Если вы видите что-то необычное, просто сообщите мне.

# Docker

- Перенос docker на отдельный раздел.
- Примеры команд Docker.
- Docker - инструмент создания инфраструктуры.
- Dockerfile, создание docker image.
- Containerd

# Перенос docker на отдельный раздел.

# Первый вариант

## Останавливаем docker

```
systemctl stop docker
systemctl stop docker.socket
```

## Обновляем конфигурацию daemon

```
echo '{ "data-root": "/new_dir_structure/docker" }' > /etc/docker/daemon.json
```

## Запускаем docker

```
systemctl start docker
```

# Второй вариант

# Создаем отдельный раздел для docker

```
тут должен быть код для создания отдельного раздела на диске\
можно воспользоваться инструментами типа  gparted
```

# Останавливаем docker

```
systemctl stop docker
systemctl stop docker.socket
```

# Прописываем fstab строчку, чтобы монтирование происходило в /var/lib/docker

Этот шаг может быть выполнен в момент создания раздела, если использовать графическую оболочку.

```
/dev/disk/by-uuid/ID-УСТРОЙСТВА /var/lib/docker auto nodev,nofail 0 0
```

или

```
/dev/sda(НОМЕР_УСТРОЙСТВА) /var/lib/docker auto nodev,nofail 0 0
```

# Запускаем docker

```
systemctl start docker
```

# Проверяем что всё работает

```
docker run nginx
```

# Если возникает ошибка

Можно встретить вот такую ошибку:

```
/bin/sh: error while loading shared libraries: /lib/x86_64-linux-gnu/libc.so.6: cannot apply additional memory protection after relocation: Permission denied
```

для решения можно воспользоваться вот такой командой:

```
chcon -Rt svirt_sandbox_file_t /var/lib/docker
```

# Примеры команд Docker.

## Команда выводит общую информацию

Состояние docker-engine:

```
sudo docker info
```

Статистику docker:

```
sudo docker stats
```

Информацию о кешированных образах:

```
sudo docker images
```

## Команда для выкачивания оригинального образа из облака

```
sudo docker pull alpine
```

По умолчанию лезет на hub.docker.com Если вы знаете, что есть какое-то свое облако, то адрес необходимо указывать полностью:

```
sudo docker pull dockerrepo.vasya.com/proect/front
```

## Запуск проекта

```
sudo docker run -i -t alpine /bin/bash
```

В этом случае, мы указываем, что нам необхдимо запустить проект `alpine` в интерактивном режиме, и внутри контейнера запустить процесс `/bin/bash`. В этом случае, если локально

нет образа, то система сначала выполнит `docker pull` и только затем уже его запустит.

# Управленние запущенными контейнерами

Запуск контейнера с определенным именем.

```
sudo docker run —-name our_container -it ubuntu /bin/bash
```

Если мы выйдем из контейнера, то он прекратит свою работу. Для того, чтобы запустить его снова, можно воспользоваться этой командой:

```
sudo docker start container_name
```

Для того, чтобы остановить контейнер, используйте эту команду:

```
sudo docker stop container_name
```

# Docker - инструмент создания инфраструктуры.

Docker (Докер) — программное обеспечение с открытым исходным кодом, применяемое для разработки, тестирования, доставки и запуска веб-приложений в средах с поддержкой контейнеризации. Он нужен для более эффективного использование системы и ресурсов, быстрого развертывания готовых программных продуктов, а также для их масштабирования и переноса в другие среды с гарантированным сохранением стабильной работы.

# Преимущества использования Docker

- Минимальное потребление ресурсов
- Скоростное развертывание
- Удобное скрытие процессов
- Работа с небезопасным кодом
- Простое масштабирование
- Удобный запуск
- Оптимизация файловой системы

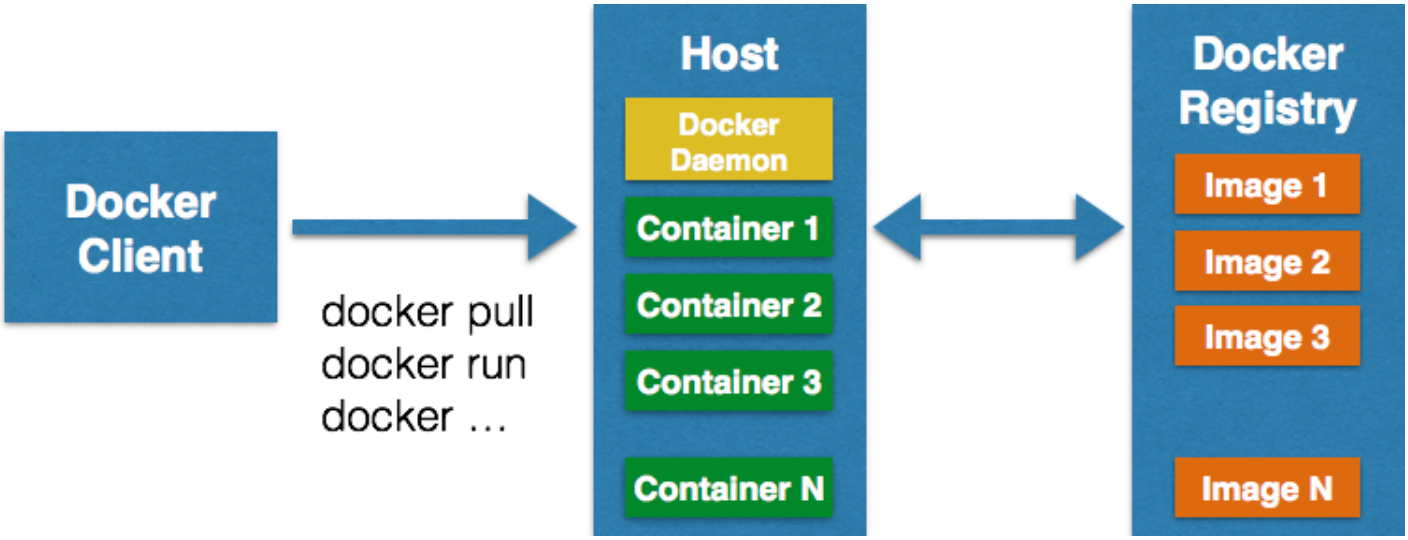
# Компоненты Docker

- Docker-демон (Docker-daemon) — сервер контейнеров, входящий в состав программных средств Docker. Демон управляет Docker-объектами (сети, хранилища, образы и контейнеры). Демон также может связываться с другими демонами для управления сервисами Docker.

- Docker-клиент (Docker-client / CLI) — интерфейс взаимодействия пользователя с Docker-демоном. Клиент и Демон — важнейшие компоненты «движка» Докера (Docker Engine). Клиент Docker может взаимодействовать с несколькими демонами.
- Docker-образ (Docker-image) — файл, включающий зависимости, сведения, конфигурацию для дальнейшего развертывания и инициализации контейнера.
- Docker-файл (Dockerfile) — описание правил по сборке образа, в котором первая строка указывает на базовый образ. Последующие команды выполняют копирование файлов и установку программ для создания определенной среды для разработки.
- Docker-контейнер (Docker-container) — это легкий, автономный исполняемый пакет программного обеспечения, который включает в себя все необходимое для запуска приложения: код, среду выполнения, системные инструменты, системные библиотеки и настройки.
- Docker-volume — эмуляция файловой системы для осуществления операций чтения и записи. Она создается автоматически с контейнером, поскольку некоторые приложения осуществляют сохранение данных.
- Реестр (Docker-registry) — зарезервированный сервер, используемый для хранения docker-образов. Примеры реестров:
- Docker-хост (Docker-host) — машинная среда для запуска контейнеров с программным обеспечением.
- Docker-сети (Docker-networks) — применяются для организации сетевого интерфейса между приложениями, развернутыми в контейнерах.

# Как работает Docker

Работа Docker основана на принципах клиент-серверной архитектуры, которая основана на взаимодействии клиента с веб-сервером (хостом). Первый отправляет запросы на получение данных, а второй их предоставляет.




# Схема работы

1. Пользователь отдает команду с помощью клиентского интерфейса Docker-демону, развернутому на Docker-хосте. Например, скачать готовый образ из реестра (хранилища Docker-образов) с помощью команды `docker pull`. Взаимодействие между клиентом и демоном обеспечивает REST API. Демон может использовать публичный (Docker Hub) или частный реестры.
2. Исходя из команды, заданной клиентом, демон выполняет различные операции с образами на основе инструкций, прописанных в файле `Dockerfile`. Например, производит их автоматическую сборку с помощью команды `docker build`.
3. Работа образа в контейнере. Например, запуск docker-image, посредством команды `docker run` или удаление контейнера через команду `docker kill`.

# Примеры применения

- Быстрая доставка приложений (команды docker pull и docker push) позволяет организовать коллективную работу над проектом. Разработчики могут работать удаленно на локальных компьютерах и выполнять пересылку фрагментов кода в контейнер для тестов.

- Развертывание и масштабирование — контейнеры работоспособны на локальных компьютерах, серверах, в облачных онлайн-сервисах. Их можно загружать на хостинг для дальнейшего тестирования, создавать (docker run), останавливать (docker stop), запускать (docker start), приостанавливать и возобновлять (docker pause и docker unpause соответственно).
- Множественные нагрузки — осуществление запуска большого количества контейнеров на одном и том же оборудовании, поскольку Docker занимает небольшой объем дисковой памяти.
- Диспетчер процессов — возможность мониторинга процессов в Docker посредством команд docker ps и docker top, имеющими схожий синтаксис с Linux.
- Удобный поиск — в реестрах Docker он осуществляется очень просто. Для этого следует использовать команду docker search.

# Установка докера

```
#!/bin/bash
apt update
apt install -y docker.io
groupadd docker
usermod -aG docker $USER
```

Две последние команды необходимы для того, чтобы текущий пользователь мог управлять контейнерами, не запрашивая повышенных привелегий.

# Управлением демоном

Запуск демона:

```
systemctl start docker
```

Остановка демона:

```
systemctl stop docker
```

Включение/отключение автозагрузки демона:

```
systemctl enable/disable docker
```

При добавлении ключа `--now` - выполняет команду не дожидаясь перезагрузки. В противном случае, команда выполнится только после перезагрузки.

# Dockerfile, создание docker image.

# Dockerfile

Представим, приложение уже работает на вашей машине, но еще не имеет образа. Для того, чтобы получить docker образ, нам необходимо описать его.

# Создадим инструкцию для сборки образа

В папке с проектом(мы рассматриваем node проект) создадим файлы:

```
touch Dockerfile
```

Для того, чтобы в докер не попапи ненужные файлы - рядом создадим `dockerignore` и впишем в него всё, что нам не нужно.

```
touch .dockerignore
```

Пример Dockerfile:

```
#Выберем базовый образ из которого мы будем создавать наш проект
FROM node:10-apline
# Укажем папку внутри докера, которая будет являться домашней при выполнении различных команд
WORKDIR /usr/src/app
# Скопируем из папки с проектом все файлы по маске package*.json в папку WORKDIR
COPY package*.json ./
```

```
# Запускаем установку пакетов npm
RUN npm install
# Копируем всё что осталось в папке с проектом в папку WORKDIR
COPY . .
# Указываем порт, через который будет доступен наш проект
EXPOSE 3000
# Указываем команду которая будет работать в тот момент, когда запустится контейнер
CMD["npm", "start"]
```

# Список ключей используемых при создании образа

| Ключ | Назначение |
|---|---|
| FROM | Указываем на основе которого из образов будем создавать новый образ |
| MAINTAINER | Указываем автора созданного образа |
| RUN | Запукаем команды в внутри контейнера необхдимые для работы образа |
| CMD | Команда которая будет выполнена при запуске контейнера(может быть только одна) |
| EXPOSE | Говорит докеру, что контейнер слушает на определенном порту |
| ENV | Указываем с каким переменным окружением необходимо создавать контейнер |
| ADD | Копирует файлы, папки, URL и добавляет их в файловую систему образа, может распаковать архив |
| COPY | Копирует файлы, папки из и по указанному пути |
| ENTRYPOINT | Позволяет задавать команду запуска контейнера, при этом при старте указывать ключи запуска этого контейнера, переопределяет CMD |
| USER | Указывает имя пользователя под которым будет происходить выполнение команд RUN, CMD, ENTRYPOINT внутри контейнера |
| WORKDIR | Указываем директорию внутри которой будет происходить выполнение команд RUN, CMD, ENTRYPOINT, COPY, ADD |
| ARG | Определяем переменные, которые могут быть переданы билдеру при запуске `docker build` используя ключ `--build-arg <varname>=<value>` |

# Билдим проект

Создаем билд с помощью команды:

```
docker build . -t firstimage
```

Ожидаем заверешения команды. Для этого может потребоваться много времени.

```
В случае, если билд завершился неудачно, смотрим что за ошибка и правим Dockerfile
```

После завершения: мы можем посмотреть на наш `firstimage` в списке, который можно получить с помощью команды:

```
docker image
```

# Запуск нашего проекта

Чтобы запустить наш проект - необходимо указать несколько параметров:

```
docker run -p 80:3000 firstimage
```

-p - говорит о том, что мы мапируем(прим. ред. mapping - сопоставление) системный 80 порт в порт(помните внутри контейнера использовали EXPOSE 3000) внутри контейнера.

Заходим на http://localhost и убеждаемся, что наше приложение доступно внутри контейнера.

# Containerd

В качестве альтернативы можно воспользоваться другим решением.

# Запуск

Вот пример того, как можно запустить контейнер с использованием ctr:

Сначала убедитесь, что у вас есть образ контейнера, который вы хотите запустить. Если у вас его нет, вы можете сначала его загрузить с помощью ctr командой, например:

```
ctr images pull docker.io/library/alpine:latest
```

Затем запустите контейнер с помощью ctr run команды, указав необходимые параметры, например:

```
ctr run --rm -t --net-host docker.io/library/alpine:latest my-container sh
```

Здесь:

- `--rm` - указывает на то, что контейнер должен быть удален после завершения работы
- `-t` - выделяет псевдотерминал для контейнера
- `--net-host` - позволяет контейнеру использовать сеть хоста
- `docker.io/library/alpine:latest` - образ контейнера, который мы будем использовать
- `my-container` - имя контейнера
- `sh` - команда, которая будет выполнена внутри контейнера (в данном случае, запуск оболочки sh) Таким образом, вы сможете запустить контейнер на хосте только с помощью containerd. □□

# Сборка

Для сборки образа с помощью containerd вам обычно требуется создать и сконфигурировать контейнер с помощью ctr CLI (Command Line Interface). Вот пример того, как можно собрать образ с использованием containerd:

Установите containerd и ctr CLI на вашем хосте. Создайте конфигурационный файл для контейнера (например, config.toml):

```
[
process
]
args = ["echo", "Hello, World!"]
```

Создайте снимок (snapshot) с этим содержимым:

```
ctr snapshot create my-snapshot my-snapshot-bundle bundle.tar config.toml
```

Создайте контейнер с использованием этого снимка:

```
ctr container create my-container --snapshot my-snapshot
```

Запустите контейнер:

```
ctr container start my-container
```

Теперь у вас есть контейнер, запущенный с заданным содержимым. Для создания образа вы можете осуществить экспорт контейнера в архив и использовать его в дальнейшем:

```
ctr snapshot mount my-snapshot mountpoint
tar -C mountpoint -c . | docker import - my-image:tag
```

Образ my-image:tag теперь содержит содержимое вашего контейнера и может быть запущен через docker run.

Учтите, что containerd предоставляет более низкоуровневый доступ к контейнерам, поэтому процесс сборки образов может потребовать больше ручной работы и конфигурирования по сравнению с более высокоуровневыми инструментами типа Docker.