МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

Технологии хранения в системах кибербезопасности

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень	бакалавриат
	(бакалавриат, магистратура, специалитет)
Форма обучения	очная
	(очная, очно-заочная, заочная)
Направление(-я) подготовки	10.05.04 Информационно-аналитические системы безопасности
	(код(-ы) и наименование(-я))
Институт	Кибербезопасности и цифровых технологий (ИКБ)
	(полное и краткое наименование)
Кафедра	КБ-2 «Прикладные информационные технологии»
	(полное и краткое наименование кафедры, реализующей дисциплину (модуль))
Лектор	к.т.н., Селин Андрей Александрович
	(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года 2024/2025

(учебный год цифрами)

Проверено и согласовано «____» _____2024 г. _____ А.А. Бакаев

(подпись директора Института/Филиала с расшифровкой)

Москва 2024 г.

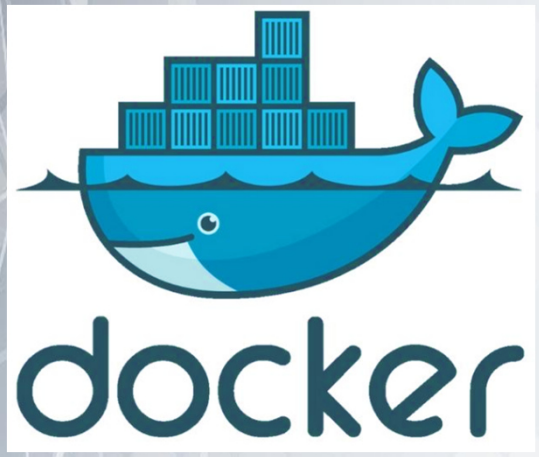
Технологии хранения в системах кибербезопасности

2024 год

Лекция 5.
Технологии виртуализации и контейнеризации

Учебные вопросы лекции:

1. Виртуализация и виртуальные машины
2. Контейнеры
3. Docker



Введение

Docker – популярная технология контейнеризации, появившаяся в 2013 году. Тогда одноименная компания предложила способ виртуализации ОС, при котором код приложения, среда запуска, библиотеки и зависимости упаковываются в единую «капсулу» – контейнер Docker.

Одной из реализаций идеи о разделении ресурсов стали Croot jail и операция Chroot, которые появились в 1979 году в UNIX версии 7. С помощью Chroot jail процесс и его дочерние элементы изолировались от основной ОС.

Спустя 20 лет появился FreeBSD Jail — механизм виртуализации, позволяющий внутри одной ОС использовать несколько изолированных систем, которые назвали тюрьмами. Далее технологии контейнеризации развивались стремительно.

В 2005 году представили OpenVZ с виртуализацией на уровне ОС, благодаря которой усовершенствовалась технология изоляции. Был и серьезный недостаток: у контейнеров и хоста была одна архитектура и версия ядра. Если требовалась другая, возникали проблемы.

В 2007 году компания Google представила функцию CGroups, ограничивающую использование ресурсов (CPU, ROM, дисковый ввод-вывод, сеть и т. д.) на уровне групп процессов. Спустя год выпустили Linux Containers (LXC), который имел много общего с OpenVZ и Linux-VServer, но использовал механизмы namespaces и CGroups из Linux-ядра вместо сторонних механизмов, внедряемых в ядро с помощью патчей.

В 2013 году компания Cloud Foundry создала Warden. Новая утилита предназначалась для запуска приложений, которые получают все свои зависимости от частей ПО, называемых buildpacks. Контейнеры Warden обычно имеют два слоя: слой только для чтения с корневой файловой системой ОС и неперсистентный слой чтения/записи самого приложения и его зависимостей. **В 2013 году на рынке появился и Docker.**

Введение

Docker – это платформа контейнеризации с открытым исходным кодом, с помощью которой можно автоматизировать создание приложений, их доставку и управление. Платформа позволяет быстрее тестировать и выкладывать приложения, запускать на одной машине требуемое количество контейнеров.

Благодаря контейнеризации и использованию Docker разработчики больше не задумываются о том, в какой среде будет функционировать их приложение и будут ли в этой в среде необходимые для тестирования опции и зависимости. Достаточно упаковать приложение со всеми зависимостями и процессами в контейнер, чтобы запускать в любых системах: Linux, Windows и macOS. Платформа Docker позволила отделить приложения от инфраструктуры. Контейнеры не зависят от базовой инфраструктуры, их можно легко перемещать между облачной и локальной инфраструктурами.

У контейнеризации и виртуализации есть сходство, но есть и различия. Виртуализация напоминает отдельный компьютер со своим оборудованием и ОС, внутри которого можно запустить еще одну ОС. А контейнеризация предполагает, что виртуальная среда запускается из ядра ОС, не предусматривает виртуализации оборудования и снижает потребление ресурсов.

Существуют решения вроде Yandex Serverless Containers которые позволяют запускать контейнеры без создания виртуальных машин и кластеров Kubernetes. Платформа избавляет пользователей от необходимости заниматься рефакторингом и отменяет ограничения на использование языков программирования.

Виртуализация и виртуальные машины

Виртуализация позволяет абстрагироваться от аппаратных средств (Hardware) для изоляции нескольких вычислительных процессов на одном компьютере. При этом, фактически, происходит следующее:

1. Имитируется физический объект;
2. Ресурсы этого объекта отделены от аппаратной части;
3. Появляется возможность внедрить несколько вычислительных процессов (операционных систем) на одной машине;
4. Все операции изолированы.

Виртуализация необходима в современной разработке по ряду причин:

1. Не нужно выделять физические серверы для серверной инфраструктуры под каждый отдельный случай (можно на одном сервере реализовать несколько);
2. Запуск программ под конкретную операционную систему (так, пользователь Windows без проблем установит программу под macOS и наоборот);
3. Необходимость организации виртуальных сетей (VPN). Они надежнее и безопаснее, особенно в корпоративных целях. Один из вариантов виртуализации - виртуальные машины.

Виртуализация и виртуальные машины

Основные свойства виртуальных машин:

1. **Инкапсуляция**. Виртуальный компьютер, операционная система, программные средства записываются на ваш ПК в виде отдельного набора файлов, которые легко переносятся;
2. **Изоляция**. Любой набор виртуальных машин является полностью независимым и не связан с другими;
3. **Совместимость** – виртуальные машины способны содержать разнообразные наборы операционных систем, оборудования и программ, и в то же время их легко реализовать на любом ПК;
4. **Независимость от железа**. Так как они запускаются в виртуальном окружении, то полностью независимы от устройств домашнего компьютера.

До того, как контейнеры стали популярными, для изоляции приложений использовались именно виртуальные машины (**VirtualBox**, **VMware**). Главные **недостатки виртуальных машин**:

- требуют много места в памяти;
- относительно долго загружаются;
- для их корректной работы необходима идентичность окружения (приходится устанавливать те же операционные системы и версии программ, которые имеются на тестовом оборудовании). Другими словами, следует полностью "клонировать" всю среду образа.

Контейнеры

Контейнер – изолированная среда выполнения с центральным процессором, памятью, блоками ввода и вывода и сетевыми ресурсами, использующая ядро гостевой операционной системы.

Один и тот же контейнер всегда работает одинаково, независимо от окружения, в котором он активирован. Контейнеры подобны реальным, в которые можно что-то положить, их легко переместить на другую машину. Они изолированы, друг с другом не «перемешиваются». Дополнительные установки не нужны, так как все уже упаковано внутри.

Поддержка контейнеров встроена в современные операционные системы. Docker упрощает создание и управление этими контейнерами. Таким образом, контейнеры позволяют создавать независимые, стандартизованные пакеты приложений.

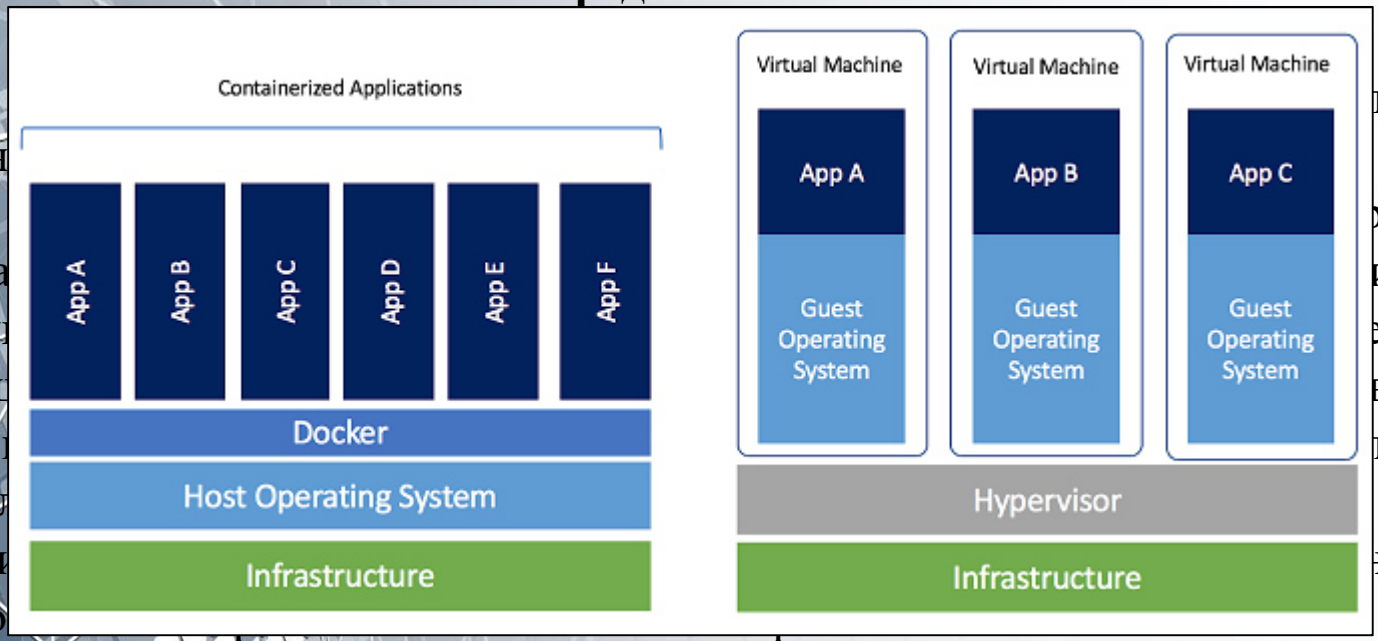
В настоящее время практически все операционные системы поддерживают контейнеры: – любая версия Linux; – версия macOS не ниже 10.14; – версия Windows 10 64-bit Pro или выше.

Особенности контейнеров

- Важнейшая особенность контейнеров – их сравнительно короткий жизненный цикл. Любой контейнер можно остановить, перезапустить или уничтожить, если это необходимо. Данные, которые содержатся в контейнере, при этом тоже пропадут. Так выработалось правило проектирования приложений: не хранить важные данные в контейнере. Такой подход называют Stateless.
- Объем контейнеров измеряется в мегабайтах, поскольку в них упаковывают лишь те процессы и зависимости ОС, которые необходимы для выполнения кода. Легковесные контейнеры быстро запускаются и экономят место на диске.
- Один контейнер соответствует одному запущенному процессу. Отключение отдельного контейнера для отладки или обновления никак не помешает нормальной работе всего приложения.
- Контейнеризация обеспечивает надежную изоляцию процессов и повышает уровень безопасности систем. Приложения, которые работают внутри контейнера, не имеют доступа к основной ОС и не могут на неё влиять.
- Благодаря контейнерам можно автоматизировать развертывание приложений на разных хостах.
- Использование контейнеров позволяет перейти с монолита на микросервисную архитектуру. За счет этого ускоряется разработка новой функциональности, поскольку нет опасений, что изменения в одной компоненте затронут всю остальную систему.
- С точки зрения эффективности контейнеры котируются выше виртуальных машин. На одинаковом оборудовании можно запустить большое количество контейнеров, тогда как ВМ будет в разы меньше. Это важно при использовании облачной инфраструктуры – потребуется меньше ресурсов.

Docker

Определения:



Docker ложений внутри кон

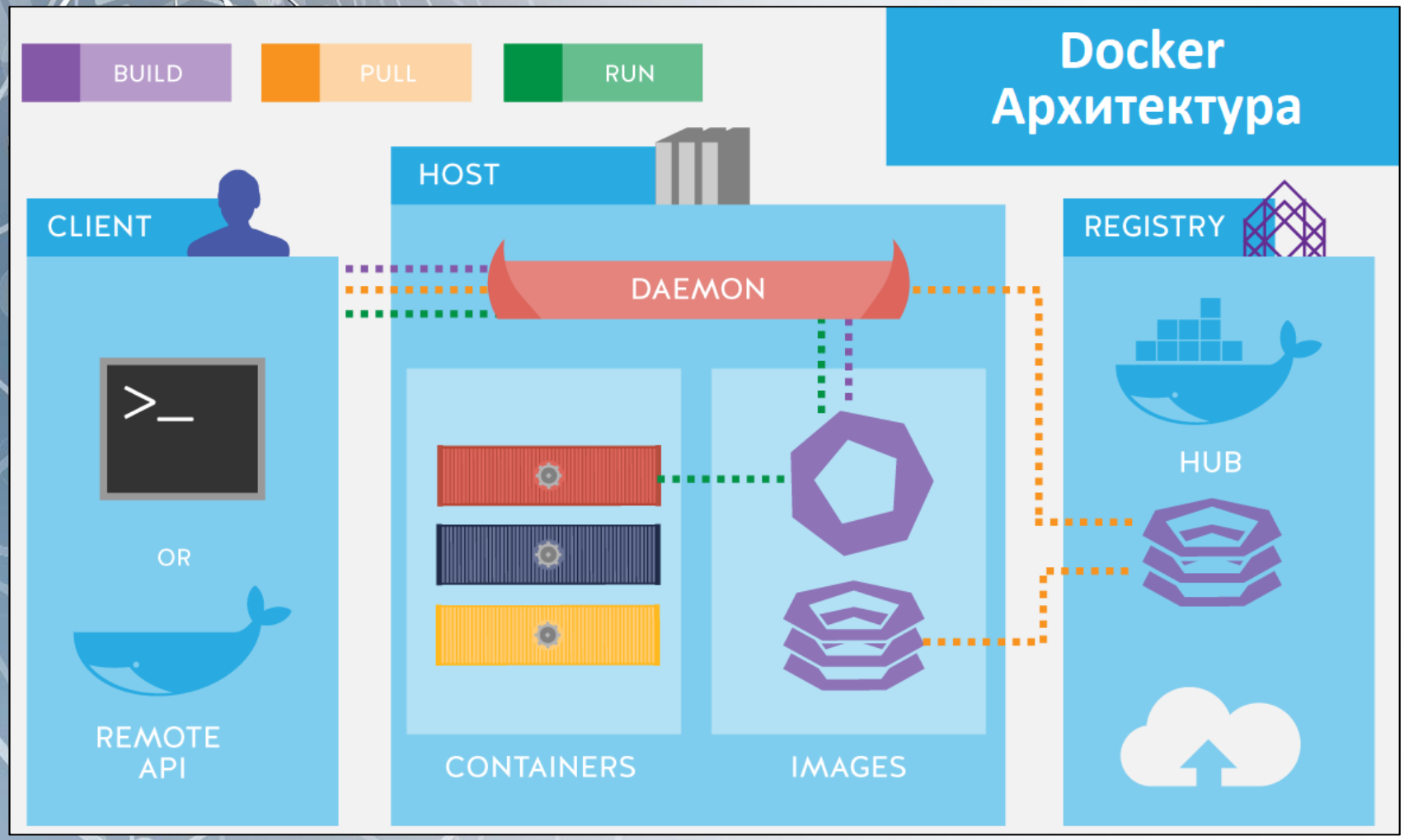
Docker роблемы развертыва имостей инфрастру ейнеров, позволяю включая структуру портов и т. д. При любых машинах нологию Docker осо

Docker – это инструмент с открытым исходным кодом, который позволяет вам включать и хранить ваш код и его зависимости в удобном пакете, который называется *образом*. Затем этот образ можно использовать для создания экземпляра вашего приложения (сервиса) – *контейнера*. Основное различие между контейнерами и виртуальными машинами заключается в том, что контейнеры охватывают только уровень приложения и полагаются на базовое ядро операционной системы, в случае виртуальной машины создается новый экземпляр операционной системы.

Docker

Сущности Docker: docker daemon, container, image, Dockerfile, Docker Registry

В Docker используется архитектура клиент/сервер, в соответствии с которой клиент взаимодействует с **docker daemon**, а тот предоставляет все необходимые клиенту услуги.



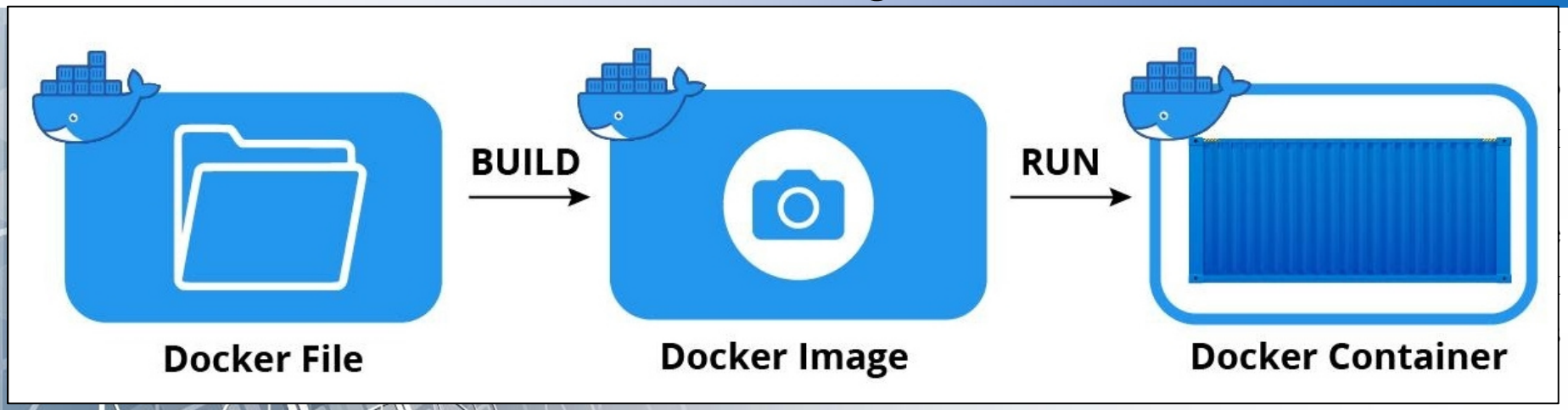
Компоненты экосистемы Docker

- **Daemon Docker** (Сервер): Выполняется в хост-системе и управляет всеми запущенными контейнерами. Docker Daemon – это сервер Docker, который прослушивает запросы Docker API. Docker Daemon управляет образами, контейнерами, сетями и томами.
- **Docker Container** (Контейнер): Автономная виртуальная система, содержащая выполняющийся процесс, все файлы, зависимости, адресное пространство процесса и сетевые порты, необходимые приложению. Так как каждый контейнер имеет свое пространство портов, следует организовать их отображение в фактические порты на уровне Docker;
- **Docker Client** (Клиент): Пользовательский интерфейс, или интерфейс командной строки, для взаимодействий с Daemon Docker. Клиент Docker – это основной способ взаимодействия с Docker. Когда вы используете интерфейс командной строки (CLI) Docker, вы вводите в терминал команду, которая начинается cdocker. Затем клиент Docker использует API Docker для отправки команды Daemon Docker.
- **Docker Image** (Образ): это шаблон только для чтения, который содержит набор инструкций по созданию контейнера, который может работать на платформе Docker. Он предоставляет удобный способ упаковать приложения и предварительно настроенные серверные среды, которые вы можете использовать для личного использования или публично публиковать с другими пользователями Docker. Также можно воспользоваться командой docker diff, чтобы увидеть различия между двумя образами. Каждый образ состоит из нескольких уровней, или слоев, которые могут совместно использоваться несколькими образами.

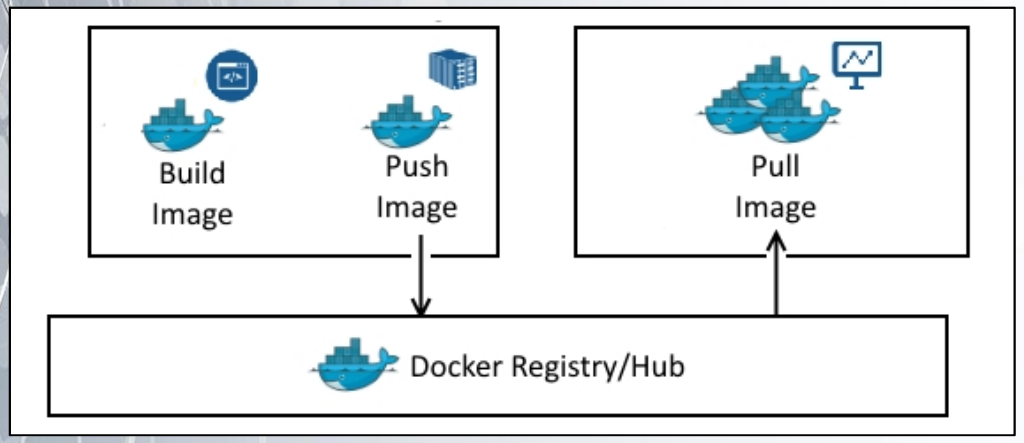
Компоненты экосистемы Docker

- **Реестр Docker**: Репозиторий для хранения и распространения образов контейнеров Docker. Пример известного реестра – Docker Hub, куда можно помещать и откуда можно извлекать образы.
- **Dockerfile**: Это очень простой текстовый файл, содержащий команды, которые выполняют сборку образов Docker. Посредством этих команд можно устанавливать дополнительные программные компоненты, настраивать переменные окружения, рабочие каталоги и точку входа ENTRYPOINT, а также добавлять новый код;
- **Docker Swarm**: По сути своей, это готовый к использованию механизм кластеризации, позволяющий объединить несколько узлов Docker в один большой хост Docker.
- **Docker Compose**: Приложения часто состоят из множества компонентов, и соответственно они будут выполняться в нескольких контейнерах. В состав Docker входит инструмент Compose, с помощью которого можно легко запустить приложение в нескольких контейнерах. Вы можете определить окружение для приложения в общем файле Dockerfile и определить перечень служб в файле docker-compose.yml, после чего Docker автоматически будет создавать и запускать необходимые контейнеры, как определено в этих файлах.

Docker image



Интерактивный: запустив контейнер из существующего образа Docker, вручную изменив среду контейнера с помощью серии активных шагов и сохранив полученное состояние как новый образ.



Dockerfile: путем создания текстового файла, известного как Dockerfile, который предоставляет спецификации для создания образа Docker.

Dockerfile – это файл с инструкциями о том, как **Docker** должны строить свой **image**

Docker image

С физической точки зрения docker image состоит из набора слоев, доступных только для чтения (read-only layers). Слои image работают следующим образом:

- Каждый слой image является результатом одной команды в файле Dockerfile. Образ докера представляет собой сжатый (tar) файл, содержащий серию слоев.
- Каждый дополнительный слой image включает только набор отличий от предыдущего слоя (попробуйте запустить для docker image команду docker history, которая выведет все его слои и те команды, которые их создало).

Пример Dockerfile:

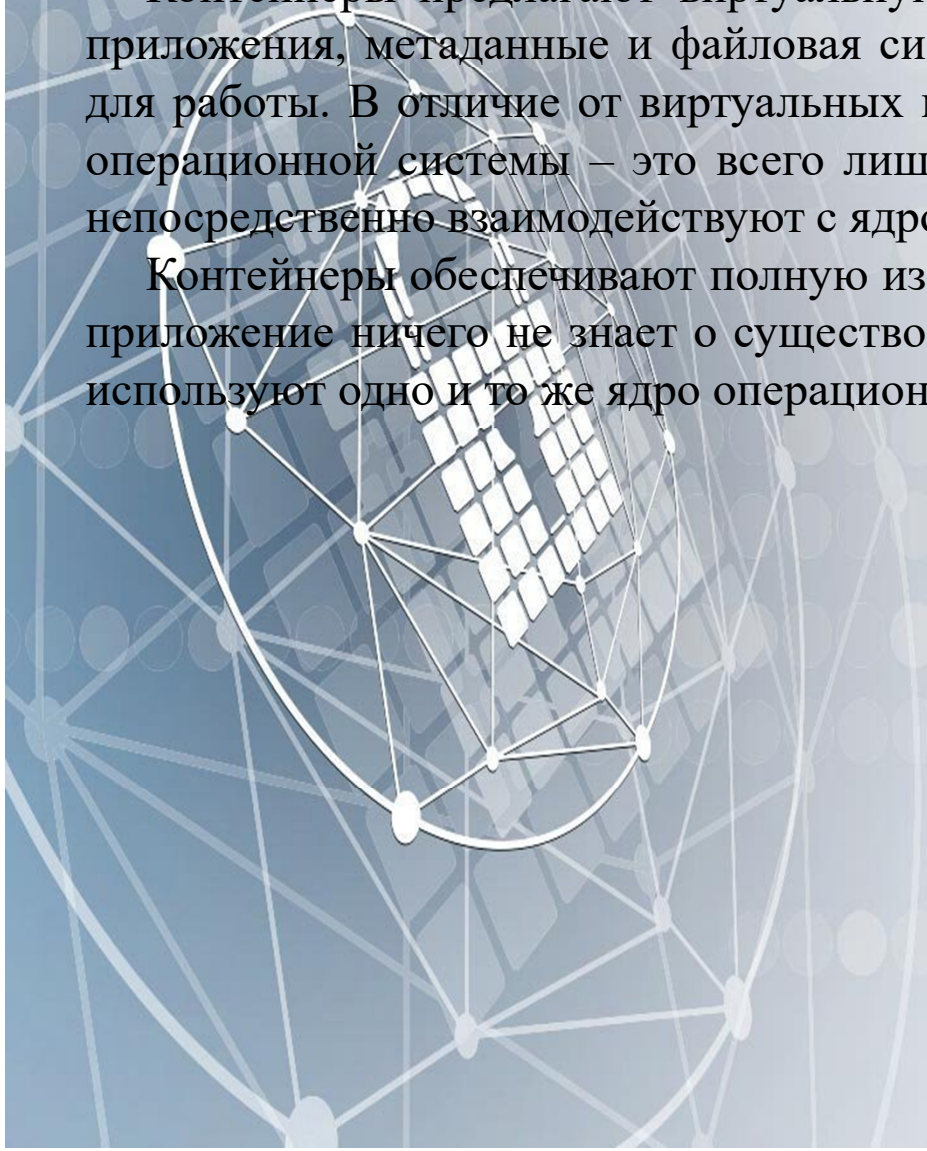
```
FROM node:13.12.0-alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
```

Docker container

Docker Container – это исполняемый экземпляр образа. Образ Docker плюс команда docker run image_name создает и запускает контейнер из образа.

Контейнеры предлагают виртуальную среду, в которую упаковываются процесс приложения, метаданные и файловая система, т.е. все, что необходимо приложению для работы. В отличие от виртуальных машин, контейнеры не требуют собственной операционной системы – это всего лишь обертки вокруг процессов UNIX, которые непосредственно взаимодействуют с ядром.

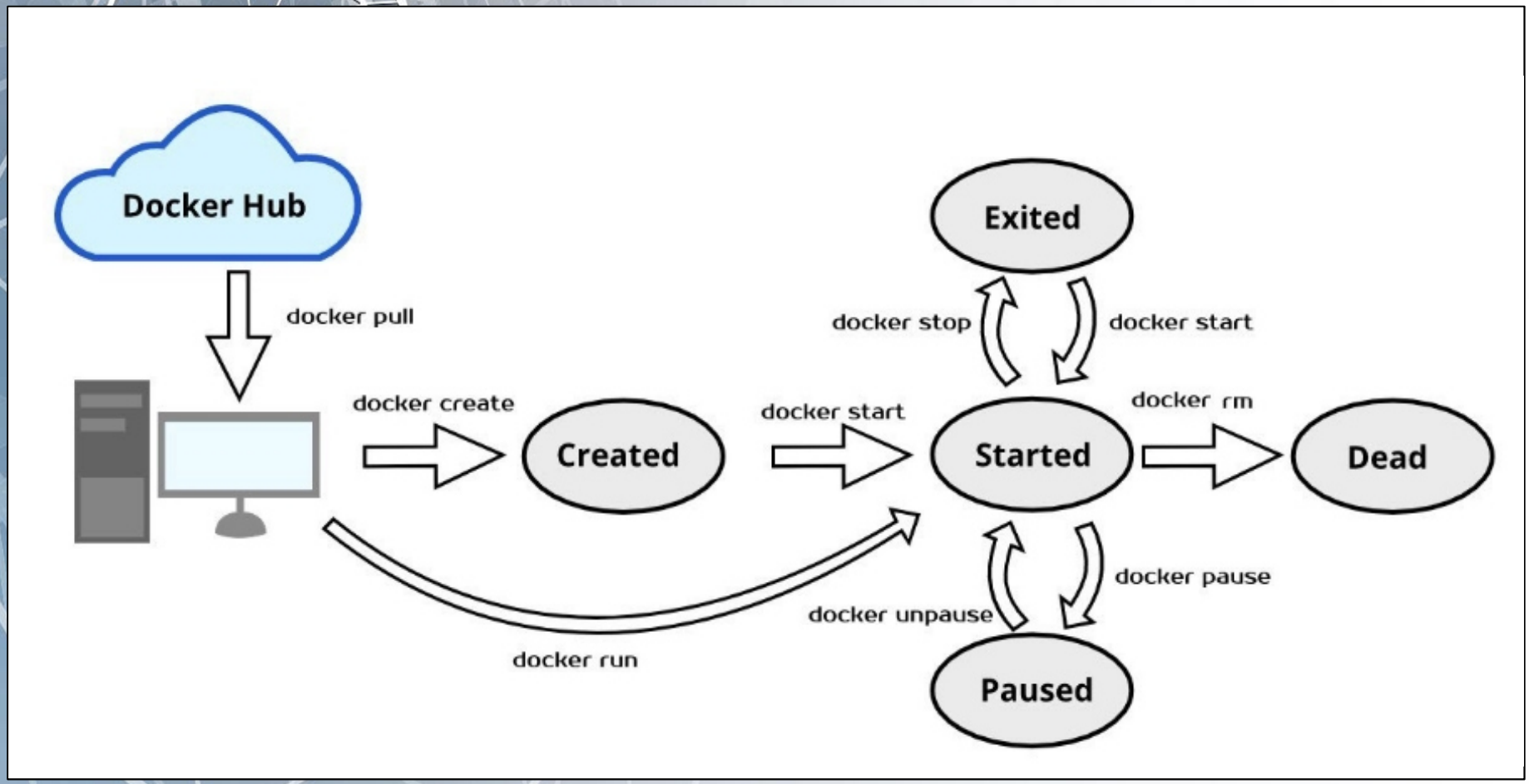
Контейнеры обеспечивают полную изоляцию приложений и процессов, когда одно приложение ничего не знает о существовании других приложений. Но все процессы используют одно и то же ядро операционной системы.



Управление контейнерами

Схема Lifecycle of Docker Container

Образ Docker – это статическая модель, содержащая предустановленное приложение. При запуске этого образа создается контейнер. Образ можно использовать для запуска любого количества контейнеров. На следующей диаграмме показан жизненный цикл контейнера Docker и связанных команд.



Управление контейнерами

Restart container

Используется для перезапуска контейнера, а также процессов, работающих внутри контейнера.

docker restart <container-id/name>

Kill container

Мы можем убить работающий контейнер.

docker kill <container-id/name>

Destroy container

Лучше уничтожать контейнер, только если он находится в остановленном состоянии, вместо того, чтобы принудительно уничтожать работающий контейнер.

docker rm <container-id/name>

Чтобы удалить все остановленные контейнеры докеров

docker rm $ (docker ps -q -f status = exited)

Start container

Запустите контейнер, если он находится в остановленном состоянии.

docker start <container-id/name>

Stop container

Остановить контейнер и процессы, запущенные внутри контейнера:

docker stop <container-id/name>

Чтобы остановить все запущенные контейнеры докеров

docker stop $(docker ps -a -q)

Dockerfile

Dockerfile – это текстовый файл конфигурации, написанный с использованием специального синтаксиса.

В нем описываются пошаговые инструкции по всем командам, которые необходимо выполнить для сборки образа Docker.

Команда docker build обрабатывает этот файл, создавая образ Docker в вашем локальном кэше образов, который затем можно запустить с помощью docker run команды или отправить в постоянный репозиторий образов (push).

```
# Каждая инструкция в этом файле генерирует новый слой, который помещается в ваш локальный кеш docker image
# Строки, которым предшествует #, считаются комментариями и игнорируются
 # В строке ниже указано, что мы будем основывать наш новый образ на последней официальной версии Ubuntu
FROM ubuntu: latest
 # Указываем, кто поддерживает docker image (автор)
LABEL Maintainer = "myname@somecompany.com"
 # Обновляем образ до последних пакетов
RUN apt-get update && apt-get upgrade -y
 # Устанавливаем NGINX
RUN apt-get install nginx -y
 # Выставить порт 80
EXPOSE 80
 # Последний пункт - это команда для запуска NGINX в нашем контейнере
CMD ["nginx", "-g", "daemon off;"]
```

Dockerfile. Команды

RUN – команды для внесения изменений в ваш образ, а затем контейнеры, запускаемые с этого образа. Включает в себя обновление пакетов, установку программного обеспечения, добавление поль ификатов и т. д. Это команды, я ля установки и настройки ваше e

RUN apt-get upd /var/lib/apt/lists/*

USER – Опред пускаться, как в любом контейн UID, либо имя пользователя.

Пример dockerfile для приложения flask app python

```
FROM python:3.6.4-alpine3.6
ENV FLASK_APP=minitwit
COPY . /app
WORKDIR /app
RUN pip install -- editable .
RUN flask initdb
EXPOSE 5000
CMD [ "flask", "run", "--host=0.0.0.0" ]
```

VOLUME – создает точку монтирования в контейнере, связывая ее с файловыми системами, доступными хосту Docker. Новые тома заполняются уже существующим содержимым указанного места на image. Особенно уместно упомянуть, что определение томов в Dockerfile может привести к проблемам. Томами следует управлять с помощью команд docker-compose или docker run. Volumes не являются обязательными. Если у вашего приложения нет состояния (и большинство веб-приложений работают так), вам не нужно использовать тома. Volume требуются для баз данных, например, или для сохранения логов приложения.

VOLUME /var/log

WORKDIR – Определите рабочий каталог по умолчанию для команды, определенной в инструкциях «ENTRYPOINT» или «CMD».

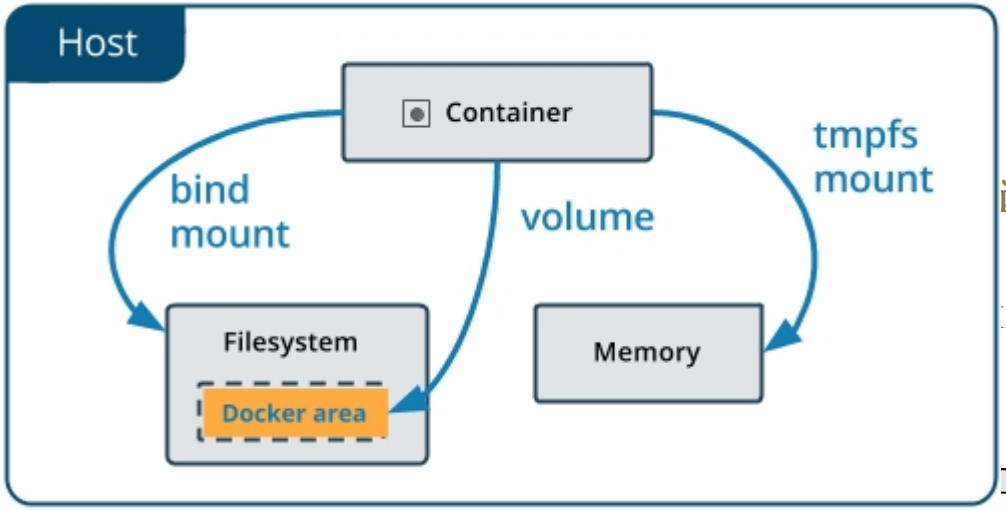
WORKDIR /home

Docker Volume

В docker-compose.yml, volumes может появляться в двух разных местах:

```
version: "3"
 services:
   database:
      # ...
      volumes: # Влож                                    й службы.

 volumes: # Ключ ве                                     о ссылаться из
нескольких сервисо
  # ...
```



Команды Docker Volume

Команда	Описание
docker volume create	Create a volume
docker volume inspect	Display detailed information on one or more volumes
docker volume ls	List volumes
docker volume prune	Remove all unused local volumes
docker volume rm	Remove one or more volumes

вне жизненного цикла данного контейнера.

Взаимодействие с контейнером

Docker не открывает порты по умолчанию, каждый открытый порт необходимо настроить самостоятельно…

Чтобы сопоставить порт на хосте с контейнером, нам нужно использовать флаг -p команды docker run:

docker run -p <port_number_on_host>:<port_number_on_container> <image>

или

docker run -v <порт_на_хосте>:<порт_в_контейнере> <образ>

Следующая команда запустит контейнер для mlflow и сопоставит порт 7000 этого контейнера с портом 7000 хоста докера:

docker run -p 7000:7000 mlflow

Аналогичным образом будет работать контейнер для POSTGRESQL и порта 5432 этого контейнера в порт 5432 от DOCKER хоста:

docker run -p 5432:5432 postgresql

Из-за того, что многие контейнеры связаны с одним хостом, когда служба уже запущена на порту на хосте, мы не можем запустить другой контейнер с этим же портом. Следовательно, чтобы запустить другой экземпляр postgresql на хосте, мы должны выбрать порт, который не используется.

docker run -p 1234:5432 postgresql

Чтобы открыть только один порт, выполните следующую строку:

docker container run -p 8080:80 -d nginx

Порт 80 контейнера Nginx доступен для внешнего мира через порт хоста 8080.

Чтобы привязать порт 80 контейнера Docker к порту 8000 хост-системы и IP-адресу 127.0.0.1 (он же localhost), просто выполните следующую команду:

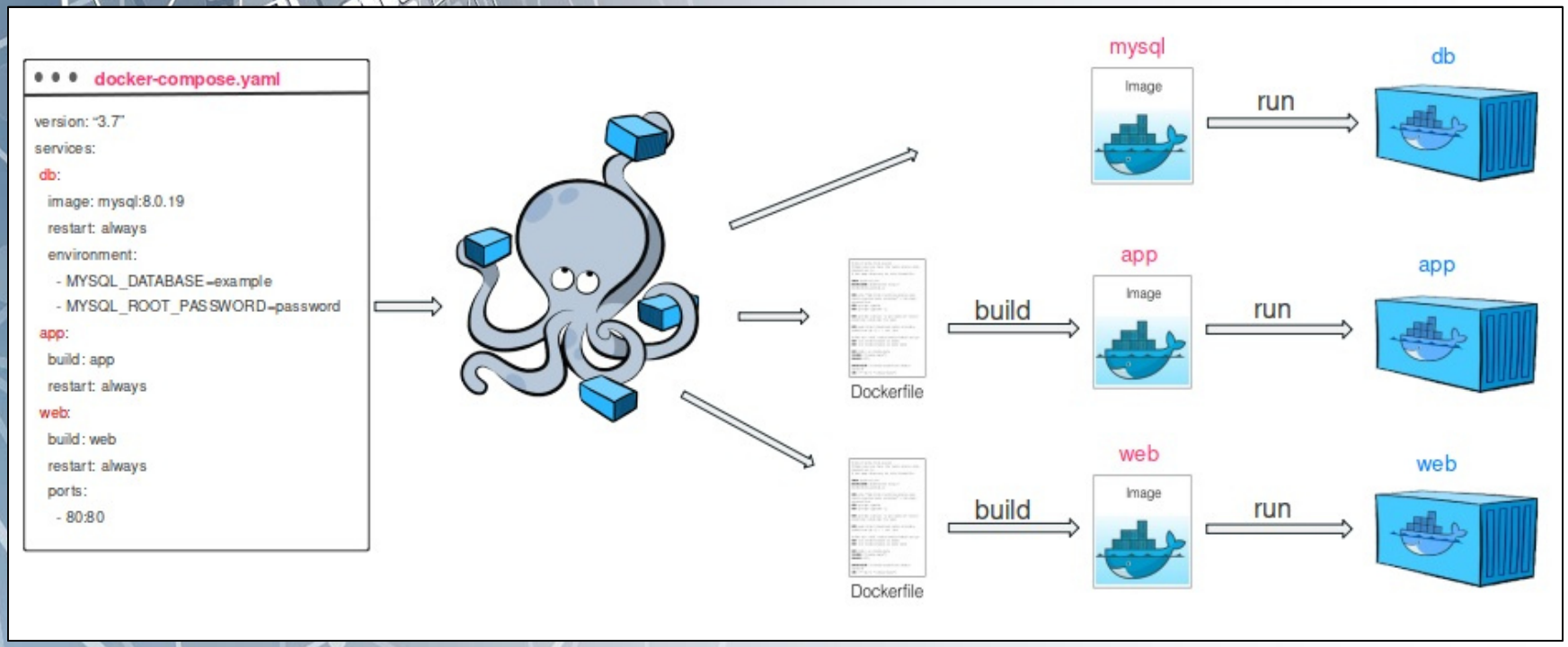
docker run -d -p 127.0.0.1:8000:80 nginx

Docker Compose

Docker Compose – это инструмент, который упрощает запуск приложений, состоящих из нескольких контейнеров.

Docker Compose позволяет записывать команды в docker-compose.yml файл для повторного использования. Интерфейс командной строки Docker Compose (cli) упрощает взаимодействие с вашим многоконтейнерным приложением.

Docker Compose управляет контейнерами, запускает их вместе, в нужной последовательности, необходимой для вашего приложения. Его можно назвать дирижёром в мире Docker-а.



Docker Compose

Docker-compose написан в формате YAML который по своей сути похож на JSON или XML. Но YAML имеет более удобный формат для его чтения, чем вышеперечисленные. В формате YAML имеют значения пробелы и табуляции, именно пробелами отделяются названия параметров от их значений.

Пример создания нового файла docker-compose.yml, для рассмотрения синтаксиса Docker Compose:

```
version: '3'

services:
  app:
    build:
      context: .
    ports:
      - 8080:80
```

version: какая версия docker-compose используется.
services: контейнеры которые мы хотим запустить.
app: имя сервиса, может быть любым, но желательно, чтобы оно **описывало суть этого контейнера.**
build: шаги, описывающие процесс билдинга.
context: где находится Dockerfile, из которого будем билдить образ для контейнера.
ports: маппинг портов основной ОС к контейнеру.

Можно использовать этот файл для билдинга нашего предыдущего образа apache:

```
docker-compose build
```

После выполнения этой команды, Docker спарсит файл docker-compose и создаст описанные сервисы на основе инструкций во вкладке build.

Docker Compose

A context говорит о том, из какой директории мы берём Dockerfile для создания образа сервиса (в текущем случае – это означает текущую директорию, но могло быть и /php-cli, /nginx, и т.д.).

Запуск созданных сервисов: `docker-compose up`

В результате чего, сервер должен был запуститься, и стать доступным по адресу localhost:8080.

Когда контейнер под названием app запускается, docker-compose автоматически связывает указанные порты во вкладке ports. Docker-compose избавляет нас от указания параметров в командной строке напрямую.

С docker-compose.yml мы переносим все параметры, ранее записываемые в командной строке при запуске контейнера в конфигурационный YAML-файл.

В этом примере мы записали BUILD и RUN шаги для нашего сервиса в docker-compose.yml. Преимущество такого подхода в том, что теперь для билдинга и для запуска этих сервисов необходимо запомнить только 2 команды: docker-compose build и docker-compose up. При таком подходе не нужно помнить, какие аргументы нужно указывать, какие опции нужно задавать при запуске контейнера.

Оркестрация

Когда контейнеров становится слишком много, ими трудно управлять. На помощь приходят системы оркестрации.

Docker Swarm

Стандартная система оркестрации контейнеров, достаточная для решения базовых задач. Позволяет быстро создать из нескольких хостов с контейнерами последовательный кластер Swarm, считая все кластерные хосты единым контейнерным пространством. В Docker-кластере должна быть как минимум одна управляющая нода (manager).

Kubernetes

Платформа для автоматизации работы с контейнерами на Ubuntu, CentOS и других ОС Linux. Позволяет централизованно группировать контейнеры, балансировать нагрузку, активировать сервисы из сотен приложений одновременно. Kubernetes предоставляет пользователям больше возможностей по сравнению со Swarm, но и настраивать его сложнее.

Преимущества использования Docker

1. **Гибкость и адаптивность.** Благодаря Docker можно легко запускать контейнер в облачной инфраструктуре и на любом локальном устройстве.
2. **Меньше ошибок и несовпадений окружений.** В контейнерах Docker содержится всё, что требуется для запуска приложения, поэтому перенос приложений из одной среды в другую не вызывает затруднений.
3. **Скорость развертывания.** Так как настраивать окружение для разработки, тестирования и боевого режима больше не нужно, время развертывания сокращается в несколько раз.
4. **Рост универсальности.** Docker позволяет использовать любые языки программирования и стек технологий на сервере, избавляя от проблемы несовместимости разных библиотек и технологий.
5. **Комьюнити и поддержка.** Существует огромная библиотека контейнеров с открытым исходным кодом. Можно скачать нужный образ для конкретной задачи или обратиться за помощью к большому комьюнити разработчиков, которые используют Docker.
6. **Непрерывность работы.** С учетом инструментов управления трафиком можно построить процесс обновления приложения так, чтобы обновление одних контейнеров не влияло на работоспособность системы и оказание услуг пользователям.
7. **Упрощение администрирования.** С помощью Docker легче перенести контейнер с одного хоста на другой, запустить сразу несколько образов, обновить группы контейнеров и откатиться к старой версии.
8. **Повышение уровня безопасности.** Контейнеры в Docker частично изолированы друг от друга на уровне процессов и ОС, поэтому запуск большого количества контейнеров на одной машине не несет рисков.
9. **Экономическая эффективность.** Контейнеры легковесны и производительны, а благодаря использованию Docker можно эффективнее управлять имеющимися ресурсами и сократить расходы компании.
10. **Современный подход.** Отказ от монолитной архитектуры в пользу микросервисной позволяет более гибко развивать продукт, добавлять в него новые функции.

Полезные ссылки

- Документация на Docker
  https://docs.docker.com/
- Краткая шпаргалка
  https://www.docker.com/sites/default/files/d8/2019-09/docker-cheat-sheet.pdf
- Полезные команды
  https://github.com/wsargent/docker-cheat-sheet
- Настройка локального рабочего окружения
  https://www.notion.so/joomteam/Local-Workspace-e0356b5824554152a15e964eb15be032

Заключение

Docker стандартизирует среду разработки и позволяет запускать приложения в едином окружении на любом компьютере. На основании образов (как готовых, так и личных) можно активировать требуемое количество контейнеров. Для этого не нужно думать ни о типе операционной системы, ни о зависимостях.

СПАСИБО ЗА ВНИМАНИЕ!