# Лекция 7. Основы Docker

## Цель лекции

- Понять основы технологии контейнеризации
- Изучить инфраструктуру Docker
- Разобрать схему команды управления контейнерами
- Научиться запускать контейнеры из образов
- Познакомиться с запуском систем контейнеров

## Дополнительные материалы (ссылки, файлы)

1. Установка Docker в Ubuntu
2. Установка Docker Compose
3. Compose file reference
4. Введение в Docker
5. Шпаргалка по командам Docker
6. Docker и сети
7. Docker Compose

## Термины

**Контейнеризация** — метод виртуализации, при котором ядро операционной системы поддерживает несколько изолированных экземпляров пространства пользователя вместо одного. Эти экземпляры с точки зрения пользователя полностью идентичны отдельному экземпляру операционной системы. Простыми словами, контейнеризация позволяет виртуализировать процесс.
**Docker Hub** — платформа для распространения Docker-контейнеров и управления ими.
**Dockerfile** — это сценарий, который состоит из последовательности команд и аргументов, необходимых для создания образа. Такие сценарии упрощают развёртывание и процесс подготовки приложения к запуску.
**YAML** — дружественный формат сериализации данных, концептуально близкий к языкам разметки, но ориентированный на удобство ввода-вывода типичных структур данных многих языков программирования.

## План

1. Технология контейнеризации приложений.
2. Сравнение виртуализации и контейнеризации.
3. Docker и его архитектура.
4. Реестр образов Docker Hub.
5. Запуск контейнера hello-world.
6. Базовые операции с утилитой docke.
7. Работа сети в Docker. Проброс портов.

8. Подключение томов.
9. Запуск контейнера nginx.
10. Работа внутри контейнера.
11. Docker Compose и его синтаксис.

# Текст лекции

## Технология контейнеризации приложений

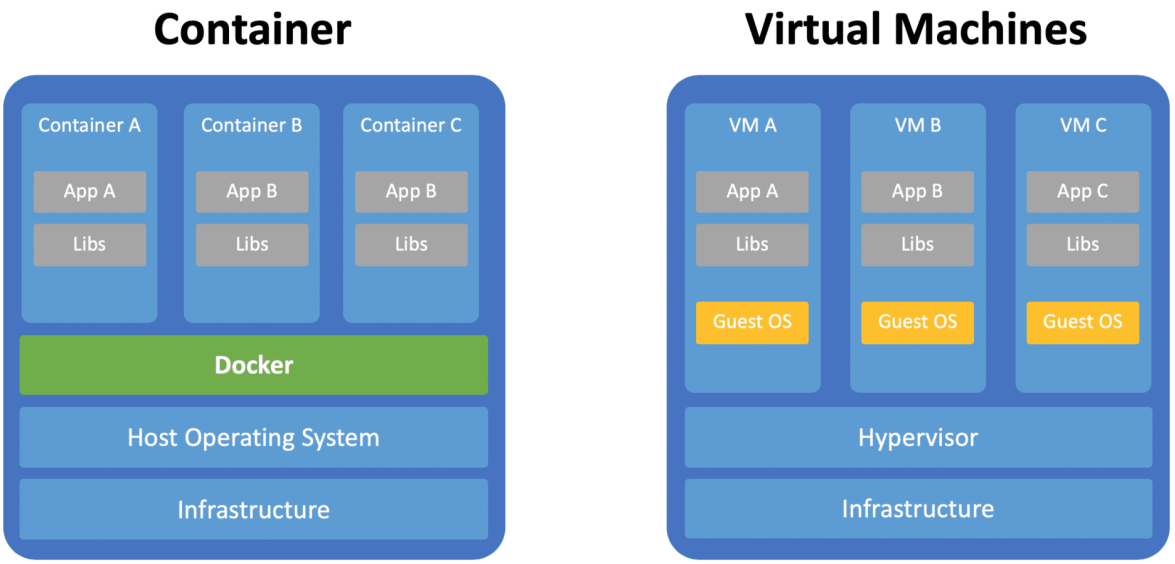
В самом начале курса мы задействовали технологию виртуализации с использованием продукта VirtualBox. Мы получили возможность запуска полноценной версии Linux на другой операционной системе (Windows, macOS, Linux). То есть, VirtualBox виртуализировал наше железо, чтобы предоставить его другой операционной системе, которая будет работать поверх нашей основной (хостовой). Мы получили полную изоляцию хостовой и гостевой ОС и возможность управления всеми ресурсами виртуальной машины (ВМ).
Однако, не всегда нам нужен именно такой уровень изоляции. Если наша основная ОС — Linux, то нам достаточно **изолировать приложение**. То есть, мы хотим создать облегченный вариант виртуализации, где используется общее **ядро** ОС (Linux), но окружение (утилиты, библиотеки, конфигурация и т.д.) приложений различное. Такая технология получила название **контейнеризация**.
Контейнеры в нашем понимании (Docker) имеют аналогию с грузовыми контейнерами. В области логистики существует понятие контейнерных перевозок, в основе которых стоят стандартные контейнеры, которые можно использовать практически на любых видах транспорта (воздушный, морской, железнодорожный, автомобильный). Появление стандартных транспортных контейнеров радикально снизило стоимость перевозок, так как появились стандартные тарифы, упростилось управление перевозками. Контейнеризация приложений призвана решать похожие задачи: стандартизация доставки приложений на системы за счет стандартной упаковки в образы.
По сути контейнер это виртуальная машина пользовательского уровня (без ядра), которая содержит одно или несколько приложений и все их зависимости. Здесь можно провести параллель с пакетами `snap`, которые также имеют некоторую изоляцию и содержат все зависимости внутри.
Основные преимущества применения контейнеризации следующие:

- быстрое развёртывание приложений из образов;
- полная повторяемость при запуске (надёжность);
- высокая скорость запуска и удаления контейнеров;
- низкие накладные расходы на виртуализацию;
- возможность преодоления несовместимости софта и ОС;
- автоматизация операций управления приложениями.

В процессе изучения технологии вы сможете проверить эти преимущества на практике. Давайте подробнее посмотрим, в чем основные различия виртуализации и контейнеризации.

## Сравнение виртуализации и контейнеризации

Полноценная виртуализация даёт полный простор для использования любых совместимых операционных систем на любой основе (хосте). Единственное требование — совместимость микроархитектуры (сейчас это обычно AMD64). Контейнеризация же ограничивает выбор систем — только Linux.




На картинке выше показано архитектурное различие контейнеров и виртуальных машин. Виртуальные машины используют гипервизор или хостовую ОС, поверх которой запускается другая операционная система целиком.
В контейнере мы получаем только пользовательские уровни: библиотеки (`Libs`) и само приложение (`App`). Ядро операционной системы остаётся основным, нет лишнего уровня взаимодействия между хостовым и гостевым ядрами. Компонент Docker, который показан на рисунке выполняется вспомогательные функции (работу с сетью, файловыми системами).
Следствием этих различий является меньший размер контейнеров, быстрый запуск и меньшие накладные расходы при работе. Таким образом, если нам требуется запустить Linux-приложение с изоляцией, мы можем решить свою задачу с помощью контейнеризации.
Настало время подробнее познакомиться с архитектурой контейнеров Docker и их жизненным циклом.

## Docker и его архитектура

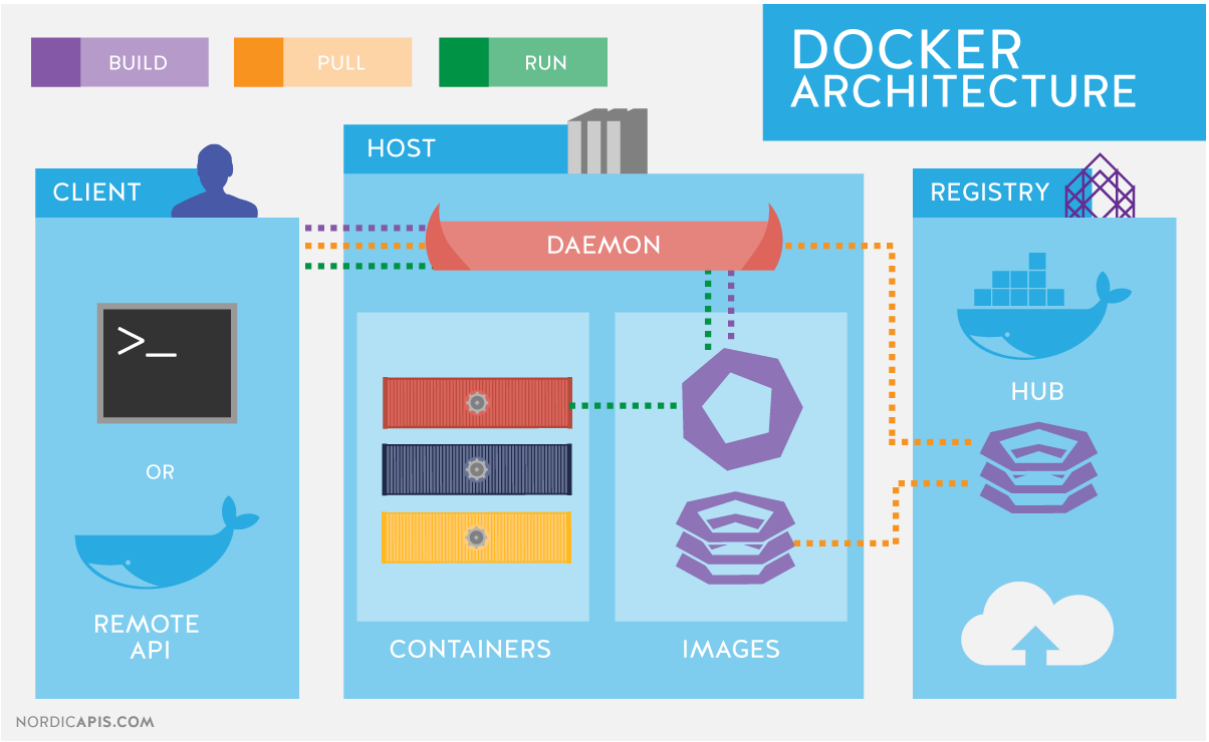
Docker с точки зрения операционной системы состоит из трёх частей:

1. **Docker-демон** — процесс, который отвечает за поиск, скачивание образов, запуск контейнеров и т. д.
2. **Docker-клиент** — интерфейс взаимодействия пользователя и демона Docker. Именно через команду Docker пользователь будет скачивать образы и запускать контейнеры.
3. **Docker Hub** — хранилище образов для Docker.

С точки зрения архитектуры Docker состоит из следующих компонентов:

1. **Images (образы)** — это своеобразный шаблон, который содержит экземпляр операционной системы с набором библиотек, необходимых для работы приложения.
2. **Registry (реестр)** — публичное или закрытое хранилище образов. Пример публичного реестра образов — Docker Hub.
3. **Container (контейнер)** — запущенное приложение, которое создано из образа.

Наша главная утилита для управления контейнерами это клиент `docker`. Мы будем использовать её для удобного управления образами, контейнерами и другими элементами инфраструктуры.
На картинке ниже показаны основные процессы, которые мы будем использовать при работе с Docker.




В левой части находится клиент — либо утилита `docker`, либо другие приложение, использующее API Docker.
В центре наша хостовая машина с установленным Docker-демоном, локальными образами и контейнерами.
В правой части находится реестр Docker Hub — централизованное удалённое хранилище образов.
Все эти элементы связаны процессами `docker build`, `docker pull` и `docker run`.
В рамках курса мы рассмотрим использование готовых образов (не будем разбирать сборку через `docker build`).

## Реестр образов Docker Hub

При установке Docker автоматически подключается реестр образов Docker Hub. Из него мы можем получить широчайший выбор образов для решения разнообразных задач. Попасть в веб-интерфейс реестра можно по адресу: https://hub.docker.com/ .
Каждый образ имеет название и набор тегов — вариантов или версий.

В реестре есть пометки официальных релизов образов — они создаются разработчиками софта и поэтому имеют максимальный уровень доверия. Если образ не имеет никаких пометок, то к его содержимому нужно относиться настороженно.
У любого образа есть тег по умолчанию: `latest`. Под этой версией понимается новейшая версия и стандартный вариант образа.
При большом наборе нестандартных образов, компании создают для них собственные реестры, которые поддерживаются самостоятельно.
Теперь, мы разобрали все основные компоненты архитектуры Docker и готовы перейти к практике.

## Запуск контейнера hello-world

Итак, нам нужно запустить простейший контейнер `hello-world`. Для начала нужно установить Docker в систему:

```
sudo apt install docker.io
```

После установки мы получаем все необходимые компоненты.
Проверить работоспособность можно командой docker без параметров:

```
sudo docker
```

Далее команду `docker` будем указывать без `sudo`, хотя мы подразумеваем выполнение команды с правами `root`. Команда без параметров покажет справку по основным режимам и вариантам команды.
Для проверки системы целиком запустим тестовый контейнер:

```
docker run hello-world
```

Вывод команды покажет скачивание образа и далее создаст из него контейнер, который сразу же запустится. Результат работы контейнера это вывод текста и предложение запустить другой контейнер. Если всё так, значит Docker работает и установлен корректно.
Мы только что запустили простой контейнер, который не работает в режиме демона (сервиса), он выполняет свою команду и завершает работу.
Далее посмотрим, какие еще команды есть в утилите `docker`.

## Базовые операции с утилитой docker

Начнём с просмотра контейнеров:

```
docker ps
```

Пока здесь нет ни одного контейнера. Но мы только что создали контейнер, почему он не отображается? Дело в том, что мы смотрели активные контейнеры. Чтобы посмотреть все контейнеры, нужно добавить параметр -a:

```
docker ps -a
```

Теперь мы видим свой первый контейнер из образа `hello-world`.
Обращаться к контейнеру можно либо по ID (использовать первые цифры) либо по имени
Для просмотра образов, которые были использованы в системе:

```
docker images
```

Искать образы можно либо через веб-интерфейс Docker Hub, либо в консоли:

```
docker search nginx
```

Отдельно можно скачать образ, не создавая контейнера:
`docker pull nginx`
Если контейнер создан в режиме демона (`-d`), то к нему применимы команды `start, stop, restart.` Работают они аналогично командам утилиты systemctl.
Удаление контейнера:
`docker rm 9cbf7c3230d0`
Контейнер можно удалить только в остановленном состоянии.
Для удаления образа используем:
`docker rmi hello-world`
Мы можем удалить только те образы, которые не используются в контейнерах, поэтому сначала удаляем контейнеры на их базе, а потом уже образы.
На этом мы разобрались с базовыми манипуляциями в Docker, теперь перейдём к более сложным конфигурациям и поговорим о сетевых режимах.

## Сеть в Docker

Сеть в Docker реализована посредством четырёх сетевых драйверов. Можно провести аналогию с типом подключения в VirtualBox:

1. **Bridge** — сети по умолчанию, аналог типа подключения NAT в VirtualBox. Связь устанавливается через Bridge-интерфейс, который поднимается в операционной системе при установке Docker и носит название `docker0`. Этот интерфейс можно увидеть, выполнив команду `ip -a`.
2. **Host** — с помощью этого драйвера контейнер получает доступ к собственному интерфейсу хоста. Аналог подключения «Мост» в VirtualBox.
3. **Macvlan** даёт контейнерам прямой доступ к интерфейсу и суб-интерфейсу (VLAN) хоста.
4. **Overlay** позволяет строить сети на нескольких хостах с Docker.

Самые часто используемые режимы это `bridge` и `host`. По умолчанию сеть изолирована и работает в режиме `bridge`. Для доступа в контейнер по сети извне нужно настраивать проброс портов (опция **`-p`**).
В режиме `host` изоляция сети отсутствует и приложения в контейнере получают прямой доступ к сети, как обычные приложения.
Посмотреть список сетей Docker:
`docker network ls`
Вывести детали настройки сети:
`docker inspect network_name`
Изоляция сети и реализация работы режимов осуществляется с помощью нескольких механизмов, в том числе `iptables`, демонов `docker-proxy` и дополнительных логических интерфейсов. При запуске демона `dockerd` он восстанавливает все необходимые правила обработки пакетов, это стоит учитывать при управлении конфигурацией правил `iptables`.
Следующий важный элемент настройки контейнера – тома (`volume`).

## Подключение томов

Каждый контейнер по умолчанию полностью изолирован с точки зрения файловой системы. То есть, имеет собственную иерархию директорий и не имеет доступа к хостовой файловой системе.
Однако, идеология контейнеров предполагает неизменное состояние (stateless). То есть, изменения файлов внутри контейнера будут сохраняться, но только до той поры, когда мы решим его обновить. Обновление контейнера состоит из удаления и создания нового из следующей версии образа. В этом процессе все изменения в файловой системе контейнера будут потеряны.
Поэтому для сохранения всех данных на постоянной основе часто используются **тома** (`volumes`) – внешние хранилища данных для контейнера. Это могут быть директории с данными, конфигурационные файлы, рабочие директории и т.д.
Подключение тома производится с помощью опции **`-v`** при создании контейнера.
После подключения тома мы получаем доступ к хостовой файловой системе и можем пользоваться ей как внутренней. При этом все данные тома при удалении контейнера будут сохранены.
Тома могут быть внутренними для Docker (`named volumes`) или точками монтирования хостовых директорий (`bind mounts`). В нашем примере с Nginx мы будем использовать второй вариант.
Итак, мы разобрали все элементы для запуска реального контейнера, на примере которого мы используем работу с сетями и томами.

## Запуск контейнера nginx

Соберём все знания по Docker для запуска контейнера с веб-сервером Nginx. Мы будем использовать стандартную сеть `bridge`, сделаем проброс портов и подключим том. Полная команда будет выглядеть так:

```
docker run -d --name nginx1 -p 80:80 \
 -v /var/www/html:/usr/share/nginx/html --restart always nginx
```

В этой команде мы создаём контейнер (`docker run`) в режиме демона (`detached -d`), с именем `nginx1`. Также мы добавили проброс портов (-p) для доступа к контейнеру извне. Слева указывается порт хоста, справа – порт внутри контейнера.
Далее мы добавили том (`-v`), который со стороны хоста (левая часть) находится в каталоге `/var/www/html`, а со стороны контейнера (правая часть) в `/usr/share/nginx/html`.
Последний параметр в команде это имя образа для контейнера.
Важно учитывать, что на момент запуска контейнера в хостовой ОС порт, который мы используем должен быть свободен, иначе контейнер не сможет запуститься.
Мы запустили контейнер, можно проверить его статус через `docker ps`. Теперь можно зайти внутрь контейнера и изучить его содержимое.
Еще один момент отличия работы с контейнерами: файлы логов (журналов) внутри контейнера не ведутся, для просмотра записей журналов пригодится команда:

```
docker logs nginx1
```

Для просмотра подробностей конфигурации контейнера можно использовать команду:

```
docker inspect nginx1
```

## Работа внутри контейнера

Сразу оговоримся, идеология Docker не предполагает работу внутри контейнера в штатном режиме. Контейнер должен быть собран таким образом, чтобы работать без изменений сразу после запуска. Поэтому при попытке работать внутри контейнера мы будем сталкиваться с неудобствами, например, контейнер часто лишён стандартных утилит, редакторов и других компонентов для экономии пространства.
Попасть внутрь контейнера с Nginx мы можем с помощью команды:

```
docker exec -ti nginx1 bash
```

Далее мы попадаем в консоль контейнера и можем посмотреть содержимое его файловой системы.
Обратите внимание на минимальный набор файлов, стандартную для Linux структуру файлов и работу тома — директории `/usr/share/nginx/html`.
Как мы уже говорили выше, все изменения в файловой системы контейнера будут сохранены, но только до момента его удаления. Поэтому хранить ценные данные в контейнере с приложением не стоит.
Мы завершили цикл знакомства с отдельным контейнером, теперь сделаем следующий шаг и рассмотрим команду docker-compose, которая позволяет объединять отдельные контейнеры в систему.

## Docker Compose и его синтаксис

Для многих задач необходимо запускать несколько контейнеров, объединённых одной задачей. Например, наше веб-приложение может состоять из базы данных (mysql) и сервера приложений (apache). С точки зрения организации мы можем поместить эти компоненты в один контейнер или создать несколько. Второй путь больше приветствуется в Docker, так как даёт большую гибкость (например, при масштабирование схемы).
При запуске системы контейнеров мы используем docker-compose. Это утилита и формат конфигурационного файла. В файле docker-compose.yml мы прописываем все свойства контейнеров, настройки томов и сетей.
Пример минимального конфигурационного файла docker-compose.yml:

```
version: '3'
services:
  nginx:
    image: nginx:latest
    ports:
      - 80:80
    volumes:
      - /var/www/html:/usr/share/nginx/html
```

Здесь строка `version '3'` говорит об использовании третьей версии формата файлов для Docker Compose. Директива service описывает службу, которую мы будем запускать, дальше идёт имя `nginx`. Собираем контейнер из последней стабильной версии nginx, доступной на Docker Hub: `image: nginx:latest` и пробрасываем 80-й порт хост-машины и каталог `/var/www/html`, используя директивы `ports` и `volumes`.

Таким образом мы можем добавлять нужные нам контейнеры в секцию `services` и получим описание всех компонентов системы в одном файле.
Сначала необходимо установить утилиту docker-compose:
`apt install docker-compose`
Для создания и запуска контейнеров воспользуемся командой (через sudo):
`docker-compose up -d`
Важно сохранять синтаксис файла docker-compose.yml:

- каждый уровень вложенности выделяется отступом;
- каждый отступ – 2 пробела;
- нельзя нарушать иерархию (вложенность) директив;
- версия файла должна быть не выше, чем версия установленной утилиты.

При копировании примеров из статей особенно внимательно относитесь к отступам в каждой строчке — они критичны для работы. Для проверки синтаксиса можно использовать утилиту `yamllint` (`apt install yamllint`).

## Итоги занятия

- Познакомились понятием контейнеризации.
- Узнали различия по сравнению с виртуализацией.
- Изучили архитектуру Docker.
- Запустили базовый контейнер.
- Разобрали работу с сетью.
- Создали полноценный контейнер с Nginx.
- Познакомились с работой docker-compose.

## Анонс Лекции 8.

На следующем занятии мы познакомимся со скриптами Bash.

## Рекомендуемые материалы и литература