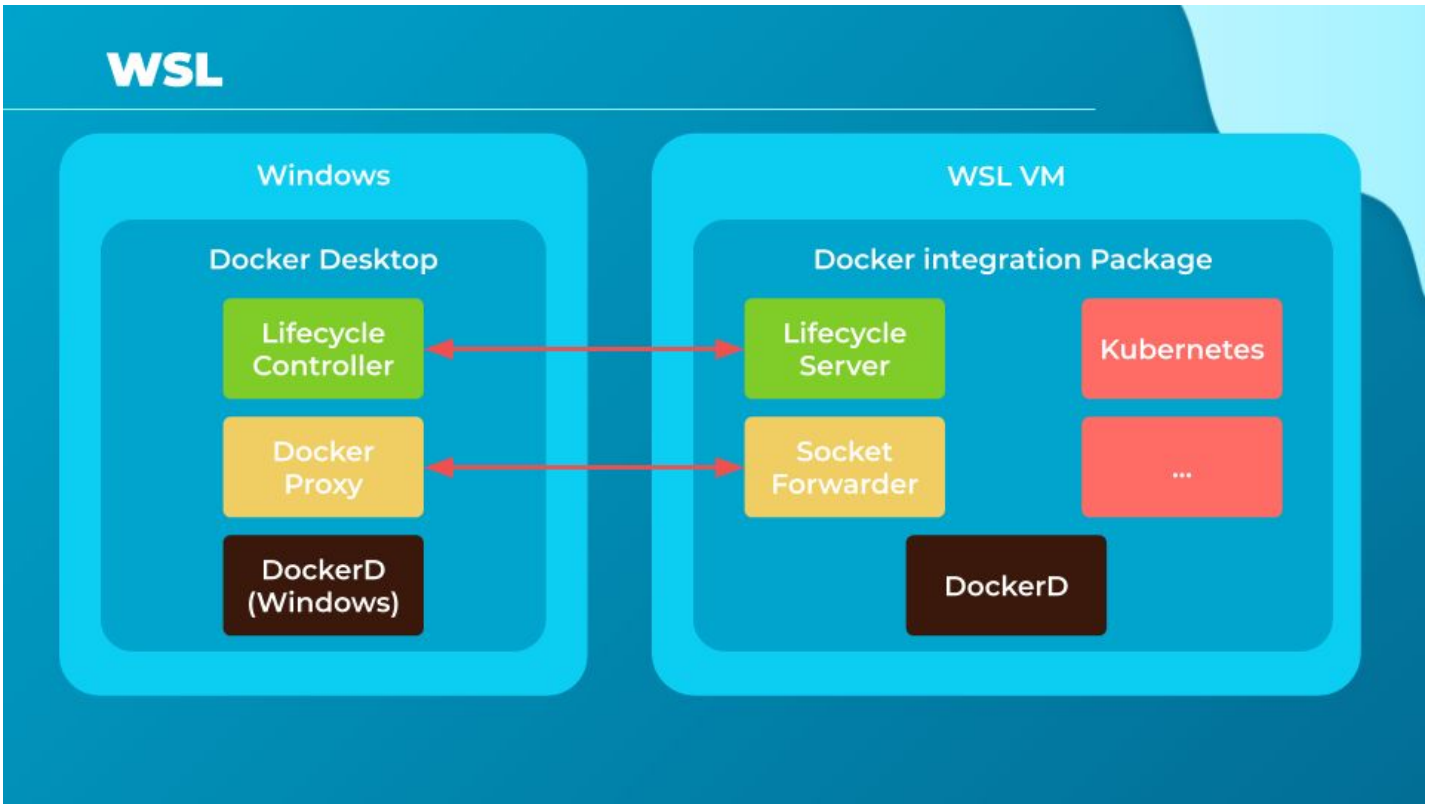
Т.е. мы можем создавать образы на основе Windows и запускать контейнеры Windows на сервере Windows точно так же, как если бы запускали контейнеры Linux в системе Linux. А следовательно, мы можем создавать образы Windows для контейнерных приложений и делиться ими через Docker hub, как и в случае с Linux.

Но, в отличии от мира Linux, в Windows есть два типа контейнеров:

Первый - это контейнер Windows, который работает точно так же, как контейнеры Linux, где ядро ОС (в данном случае NTkernel) используется совместно с базовой операционной системой, чтобы обеспечить лучшую границу безопасности и позволить сосуществовать разным версиям окружения и конфигураций приложения. Это называется Process isolation (ранее Windows server).

Также есть второй вариант, известный как Hyper-V isolation. С изоляцией Hyper-V, каждый контейнер запускается на высоко оптимизированной виртуальной машине, гарантирующей полную изоляцию ядра между контейнерами и базовым хостом

И наконец, контейнеры Windows поддерживаются на Windows Server 2016,2019, Nanoserver, Windows 10 Professional и Enterprise Edition.



Развитие контейнеризации в Windows идет достаточно быстро: пять лет назад мы могли запускать только Linux-контейнеры в сторонней виртуализации.

С появлением WLS (Windows Subsystem for Linux), стало возможно выбирать, какой тип контейнеров будет запускать Docker Desktop. Тем не менее, были ограничения - невозможно было запускать контейнеры сразу двух типов одновременно.

В 2020 году вышла WSL2, это полнофункциональное ядро Linux со всеми системными вызовами, у которой увеличена производительность файловой системы. Теперь все контейнеры могут выполнятся как бы в одном поле и управляться из одного места. Хочу отметить, что все это бывает не так просто в настройке, а контейнеры Windows до сих пор работают достаточно медленно с volumes, т.к. для работы с volumes используется SMB протокол.

Также следует отметить, что примерно раз в полугодие выходят обновления, которые сильно модернизируют ОС, так например Windows 10 до конца 2018 года не могла запускать контейнеры в режиме Process isolation, а теперь эта возможность появилась, хоть и экспериментально.

В мире Linux у нас было несколько базовых образов для систем Linux, таких как Ubuntu, Debian, Fedora, Alpine и т. д. Если ты помнишь, это то, что мы указываем в начале Dockerfile. В мире Windows у нас есть несколько вариантов:

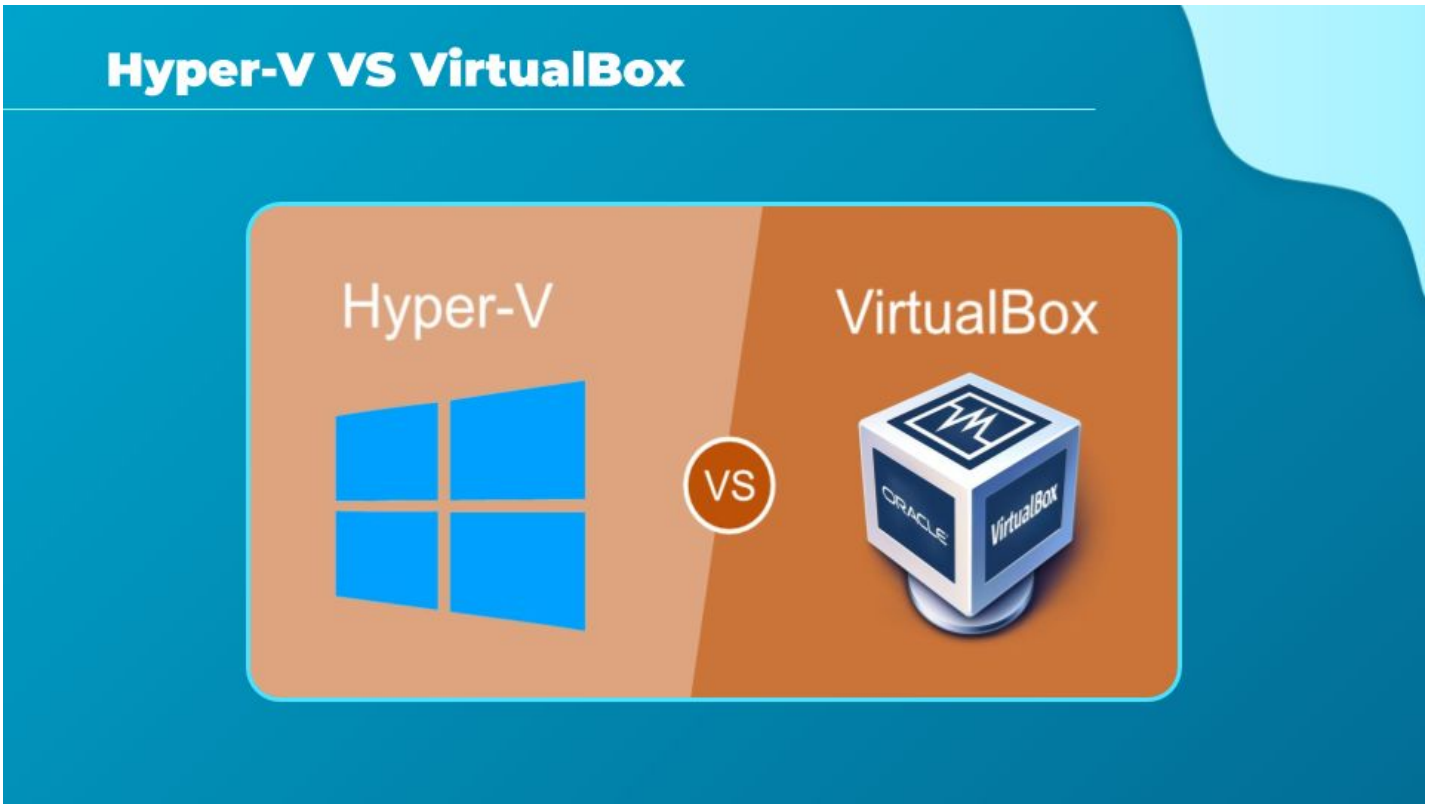
- **Nanoserver**
- **Windows Server Core**
- **Windows**
- **IoT Core**

Nanoserver - это вариант headless развертывания для Windows Server, который работает как не полноразмерная операционная система. Думай об этом, как об образе Alpine в Linux. Он предпочтителен для приложений на основе .NET Core.

Windows Server Core совсем не легкий, в нем больше API, есть PowerShell, инструменты WMI, полная версия .NET Framework.

Windows - самый жирный из всех, в нем есть все базовые библиотеки, включая GDI.

IoT Core - небольшой образ, оптимизированный под задачи автоматики и интернета вещей.



Ну вот и все о Docker для Windows. Эта тема очень интересна и быстро развивается, в ней много частных моментов, вроде работы Docker Swarm и Kubernetes с Windows-хостами.

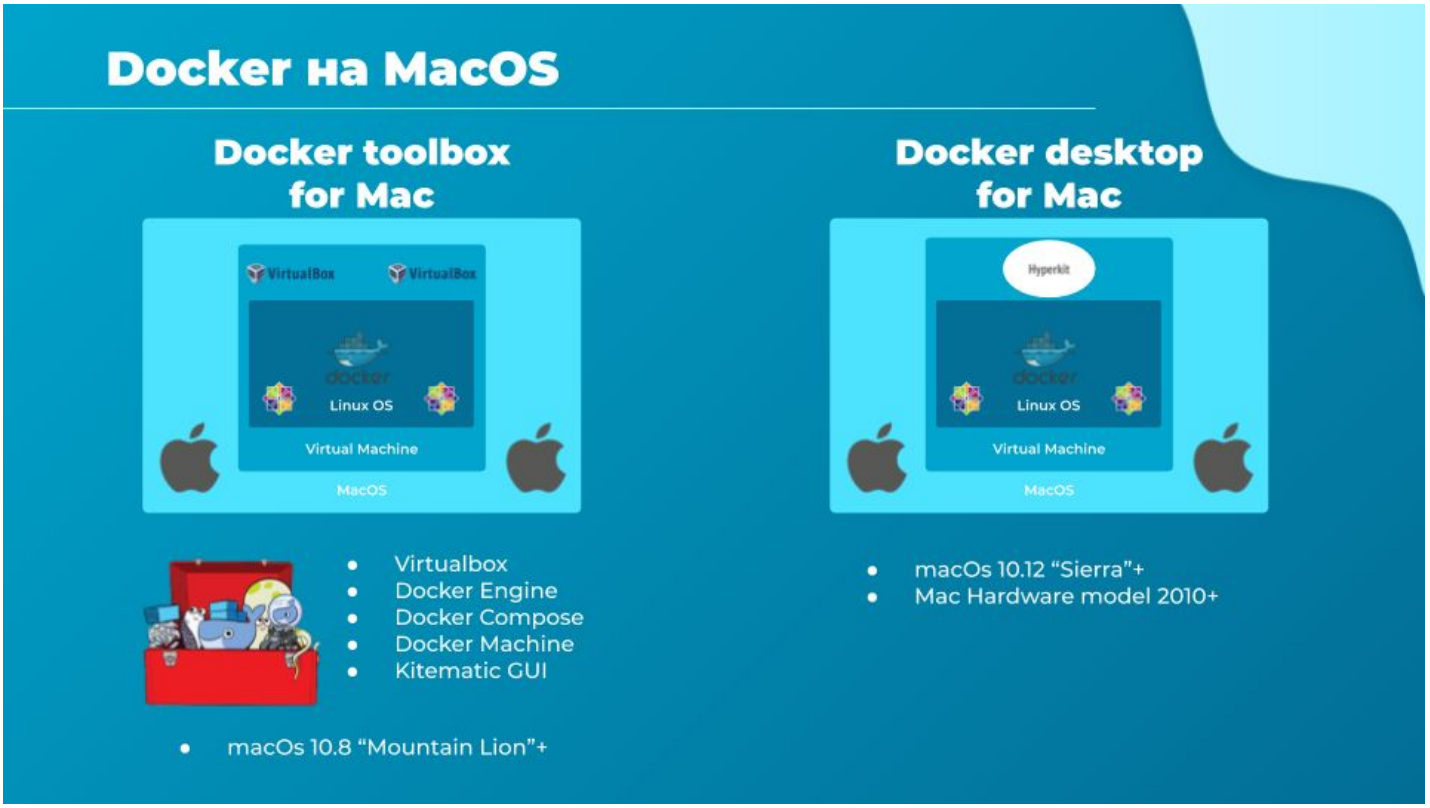
Теперь, прежде чем я закончу, я хочу указать на один важный факт, который мы видели с двух сторон - докер-контейнер на сторонней виртуализации (virtualbox) и нативной (Hyper-V) не могут сосуществовать на одном хосте Windows без головной боли.

Хотя и заявлено, что с версии 6+ Virtualbox может работать под Hyper-V, полной совместимости пока нет, и я не рекомендовал бы такое решение в продакшене. По своему опыту я не смог их подружить. Т.о. если ты начал в Docker Toolbox с Virtualbox и собираешься перейти на Hyper-V, то не получится использовать оба эти решения одновременно.

На странице документации Docker есть руководство по миграции и руководство о том, как перейти с Virtualbox на Hyper-V.

Это все в этой лекции, спасибо, и увидимся в следующей.

Привет, в этой короткой лекции мы поговорим про Docker на Mac.



История для Mac похожа на Docker в Windows. Есть два варианта начать работу с Docker на Mac:

- **с помощью Docker Toolbox**
- **с помощью Docker Desktop for Mac.**

Давай посмотрим на первый вариант с Docker Toolbox. Это была изначальная попытка запуска Docker на Mac. Представляет из себя Docker на Linux VM, созданный с использованием Virtualbox для Mac. В отличие от Windows, в MacOS нет совместного использования ядра. Поэтому не получится создать образ или контейнер с приложениями Mac. В MacOS возможно только запускать контейнеры Linux.

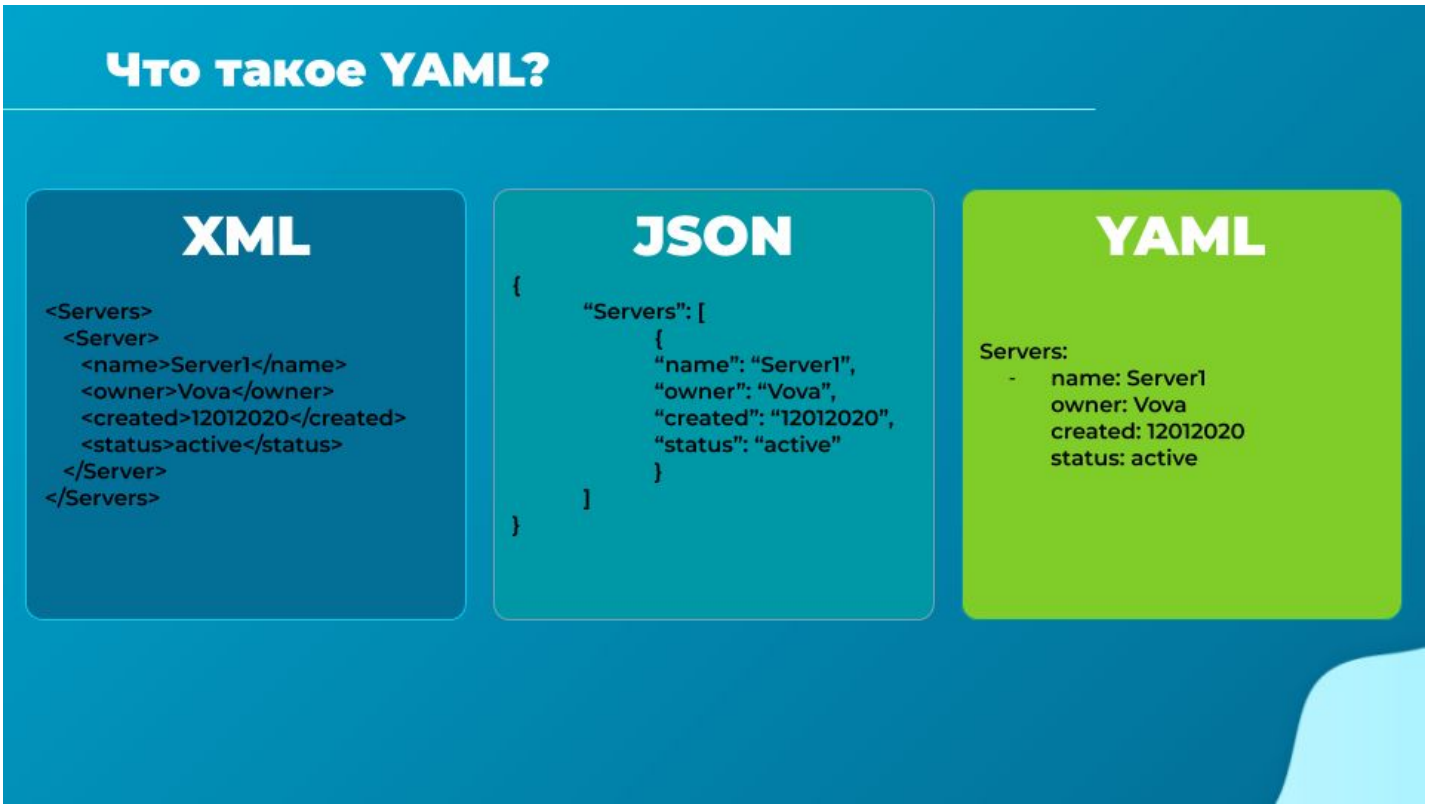
Docker Toolbox содержит в себе, такие утилиты, как Oracle VirtualBox, Docker Engine, Docker Compose, Docker Machine и пользовательский интерфейс под названием Katematic. Это развертывает легкую виртуальную машину, в которой уже настроен Docker. Для этого требуется MacOS 10.8 или новее. Этот способ считается устаревшим.

Второй вариант - это более современное решение, называется Docker Desktop for Mac. Здесь вместо VirtualBox мы используем технологию виртуализации HyperKit. Он по-прежнему будет автоматически создавать систему Linux внизу, но на этот раз в HyperKit, а не в Oracle VirtualBox. Далее в этой системе будет запущен Docker. Для этого требуется MacOS sierra 10.12 или новее, и, на мой взгляд, оборудование Mac должно быть модели 2010 года или новее.

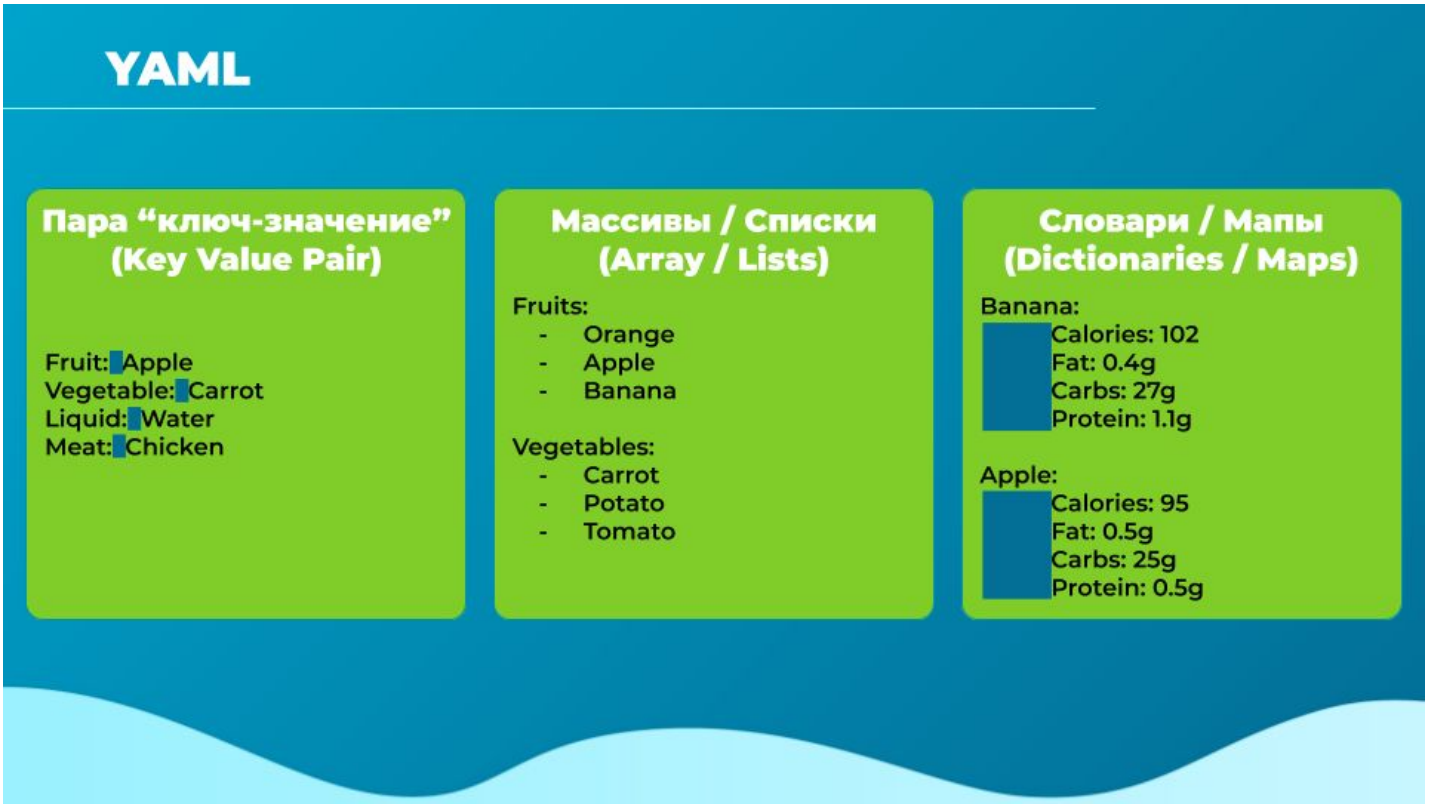
Это все, что необходимо для запуска контейнера Linux на Mac. На момент написания статьи об образах или контейнерах для Mac мне ничего не известно.

Увидимся в следующей лекции!

Привет и добро пожаловать на лекцию, где мы узнаем, что такое YAML файлы. Если ты знаком с YAML файлами, просто пропусти этот раздел. Но если ты не еще работал с YAML до этого времени, я сильно рекомендую тебе внимательно пройти эту лекцию и упражнения, потому что многое в курсе зависит от знания YAML. Если ты работал с другими форматами структурирования данных, такими как XML или JSON, ты легко все поймешь. Не волнуйся, если ты не работал ни с одним из них. Ты
также все легко поймешь, проработав навык в упражнениях, которые сопровождают курс. YAML файл используется для представления данных, в нашем случае для файлов конфигурации.



Здесь быстрое сравнение образцов одинаковых данных в трех разных форматах. Левый это XML, где мы представили список серверов и информацию о них. Те же данные представлены в JSON формате посередине. И, наконец, YAML формат справа. Посмотри внимательно на форматы и сравни их.



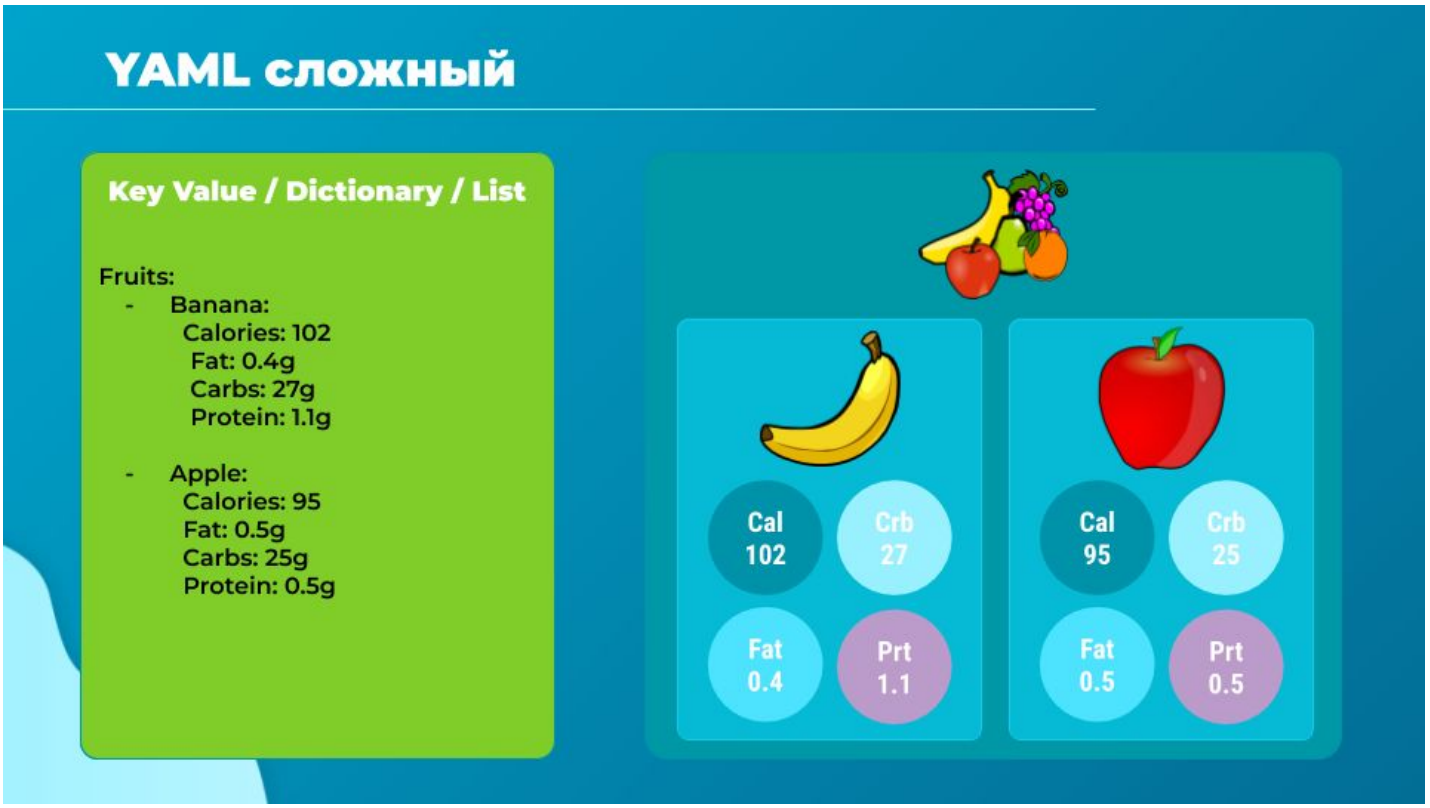
Давай взглянем ближе на YAML. Если взять данные в простейшей форме вроде "ключ-значение", вот как это выглядит в YAML. Ключ и значение разделено двоеточием. Ключи это: fruit, vegetable, liquid, и meat. Значения: apple, carrot, water, и chicken. Запомни, после двоеточия должен идти пробел, который разделит ключ и значение.

Теперь посмотрим, как представлен массив. Мы хотели бы перечислить несколько фруктов и овощей. Напишем fruits, за которым следует двоеточие. Далее в каждый элемент массива напишем с новой строки с тире впереди. Тире обозначает, что это элемент массива.

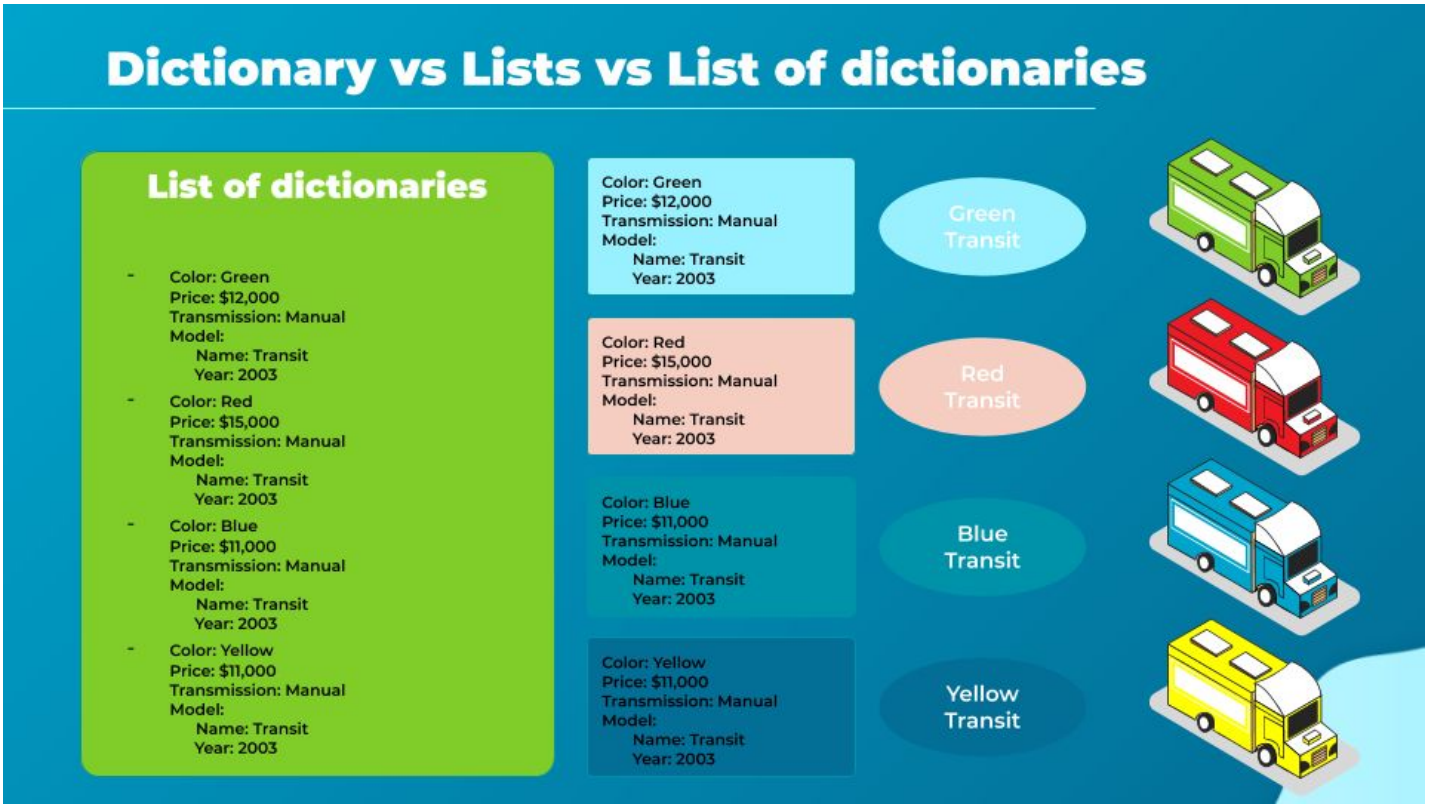
Что насчет ассоциативных массивов? Dictionary это набор свойств, сгруппированный вместе под одним элементом. Здесь мы попробуем представить пищевую ценность двух фруктов. Калории, жир, углеводы и белки отдельно для каждого фрукта. Обрати внимание на пустое место перед каждым элементом dictionary. Перед свойствами отдельного элемента должно быть равное количество пробелов, чтобы у них было одинаковое выравнивание.

Теперь взглянем внимательнее на пробелы в YAML. Здесь у нас dictionary, представляющий пищевую ценность банана. Показаны общее количество калорий, жира, углеводов и белка. Обрати внимание, количество пробелов перед каждым элементом определяет, что эти пары "ключ-значение" относятся к банану. Что произойдет, если добавить дополнительные пробелы для fat, carbs и protein?

Они попадут в категорию калорий, т.е. станут свойствами калорий, в чем нет никакого смысла. Еще это приведет к синтаксической ошибке, которая сообщит, что сопоставление значений здесь недопустимо, поскольку у ключа calories есть уже значение 102. Ты можешь присвоить ключу - значение, или установить его как название для низлежащего объекта, но только что-то одно. Количество пробелов перед каждым свойством является ключевым в YAML. Убедись в правильности расстановки пробелов, чтобы данные были представлены как предполагалось.



Усложним задачу. Давай создадим list, содержащий dictionaries. В этом случае у нас есть list фруктов - fruits. Его элементы банан и яблоко. Каждый из этих элементов представляет из себя dictionary, в котором содержится пищевая ценность.



Большинство новичков в YAML задают мне вопрос: что лучше использовать dictionary или list? Попробую объяснить по-понятнее. Во первых, и это важно понять, то, о чем мы говорим далеко от XML, JSON, или YAML, которые используются для представления данных. У нас могут быть данные организации и всех их сотрудников с персональными деталями, или данные школы со всеми ее студентами, их отметками, или данные автопроизводителя о выпущенных машинах и их деталях. Все что угодно.

Возьмем для примера автомобильный фургон. Машина это отдельный объект. Ее свойства это: цвет, цена, трансмиссия и модель. Для хранения разной информации или свойств объекта мы используем dictionary. В этом простом dictionary свойства машины представлены в формате "ключ-значение".

Все может стать сложнее, например нам нужно разделить свойство model на название модели и год выпуска. Это можно представить как dictionary внутри другого dictionary. Тогда значение поля model заменится небольшим dictionary с двумя свойствами name и year. Это dictionary в другом dictionary.

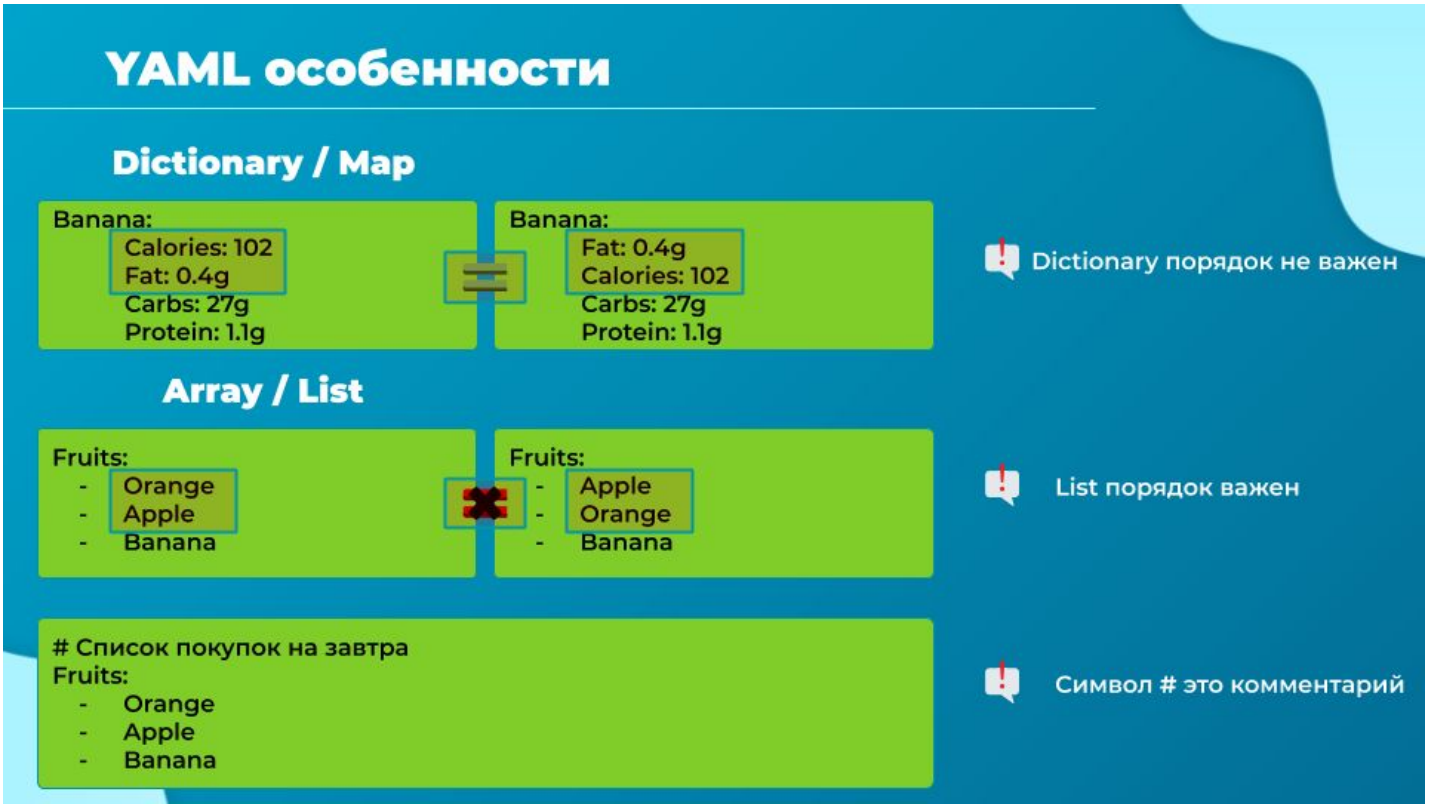
Теперь поговорим, как хранить имена четырех машин. Имена сформированы из цвета и модели фургона. Для хранения этого мы будем использовать list или array, т.к. у нас много элементов одного и того же вида. Мы храним только имена, и это простой массив строк.

Т.е. если информация однотипная, хорошо подойдет list, а если разнородная - dictionary.

А что, если мы захотим сохранить всю информацию о каждой машине? Все, что мы перечисляли раньше: цвет, цену, трансмиссию и модель. Нам нужно будет изменить list of strings на list of dictionaries. Мы расширяем каждый элемент в list и меняем имя на созданный ранее dictionary.

Таким образом, у нас получилось представить всю информацию о множестве фургонов в одном YAML файле используя list of dictionaries. В этом различие между list и list of dictionaries.

Надеюсь, различия стали очевидны и тебе стало проще понимать, когда какие использовать.



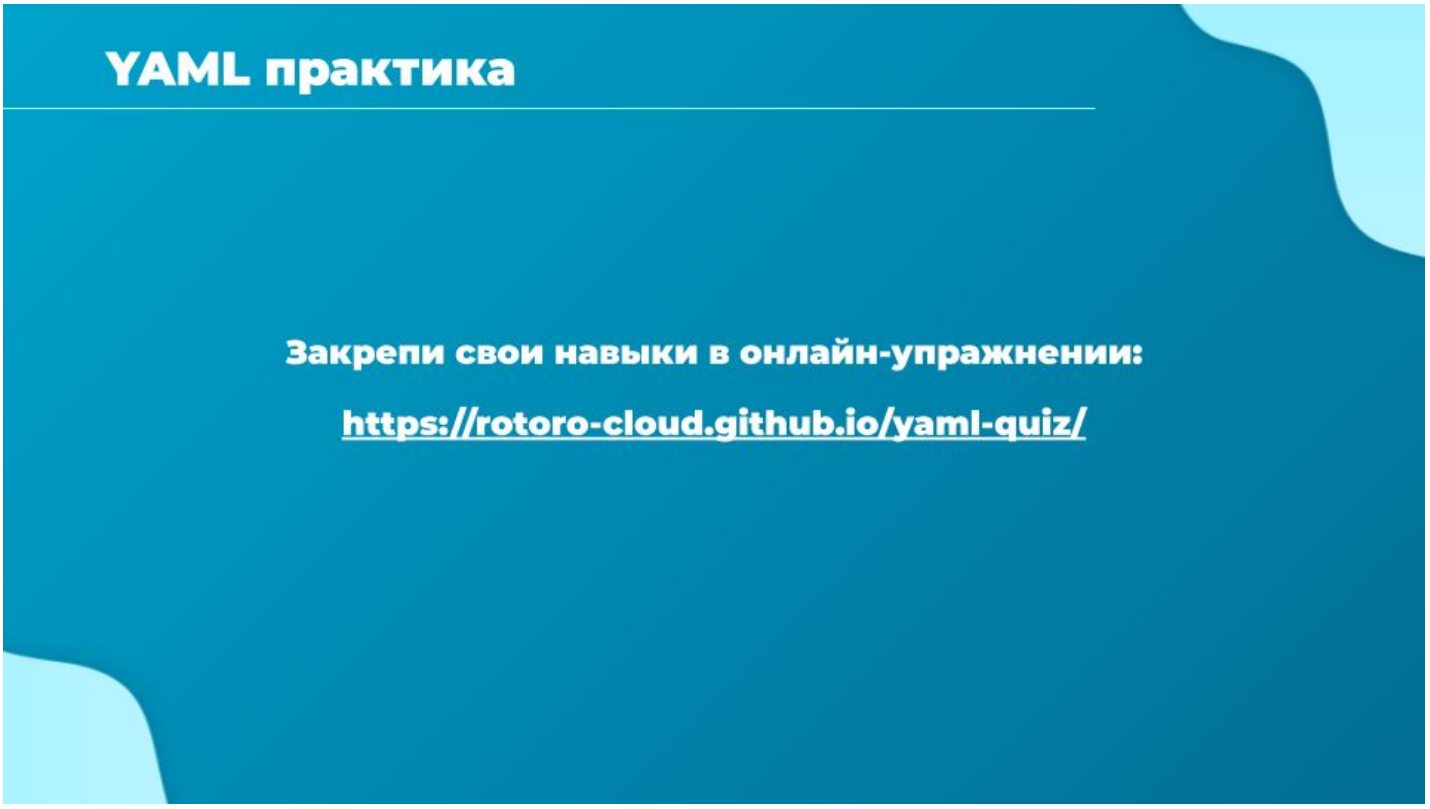
Перед началом упражнений давай примем во внимание пару особенностей формата. Dictionary - это коллекция не требовательная к порядку элементов, в отличие от list, которые порядок соблюдают.

Что это значит? Два dictionaries представленных здесь имеют одинаковые свойства для banana, но порядок свойств не совпадает. В первом fat идет после calories, а во втором наоборот.

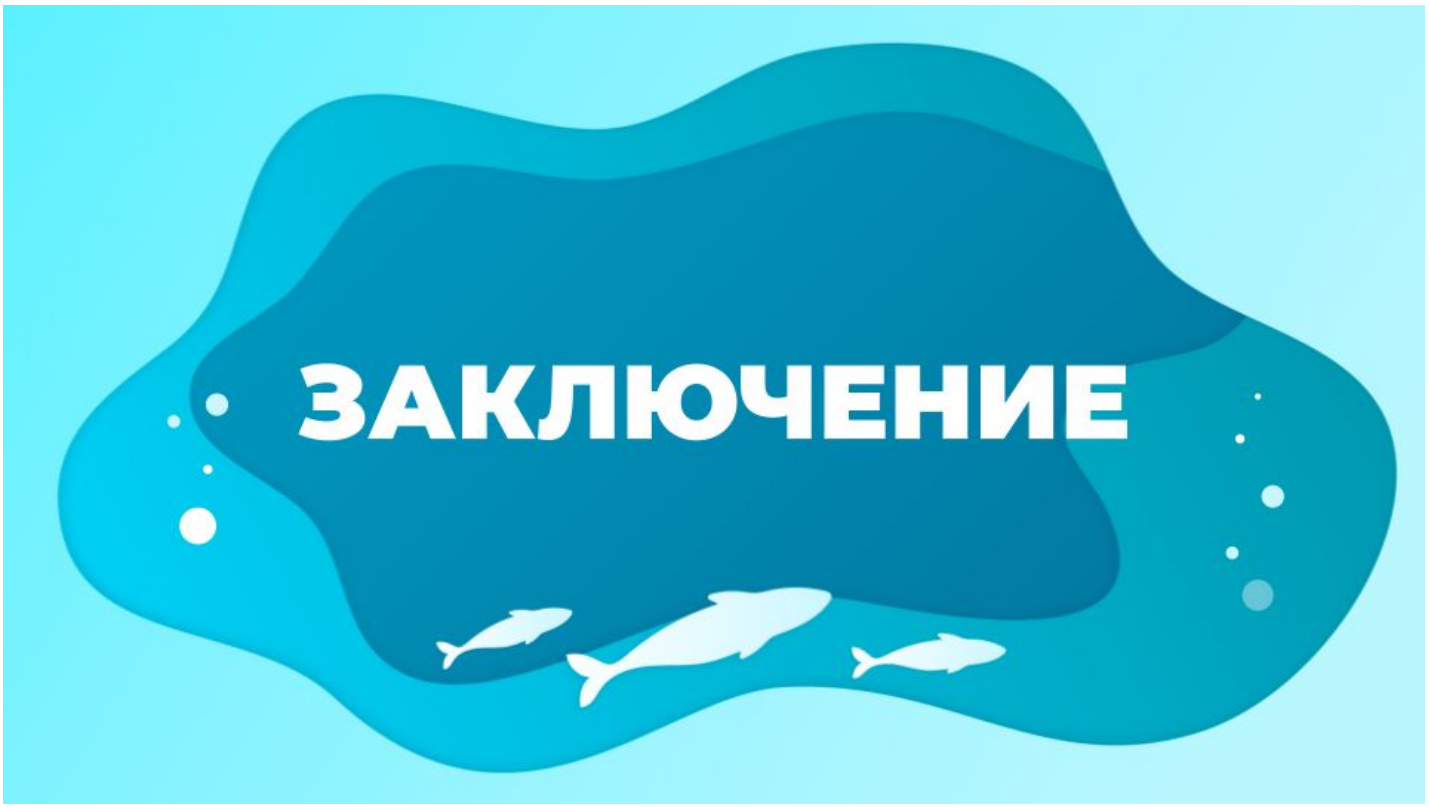
В случае dictionary это не имеет значения, свойства могут быть определены в любом порядке. Два dictionaries будут считаться идентичными, пока все значения всех их свойств совпадают.

Это не относится к массивам. Lists - упорядоченные коллекции, здесь порядок элементов имеет значение. В этом случае два lists не одинаковые, потому что яблоко и апельсин на разных позициях. Не забудь об этом, когда будешь работать со структурами данных.

Еще момент, любая строка начинающаяся с символа ХЭШ (#) будет автоматически проигнорирована парсером, она рассматривается как комментарий.



Теперь мы готовый погрузиться в упражнения с YAML, вперед!



Вот мы и в конце курса Docker для самых маленьких.

В нем я постарался дать общее представление о командах, контейнерах и образах Docker. Показал особенности Docker Compose и прошелся по особенностям хранения и сети в Docker. Также мы затронули реджистри и оркестрацию.

Я старался, чтобы информация заходила максимально просто, а практика помогала тебе закрепить новые знания. Надеюсь у меня получилось, а ты получил отличный user experience от обучения. Если да, напиши мне в группу или слак. А если тебе пришелся по душе мой способ преподавания, значит зайдут и другие курсы, которые я постепенно выкладываю на сайте rotoro.cloud.

Пока их там не очень много, и они базовые, но если я буду чувствовать, что меня поддерживают и эти знания нужны, то буду стараться и дальше. Просто оставь отзыв на Udemy или лайк в Youtube.

Спасибо за уделенное время.
И до следующего курса!

Искренне ваш, Андрей Соколов