GeekBrains

Введение в Docker

Контейнеризация

Урок 3

Оглавление

Введение

Всем привет! В рамках текущей лекции мы наконец-таки (!) прикоснемся к тому, ради чего изучались термины и механизмы работы систем прошлые два урока. На этом уроке мы установим и начнем знакомство с Docker! Разберемся с основными понятиями, инсталлируем и настроим его, а также запустим первые контейнеры.

На этом уроке

1. Архитектуру Docker
2. Процесс установки docker
3. Процесс работы с одним из репозиториев Docker
4. Основные команды, необходимые для работы с Docker

Вступление

Исходя из того, что мы изучили на прошлых уроках, давайте разберемся с тем, что же конкретно дает нам контейнеризация и какие практики работы существуют.

Технологии контейнеризации получили довольно широкое применение в сфере разработки ПО, тестирования ПО, анализа данных и еще во многих сферах. Данные технологии не только помогают сделать приложения более безопасными, облегчить их перенос между платформами и развертывание, но и улучшают возможности с точки зрения масштабирования и позволяют сэкономить деньги. Так как в последние годы тренд роста технологий контейнеризации все актуальнее и актуальнее, необходимо знать их и уметь использовать!

Docker - это платформа, предназначенная для быстрой разработки, развертывания, тестирования и запуска приложений в контейнерах и может быть использована во многих командах при разработке ПО.

На данный момент давайте запомним: контейнер - это программа. Пока контейнер запущен, внутри запущена программа, выполняя ряд полезных действий.

Вы, вероятно, задаетесь вопросом: “Для чего это нужно?”. И действительно - для чего нам изучать контейнеризацию, ограничивать приложения, запускать их в какой-то “виртуальной” среде...

Пример! Раньше чтобы установить какую-нибудь БД или библиотеку, разработчику необходимо было прочитать инструкцию на сайте, скачать набор ПО и зависимостей. А при смене версий на более новые эти зависимости необходимо было удалять и устанавливать обновленные. А если зависимостей много? А если необходимо запускать одновременно и старую версию ПО, и новую? Давайте сюда еще добавим необходимость работы с различными проектами одновременно - и все становится понятно: одного сервера с ОС недостаточно. Необходимо как-то разграничивать среды и приложения между собой.

С контейнерами все проще. Все необходимые компоненты уже находятся внутри контейнера и ПО может их использовать. Не нужно удалять что-либо из системы при переустановке, также не остается никаких следов использования тех или иных зависимостей и версий ПО.

Кроме того, есть еще и единый интерфейс управления, который позволяет исключить необходимость запоминания индивидуальных команд. Достаточно лишь знать команды самого docker и все. А, если возникнут какие-либо ошибки, благодаря изоляции они не затронут хостовую операционную систему.

Пример номер два. Предположим, ваш продукт уже готов и его необходимо протестировать. При этом все тесты занимают около получаса, а установка ПО - час. При этом, находя баг, необходимо декларировать его и отправлять разработчикам на доработку. После устранения багов, процесс повторяется. А теперь представьте: у вас сырой продукт и количество ошибок, найденных на разных этапах тестирования, может достигать десятков и сотен. Каждый раз тратить большее количество времени на установку, чем на тестирование неразумно.

Теперь представим эту же ситуацию. У нас приложение контейнеризировано. Изначально на контейнеризацию и автоматическую сборку приложения тратится некоторое количество времени, но зато потом, чтобы развернуть рабочее ПО необходимо буквально пару минут. Процесс заметно ускорится при таком подходе и у команды появится время, которое можно будет потратить на другие задачи.

Пример номер три. Но перед его началом давайте введем определение.

CI/CD (Continuous Integration/Continuous Delivery) — методология разработки программного обеспечения, которая обеспечивает надежность и скорость создания продукта. Относится к одной из типовых DevOps-практик.

Ее основное отличие в том, что тестирование и сборка кода проводится в автоматическом режиме. Такой подход позволяет избежать ошибок на ранних этапах и сводит риски к минимуму.

Автоматизация. Предположим, что у нас разрабатывается продукт, который довольно быстро выходит на рынок. Для этого необходимо обеспечить процесс работы так, чтобы разработчики занимались лишь написанием кода, тестировщики - его тестированием, администраторы - сетевыми задачами компании, а devops’ы - автоматизацией доставки кода. В этом нам тоже помогут контейнеры и докер. Всю процедуру разработки приложения можно автоматизировать только в случае, если у нас имеется контейнеризация и полная изоляция сред и приложений. Если же мы каждую версию разрабатываемого приложения будем устанавливать на один и тот же компьютер - рано или поздно мы получим конфликты из-за накопленного в ОС программного мусора, который остается при каждой установке ПО.

За счет изоляции исполняемых сред этот процесс будет протекать без проблем, так как каждая новая версия ПО устанавливается на чистую ОС. Об этом мы поговорим позже.

Виртуализация vs контейнеризация

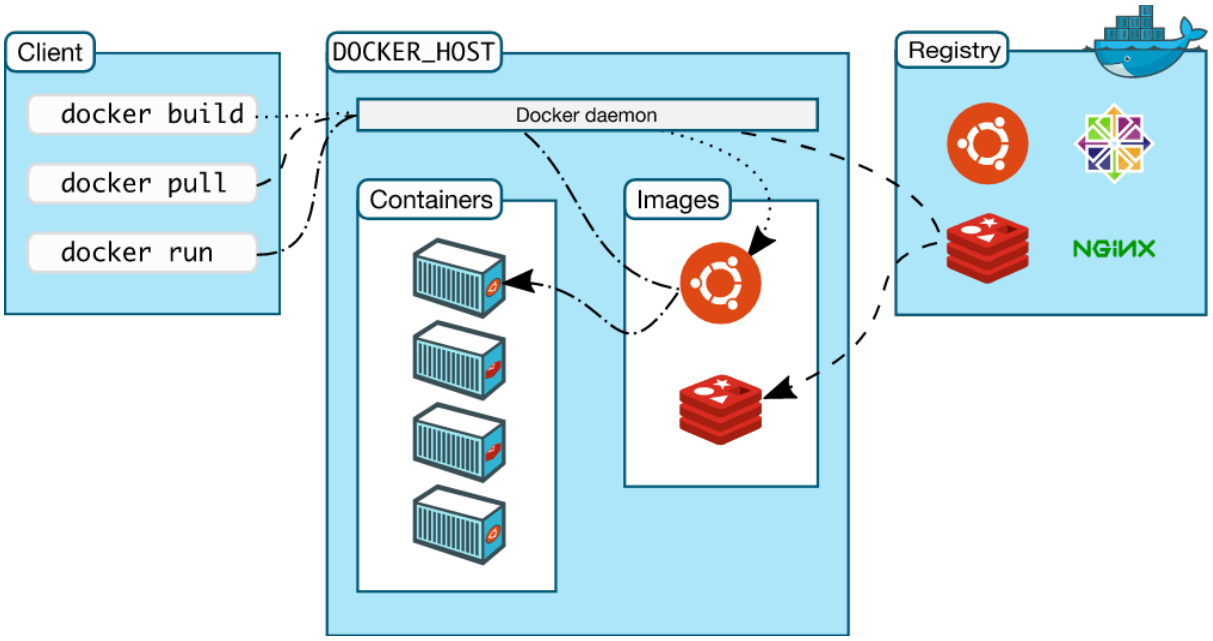
Первой попыткой экономии денег явилась технология виртуализации. Виртуальные машины существуют гораздо дольше контейнеров и точкой отсчета здесь можно считать 2005-2006 годы. Именно тогда начали появляться технологии виртуализации, которыми сейчас пользуются многие люди и компании.

Но речь сейчас не об этом. Виртуализацию мы рассмотрим несколько позднее. Вторая попытка экономии - контейнеризация. О ней мы говорили предыдущие два урока, рассматривая значимые аспекты работы механизма. Важные отличия от виртуализации, которые стоит знать на данный момент:

- Меньшее потребление ресурсов
- Легче переносить между окружениями
- Более быстрый запуск и переход в работоспособное состояние

Архитектура Docker

Начать работу с Docker лучше всего с разбора архитектуры. В этом нам поможет следующая схема:



Итак, давайте разбираться.

Образ контейнера

Давайте начнем с теории, которую потом продемонстрируем на практике. Помимо контейнера, есть еще **образ** (он же image).

Образ - это некий неизменный шаблон проекта (либо программы), который используется в дальнейшем для создания контейнеров с одинаковой базой. В таком контейнере содержится базовая система, набор необходимых библиотек и все это собрано в виде сущности - образа, из которого можно создать контейнер.

Образ также содержит в себе и файловую систему, которая в дальнейшем будет доступна приложению, и ряд метаданных (команды, которые будут выполнены при запуске контейнера, например).

Образ можно скачать с репозитория. Репозиторий - это место, где хранятся в актуальном виде (с поддержкой версий) данные. В нашем случае - образы. Иными словами, репозиторий - это коллекция различных образов контейнеров в реестре.

Важный момент: образы можно как скачать с репозитория, так и создать самостоятельно. После создания образа на локальном компьютере (либо сервере) его можно отправить в хранилище, чтобы остальные пользователи после могли бы его переиспользовать. О создании собственных образов мы поговорим чуть позже.

Каждый образ состоит из слоев, которые в совокупности обеспечивают содержимое, которое мы видим в запущенных контейнерах. Слой - это некий результат способа создания образов.

Важный момент! При работе с Docker запускаются именно контейнеры, а не образы!

Docker Container

Да, определение этой сущности мы уже давали, однако, давайте дадим расширенное понятие с дополнительной информацией.

Итак, контейнер, иными словами, это собранный проект, который состоит из образов. По сути, это упакованное приложение на основе образов. Вместе с приложением упаковке подлежит и среда контейнера. Выполняемый контейнер - это запущенный процесс, который изолирован от других процессов на выполняемой системе, имеющий свои ограничения на потребление ресурсов (ЦПУ, ОЗУ, диска и так далее).

Docker Daemon

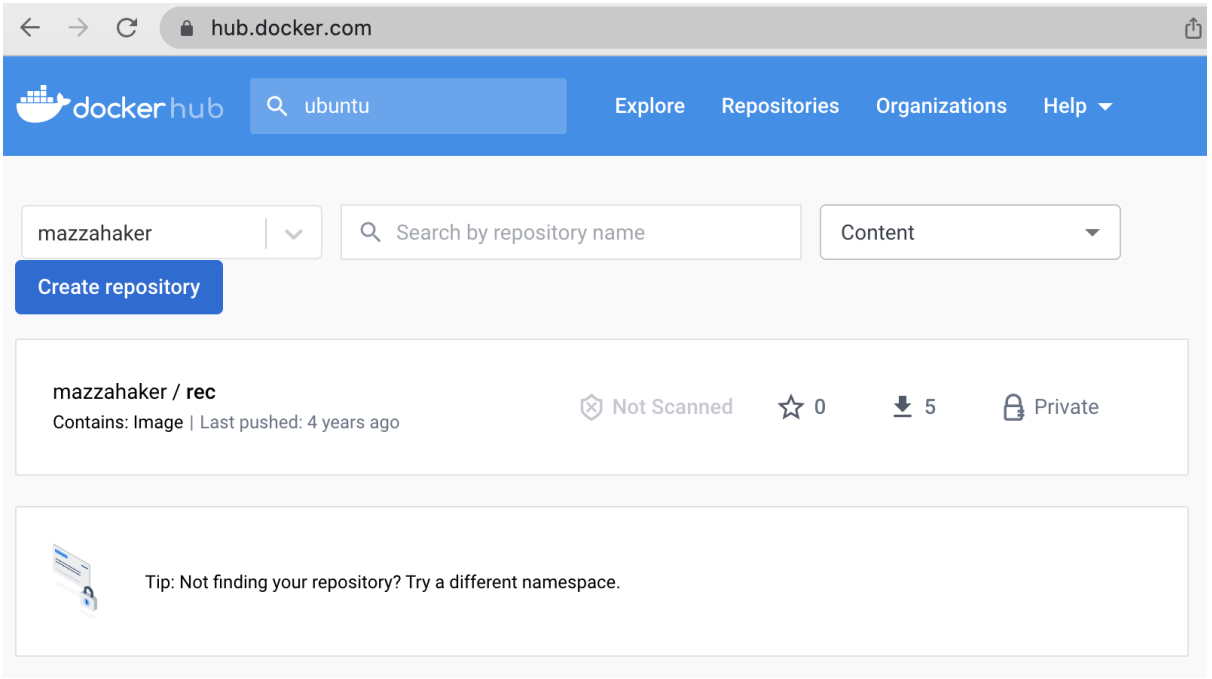
Docker daemon - это служба, управляющая Docker-объектами, такими как: образы, контейнеры, хранилища разного рода, сети и так далее. Также это программная платформа для создания, упаковки, выполнения и распространения приложения. С ее помощью можно скачать образы с хранилища (registry) и запустить из них контейнер.

Docker Registry

Registry server - это репозиторий, где хранятся образы. После создания образа на локальном устройстве, им можно поделиться с остальными, отправив образ в registry. А затем - извлечь его оттуда. Примеры репозиториев: Docker Hub, Google Cloud Container Registry и тд.

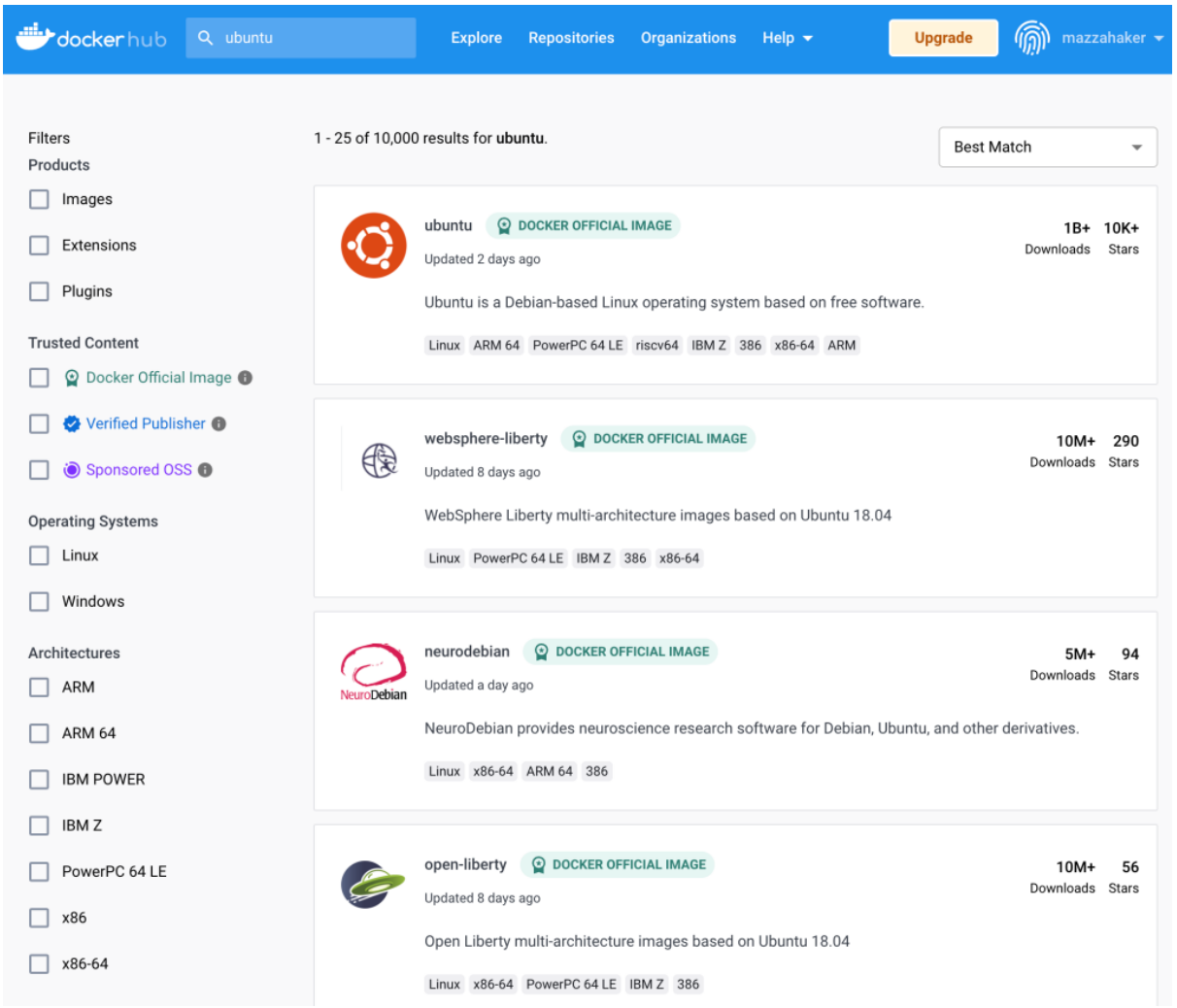
Помимо хранения образов, registry хранит в себе документацию о предоставляемом ПО. Давайте рассмотрим на примере.

Вот так выглядит заглавная страница.



На ней вы можете видеть собственные репозитории (если они у вас есть). Если их нет - не волнуйтесь, мы их создадим и добавим в репозиторий собственные образы.

Предположим, нам необходим образ с ОС ubuntu. Давайте найдем его в поиске:



Введя в строку поиска то, что нам нужно, мы получили множественный вывод. Давайте разбираться. Столбец слева: продукты. Нам необходимы только образы (не расширения, не плагины), так как мы планируем запускать контейнер.

Галочки “Доверенного содержимого”. Здесь доступны несколько вариантов:

- Docker Official Image - таким флагом помечаются официальные образы, которые и рекомендуется использовать. Они точно проверены, имеют достаточную документацию и не доставят проблем при использовании.
- Verified Publisher - таким флагом отмечаются образы, которые загружены проверенным издателем. Их также можно использовать. Чаще всего такими флагами помечаются аккаунты производителей ПО, например, Dell. Такие образы скачиваются клиентами без каких-либо ограничений скорости.
- Docker Sponsored Open Source Software (Docker OSS) - с этим флагом содержится ПО с открытым исходным кодом, которое спонсирует Docker. Это образы, которые спонсируются через программу Docker.

Далее, что можно выбрать: тип ОС - Linux или Windows. И, напоследок, архитектуру. Контейнеры могут быть построены под различную архитектуру процессоров и выполняться на разных устройствах: от привычных нам x86, x86-64 платформ до ARM и, например, IBM Z платформ.

Также образы сортируются в списке по наилучшему совпадению и большему числу загрузок. Собственно, необходимый нам образ - первый в списке. Откроем его.

Первое, что мы видим - вкладки. У каждого образа могут быть теги. С их помощью мы можем выбрать необходимый нам образ: Ubuntu 18.04, Ubuntu 20.04 и тд. По умолчанию, если не указывать тег, будет загружаться последний доступный образ. То есть образ с тегом latest. Для простейших примеров это удобно. Но при разработке и построении сложных систем крайне рекомендуется указывать конкретную версию, а не latest.

Простой пример: у нас есть сайт, состоящий из нескольких контейнеров - веб и БД. При очередном обновлении у веб-части сменился python с версии 2.7 на версию 3.9. Те, кто изучал python, знают, что версии 2.x и 3.x несовместимы и код, написанный под 2 версию не будет работать с третьей. При очередном обновлении сайт просто перестанет работать.

Идем дальше. Следом идет описание контейнера, который нас заинтересовал с подробным описанием, переменными, которые можно использовать в контейнере и куча дополнительной полезной информации.

Напоследок - лицензионное соглашение, описывающее возможности использования этого контейнера.

Установка Docker

Пожалуй, хватит теории! Давайте наконец-таки установим Docker и запустим первый контейнер, а далее - будем разбираться со слоями, сборкой контейнеров под собственные нужды и прочей информацией.

Сам Docker, как и любая другая программа, требует установки. Скачать Docker можно с официального сайта и установить, следуя инструкциям. Давайте это и сделаем!

Первое, что нужно сделать - открыть сайт https://docs.docker.com/get-docker/. Далее - выбрать Docker под нужную ОС. В нашем случае хост-системой будет выступать Linux Ubuntu.

Перед установкой предлагается ознакомиться с системными требованиями, которые имеются у каждого продукта. Для наших работ имеется машина, удовлетворяющая этим требованиям: минимум 4Гб ОЗУ, также на машине имеется 4 процессорных ядра.

Далее выбираем ОС - Ubuntu и видим перед собой список команд, которые предлагается выполнить.

На текущей системе не было установлено никаких систем, поэтому удалять старые версии не нужно.

Для начала обновим пакеты, используя команду:

```
sudo apt-get update
```

И далее установим необходимый набор зависимостей:

```
sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

Следующий шаг - добавление ключей:

```
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

Ну и напоследок давайте добавим репозиторий:

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Теперь все готово! Повторно обновляем список пакетов, чтобы добавились пакеты из свежедобавленных репозиториев:

```
sudo apt-get update
```

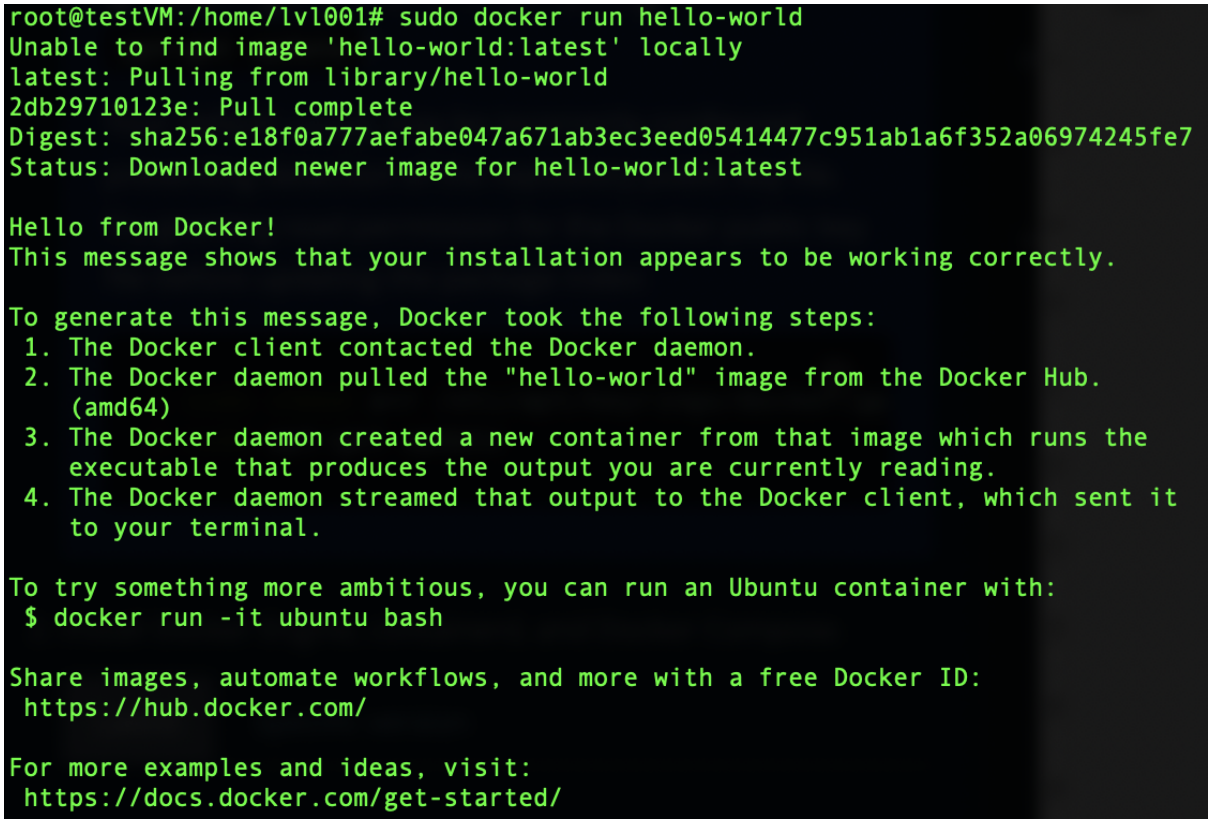
И установим набор необходимых компонентов:

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

Проверить работу достаточно просто - запустим команду:

```
sudo docker run hello-world
```

Вывод должен быть следующий:



А вот теперь давайте разбираться! Первое, что видим как результат запуска - сообщение о недоступности образа. Это система пытается найти образ с названием **hello-world**. Не найдя его локально, система идет искать образ в публичный registry. По умолчанию - это https://hub.docker.com/. Тут содержится большинство образов. Скачав образ, система сверяет хэш в целях безопасности и проверяя образ на целостность и затем запускает, о чем и уведомляет нас в консоли.

Давайте выполним последнюю команду повторно! Как мы видим, повторное скачивание не производилось, так как контейнер уже доступен локально.

Основные команды Docker

Возможно теперь, окунувшись в практику, стало понятнее как запускается ПО. Прежде чем мы продолжим, давайте разберем основные команды Docker, которыми еще неоднократно воспользуемся.

docker --help - выведет список доступных команд, если вы вдруг что-то забудете.

docker <command> --help - выведет информацию о команде. Пример: docker images --help.

docker --version - выведет нам версию установленного докера.

Если вы хотите вывести больше информации о рабочей системе, можно использовать docker info.

Ну, об установленном докере мы узнали достаточно. Самое время начать работать с контейнерами. Начнем с поиска нужных образов. Как искать образы через сайт репозитория уже показано ранее. Теперь давайте найдем образ через консольную команду: docker search debian. Здесь в списке также есть отметка об “официальности” образа. Среди всех выберем ubuntu.

Предположим, что нам необходимо только скачать образ, но не разворачивать контейнер. Для этого воспользуемся командой docker pull ubuntu. Как видим, докер начал скачивать версию образа с тегом latest (последний доступный) т.к. тэга для образа я не задавал. Теперь давайте укажем тег и скачаем ubuntu версии 18.04. docker pull ubuntu:18.04.

Итак, образы скачаны. Давайте теперь выведем список доступных образов в системе docker images. Здесь можно видеть, что в системе доступны несколько образов, которые могут отличаться тегами (и, соответственно, версиями).

Допустим, нам больше не нужна старая версия ubuntu. Давайте удалим ее: docker rmi ubuntu:18.04. теперь выведем список образов и убедимся в том, что старая версия действительно отсутствует.

Допустим другой вариант: ваше приложение состоит из множества контейнеров с ПО различных версий. После исправления ряда ошибок, вам необходимо чистить систему, удаляя все образы, скачанные локально. Можно удалять по одному, а можно просто удалить сразу все: docker rmi $(docker images -aq). Давайте разберемся. Первая часть команды понятна - удаляет образы. А что же внутри скобок? Если запустить сначала то, что внутри, то мы выведем

идентификаторы образов, доступных локально на системе. Все выглядит корректно, можно запускать.

Отлично, с образами мы поработали, теперь самое время разобраться с контейнерами!

Мы уже запускали эту команду: docker run hello-world. По сути, команда скачивает образ hello-world и запускает из него контейнер.

Давайте запустим что-то более серьезное. Например, ubuntu. Она как раз у нас уже скачана: docker run -it ubuntu bash. Теперь разберемся с командой: первая половина говорит о том, что нужно запустить контейнер из образа убунту. Флаг -it говорит о том, что нужно запустить контейнер в интерактивном режиме (не в фоновом режиме), а приписка bash говорит, какую команду из контейнера необходимо запустить для работы. В частности, можно запустить командный интерпретатор bash, shell и, тем самым, остаться работать внутри контейнера. В случае, если мы, например, запустим docker run -it ubuntu date, система просто выведет нам текущую дату и время и завершит работу контейнера.

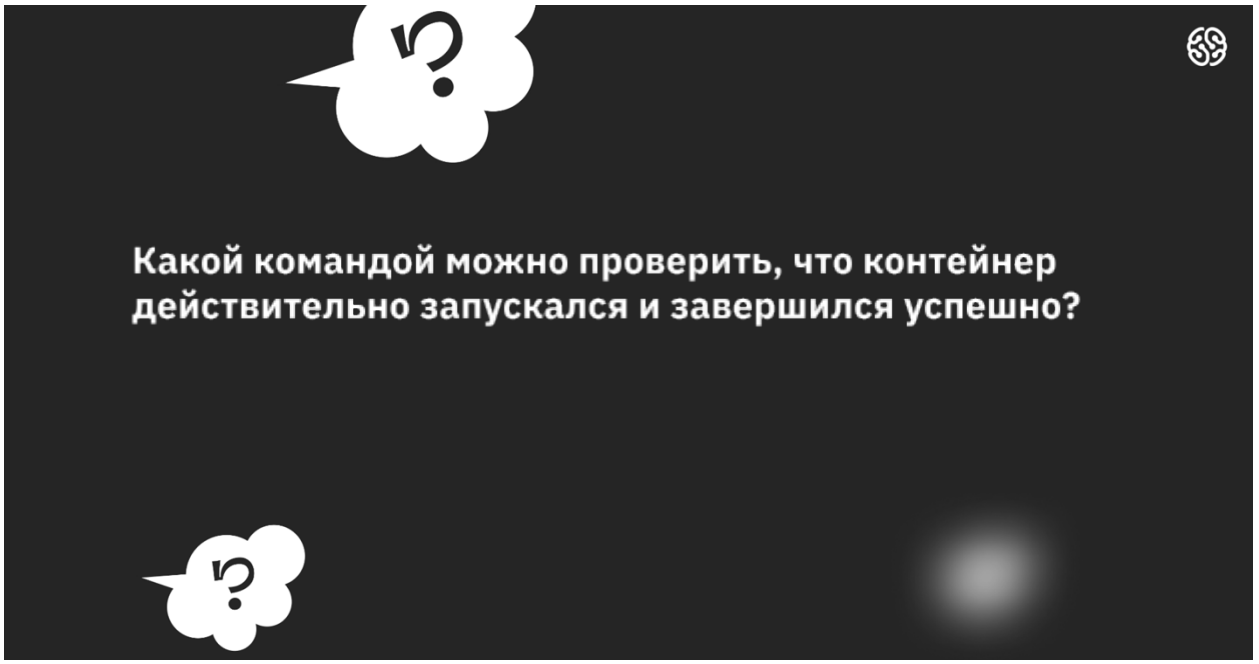
Итак, запускать контейнер мы научились, а теперь давайте сделаем публикацию портов. То есть сделаем контейнер доступным извне (аналог NAT у маршрутизаторов). В этот раз мы запустим контейнер nginx, так как он сразу позволит выводить данные docker run -–publish 8080:80 nginx. Таким образом, мы сказали системе открыть изнутри контейнера порт 80 и сделать его доступным снаружи через порт 8080. Давайте проверим, что все работает командой curl.

curl 127.0.0.1:8080

Как видим, nginx запустился. В одном окне мы получили от него ответ, а в другом - лог о том, что происходил запрос в систему.

Следующее дополнение позволяет установить имя запускаемому контейнеру docker run --name testcontainer ubuntu.

Проверить результат выполнения можно с помощью команды docker ps. Как видим, вывод пуст. Уточняю: docker ps выводит список лишь запущенных контейнеров. Чтобы посмотреть список остановленных, необходимо вводить docker ps -a. И тут мы видим кучу созданных контейнеров, которые мы запускали ранее. Имя у них выдавалось автоматически. А вот тот, которому мы давали имя. Теперь его можно запустить по имени контейнера при необходимости. Это мы и сделаем: docker start testcontainer. Вывод немногозначен, так как мы пока еще не указывали никаких аргументов. Однако, проверить то, что контейнер действительно запускался и завершился успешно можно уже известной нам командой.



Ответ: docker ps -a. Здесь мы видим, что наш контейнер запускался (такое-то время назад) и был успешно завершен. Можем проверить еще раз, запустив старт и docker ps.

Ну и напоследок рассмотрим команды, необходимые для удаления контейнеров. Чтобы удалить контейнер, нужно ввести docker rm testcontainer. Далее следует проверить, что наш контейнер недоступен командой docker ps -a, и убедиться, что он пропал.

А теперь, раз уж остальным контейнерам имена мы не давали, давайте удалим оставшиеся. Сделать это легко: docker rm $(docker ps -a -q). Опять же, проверим то, что в скобках. Введем и посмотрим на вывод. Флаг -a говорит вывести, в том числе, остановленные контейнеры, а флаг -q говорит о том, что вывести необходимо ID контейнеров. Запускаем и проверяем результат командой docker ps -a. Следом давайте удалим имеющиеся локально образы docker rmi $(docker images -a -q).

И в конце давайте научимся смотреть занятое место на диске: docker system df. Сейчас здесь нули, так как мы удалили образы и контейнеры, однако, крайне рекомендуется периодически проверять и запускать команду в целях мониторинга.

Также есть команда docker system prune -af. Она позволит удалить неиспользуемые данные и очистить дисковое пространство.

Заключение

Давайте проведем параллели этого урока с уже изученной информацией. Первые два урока на лекциях мы рассматривали механизм пространства имен и механизм контрольных групп. А на практике запускали изолированные приложения и процессы, проверяя, что все строго локализовано, ограничено и не имеет доступа в хостовую систему.

Сегодня же мы запустили те же приложения (командный интерпретатор bash, например) и изолировали теми же способами, что и ранее, но уже в полуавтоматическом режиме. Бóльшую часть работы за нас уже сделал docker: создал изолированные пространства имен, выделил ресурсы для запуска. На семинаре же мы разберем более детально эти процессы и попробуем ограничить запускаемые приложения.

В заключении важно подчеркнуть, что на данном уроке мы познакомились непосредственно с Docker, изучили набор основных команд, которые в дальнейшем будем использовать, а также научились работать с репозиторием Docker. На практике мы обязательно рассмотрим более сложное взаимодействие с контейнерами и рассмотрим различные флаги, которые используются для работы.

Термины, используемые в лекции

Зависимость - это значит полагаться на что-то. Например, если сказать, что люди слишком много полагаются на мобильные телефоны, то, получается, они зависят от них. Когда приложение A использует некоторую функциональность из приложения B, тогда говорят, что приложение A зависимо от приложения B.