

Лабораторная работа №2.

Контейнеризация Docker.

Что потребуется перед началом:

- ПК, способный запустить систему виртуализации с виртуальной машиной GNU/Linux.
- Минимум 6 GiB свободного места на жестком диске (под систему и снапшоты).
- Пакет или установщик системы виртуализации (рекомендуется VirtualBox).
- Загруженный образ дистрибутива (рекомендуется Ubuntu 20.04).
- Результаты работы первой лабораторной.

План и задачи лабораторной:

1. Часть 1. Базовые команды Docker

1. Подготовка рабочего окружения
2. Образа - `docker pull`, `docker images`
3. Метки и удаление образа - `docker tag`
4. Запускаем контейнер - `docker run`, `docker logs`
5. Списки контейнеров - `docker ps`
6. Подключаемся к контейнеру - `docker exec`
7. Список изменений - `docker diff`
8. Завершаем контейнер - `docker stop`, `docker kill`, `docker rm`
9. Не теряем данные - `docker volume`
10. Контейнер Adminer
11. Сети - `docker network`

2. Часть 2. Продвинутая работа с Docker

1. Настройка базы данных
2. Запускаем Adminer
3. Запускаем свой сервис
4. Подробнее про сборку образа
5. Оптимизируем сборку
6. Многоэтапная сборка
7. Делимся образом `docker push`

Отчет - в любом читаемом формате (pdf, md, doc, docx, pages).

Обязательное содержимое отчета:

0. Фамилия и инициалы студента, номер группы, номер варианта
1. План и задачи лабораторной работы
2. Краткое описание хода выполнения работы

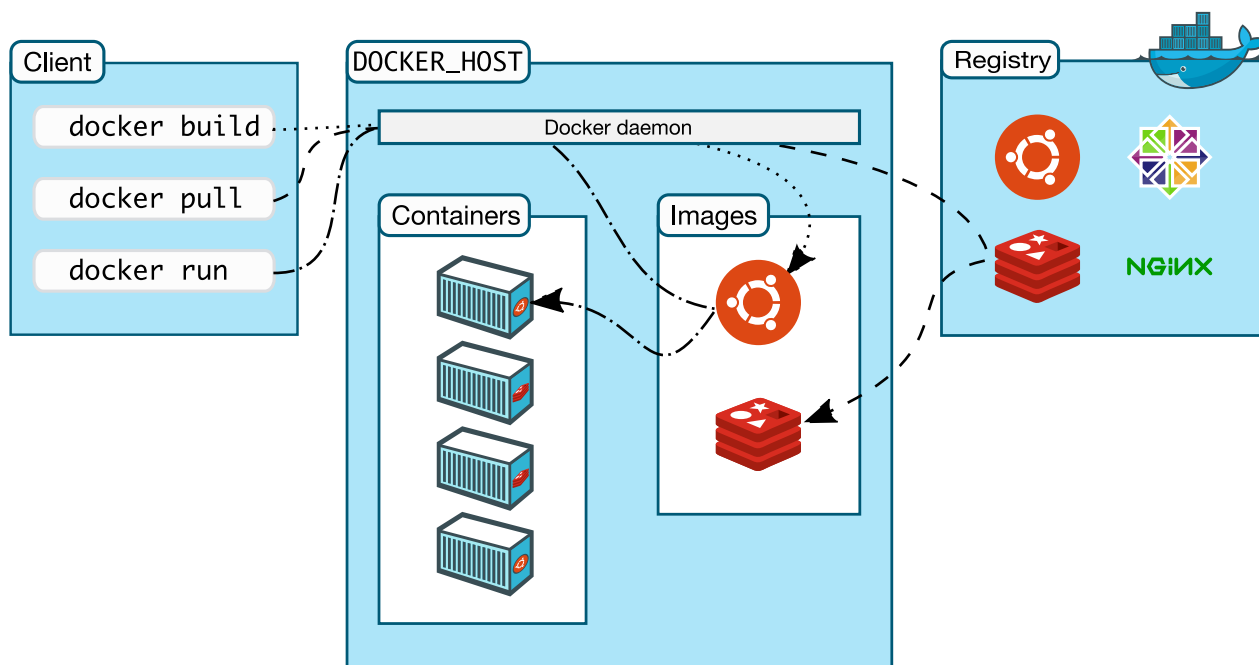
3. Приложить очищенный вывод `history` выполненных команд

Что нужно сделать, чтобы сдать лабораторную?

1. Выполнить все действия, представленные в методических указаниях и ознакомиться с материалом
2. Продемонстрировать результаты выполнения преподавателю, быть готовым повторить выполнение части задач из лабораторной по требованию
3. Ответить на контрольные вопросы

Вступление

Для полного понимания рекомендуется ознакомиться с [вводной статьей по Docker](#).



Docker демон

Docker демон (`containerd` или `dockerd`) обслуживает Docker API запросы (через сокет или по http) и управляет Docker-объектами (образа, контейнеры, сети, тома, и т.д.). Проверить: `systemctl status dockerd`.

Docker клиент

Docker клиент (`docker`) основной способ взаимодействия пользователя с Docker демоном (против работы напрямую с API). Когда вы запускаете команду `docker run`, клиент отправляет сообщение процессу `dockerd` (или `containerd`), который его обрабатывает. Утилита `docker` использует Docker API и может работать сразу с несколькими Docker демонами, не обязательно на локальной машине.

Docker Desktop

Docker Desktop приложение для Mac, Windows или Linux окружений, которое позволяет вам собирать и публиковать ваши контейнеризованные приложения и сервисы. Docker Desktop включает в себя Docker демон, Docker клиент, Docker Compose (с ним будем работать отдельно позже), Docker Content Trust, Kubernetes, и Credential Helper. Больше читайте в руководстве: [Docker Desktop](#).

Docker registry

Docker *registry* (реестр) хранит Docker образа. Docker Hub - публичный реестр для общего пользования, и Docker настроен искать образа там по умолчанию. Также вы можете использовать свой личный реестр.

Часть 1. Базовые команды Docker

1.1. Подготовка рабочего окружения

В предыдущей лабораторной мы уже настроили рабочее окружение в ОС Ubuntu и установили Docker. Если по каким-либо причинам вы этого еще не сделали - проделайте прямо сейчас:

[Установка Docker в Ubuntu 20.04](#)

```
sudo apt install apt-transport-https ca-certificates curl software-properties-  
common  
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -  
sudo add-apt-repository "deb [arch=amd64]  
https://download.docker.com/linux/ubuntu focal stable"  
sudo apt update  
sudo apt install docker-ce  
sudo systemctl status docker
```

Запустим [hello-world](#):

```
docker run hello-world
```

После запуска контейнер отрабатывает, выводит результат работы в STDOUT и сразу завершается, т.к. основной процесс печати руководства внутри него после выполнения завершился.

Если вы хотите, чтобы ваш пользователь мог пользоваться docker без ограничений и необходимости прав суперпользователя - вы можете добавить его в группу docker:

```
usermod -aG docker $username
```

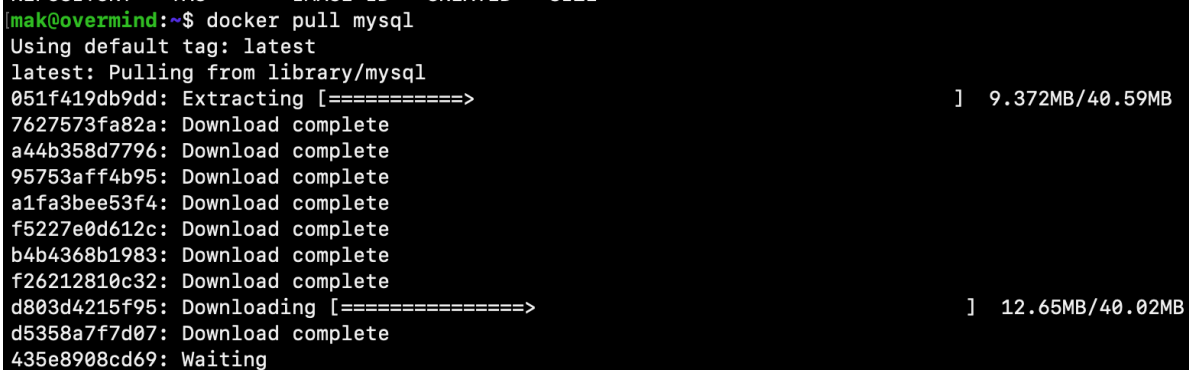
Не забудьте после этого перелогиниться, чтобы процесс командного интерпретатора (в моем случае bash) запустился с новыми правами (с правами на группу docker)

1.2. Образа - `docker pull`, `docker images`

Для того, чтобы запустить "контейнер" нам нужна сущность под названием образ. Подробно все сущности мы разбирали на второй лекции. Остановимся на том, что это заранее упакованный слепок состояния некоторого сервиса или окружения (библиотеки и исполняемые файлы), которое можно использовать без дополнительных зависимостей.

Для загрузки образа используется [команда](#) `docker pull`, загрузим образ `mysql` (найти в ручную подходящие образа можно на [docker hub](#)):

```
docker pull mysql
```

A terminal window with a black background and green text. The prompt is 'mak@overmind:~\$'. The command 'docker pull mysql' is entered. The output shows 'Using default tag: latest', 'latest: Pulling from library/mysql', and a progress bar for '051f419db9dd: Extracting' showing 9.372MB/40.59MB. Other layers are shown as 'Download complete'. The final layer 'd803d4215f95: Downloading' shows 12.65MB/40.02MB. The command ends with '435e8908cd69: Waiting'.

Когда загрузка и распаковка будет завершена, утилита вернет вам управление. Теперь проверим, что же мы скачали с помощью [команды](#) `docker image ls` или `docker images`:

```
docker image ls
docker images
```

Но что, если мы хотим скачать определенную версию образа? Нам помогут метки. Загрузим `mysql` версии 5.7.39:

```
docker pull mysql:5.7.39
```

Убедимся, что теперь у нас есть 2 образа с разными метками:

```
docker images
```

1.3. Метки и удаление образа - `docker tag`

Чтобы управлять нашими образами мы можем назначать им метки - тэги, с помощью [команды](#) `docker tag`:

```
docker tag mysql mysql:8.0-mak
```

```
[mak@overmind:~$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
mysql         latest    43fcfca0776d   7 days ago    449MB
mysql         5.7.39    daff57b7d2d1   4 weeks ago    430MB
[mak@overmind:~$ docker tag mysql mysql:8.0-mak
[mak@overmind:~$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
mysql         8.0-mak    43fcfca0776d   7 days ago    449MB
mysql         latest     43fcfca0776d   7 days ago    449MB
mysql         5.7.39     daff57b7d2d1   4 weeks ago    430MB
mak@overmind:~$
```

Это не создало новый образ (image ID одинаковый), но сделало еще одну ссылку на него.

Удалить образ можно удалив все ссылки на него [командой](#) `docker rmi` или [аналогом](#) `docker image rm`:

```
docker rmi mysql:8.0-mak
docker image rm mysql:5.7.39
```

Если вы не хотите вручную выяснять, какие образа вам более не требуются воспользуйтесь [командой](#) `docker image prune`:

```
docker image prune
# docker image prune -a # удалить все, на которых не запущены контейнеры
```

1.4. Запускаем контейнер - `docker run`, `docker logs`

Запустим контейнер с СУБД [командой](#) `docker run`:

```
docker run mysql
```

```
mak@overmind:~$ docker run mysql
2022-09-21 23:33:40+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL
Server 8.0.30-1.el8 started.
2022-09-21 23:33:40+00:00 [Note] [Entrypoint]: Switching to dedicated user
'mysql'
2022-09-21 23:33:40+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL
Server 8.0.30-1.el8 started.
2022-09-21 23:33:40+00:00 [ERROR] [Entrypoint]: Database is uninitialized and
password option is not specified
    You need to specify one of the following:
    - MYSQL_ROOT_PASSWORD
    - MYSQL_ALLOW_EMPTY_PASSWORD
    - MYSQL_RANDOM_ROOT_PASSWORD
```

Окей, исправляемся, добавим аргумент `-e` для передачи переменной окружения, зададим пароль:

```
docker run -e MYSQL_ROOT_PASSWORD=password mysql
```

База запустилась, но как будто мы просто запустили ее командой `mysql` из консоли.

Задача: поднять еще одно окно и прервать процесс контейнера, который подвесил вашу консоль без ее завершения.

Пригодится: `tmux`, `ps aux`, `kill`

Теперь попробуем запустить в фоновом режиме с аргументом `-d`, кроме того, дадим ему вменяемое имя `--name $NAME`:

```
docker run -d -e MYSQL_ROOT_PASSWORD=password --name db1 mysql
```

Получилось, в ответ Docker выдал нам хэш-ID контейнера (далее `$ID`).

Посмотрим логи контейнера по его ID [командой](#) `docker logs`:

```
docker logs $ID
# Это можно сделать и по имени
docker logs db1
```

1.5. Списки контейнеров - `docker ps`

Посмотрим списки контейнеров с помощью [команды](#) `docker ps`:

```
docker ps
docker ps -a # включая завершённые
```

1.6. Подключаемся к работающему контейнеру - `docker exec`

Подключимся (запустим еще один процесс внутри контейнера) с помощью [команды](#) `docker exec`:

```
# i - interactive - держать STDIN открытым
# t - tty - создать псевдо-tty
docker exec -it db1 /bin/bash
```

```
mak@overmind:~$ docker exec -it db1 /bin/bash
bash-4.4#
bash-4.4# uname -a
Linux 4d5ed4db417d 5.4.0-125-generic #141-Ubuntu SMP Wed Aug 10 13:42:03 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
bash-4.4# mysql
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
bash-4.4# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.30 MySQL Community Server - GPL

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show schemas;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.01 sec)

mysql>
```

1.7. Список изменений - `docker diff`

Посмотрим список изменений в слое на ФС с помощью [команды](#) `docker diff`:

```
docker diff db1
```

```
[mak@overmind:~$ docker exec -it db1 /bin/bash
[bash-4.4#
[bash-4.4#
[bash-4.4# touch file.txt
[bash-4.4# echo "Hello" >> file2.txt
[bash-4.4#
[bash-4.4# exit
[mak@overmind:~$
[mak@overmind:~$
[mak@overmind:~$ docker diff db1
C /root
A /root/.bash_history
A /root/.mysql_history
C /run
C /run/mysqld
A /run/mysqld/mysqlx.sock.lock
A /run/mysqld/mysqld.pid
A /run/mysqld/mysqld.sock
A /run/mysqld/mysqld.sock.lock
A /run/mysqld/mysqlx.sock
A /file.txt
A /file2.txt
mak@overmind:~$
```

Видимо наличие новых файлов `file.txt` и `file2.txt`.

1.8. Завершаем контейнер - `docker stop`, `docker kill`, `docker rm`

Завершим контейнер нежно с помощью [stop](#), сразу завершим с помощью [kill](#) и удалим остатки (из списка завершенных, включая логи контейнера) с помощью [rm](#):

```
docker stop db1
docker kill db1
docker rm db1
```

1.9. Не теряем данные - `docker volume`

Запустим контейнер обратно:


```
docker run -d -e MYSQL_ROOT_PASSWORD=password --name db1 mysql
# И тут я спалил пароль в истории в открытом виде
```

Как избежать утечки пароля через историю bash?

```
# Грузите из переменных окружения из файла:
nano .docker_mysql_rc
# Вписываем необходимые переменные

# Передаем как переменную (если одна)
source .docker_mysql_rc
docker run -d -e MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD} --name db1
mysql

# Или целиком файл (если много):
docker run -d --env-file ./docker_mysql_rc --name db1 mysql
```

Выведем содержимое файла:

```
docker exec -it db1 cat file2.txt
```

О ужас! Его нет. Как и всего содержимого базы. **Все данные удалились при завершении контейнера.**

Чтобы этого избежать запустим контейнер с томом:

```
docker run --rm -d \
-v mysql:/var/lib/mysql \
-v mysql_config:/etc/mysql \
--name db1 \
-e MYSQL_ROOT_PASSWORD=password \
mysql
```

Внесем изменения:

```
docker exec -it db1 mysql -ppassword
# И тут я снова спалил пароль в истории в открытом виде
# Лучше написать -p, а дальше вводить интерактивно
# Но так как задача образовательная – пока игнорируем
```

```
create database testdb;
create database blog;
show schemas;
```

Завершим контейнер:

```
docker stop db1
```

А теперь перезапустим и убедимся, что базы testdb и blog не исчезли (как и все изменения на ФС):

```
docker run --rm -d \  
  -v mysql:/var/lib/mysql \  
  -v mysql_config:/etc/mysql \  
  --name db1 \  
  -e MYSQL_ROOT_PASSWORD=password \  
  mysql  
  
docker exec -it db1 mysql -ppassword -e "show schemas;"
```

[Просмотреть список томов:](#)

```
docker volume ls
```

[Создать новый:](#)

```
docker volume create test
```

[Посмотреть информацию о томе:](#)

```
docker volume inspect test
```

[Удалить том:](#)

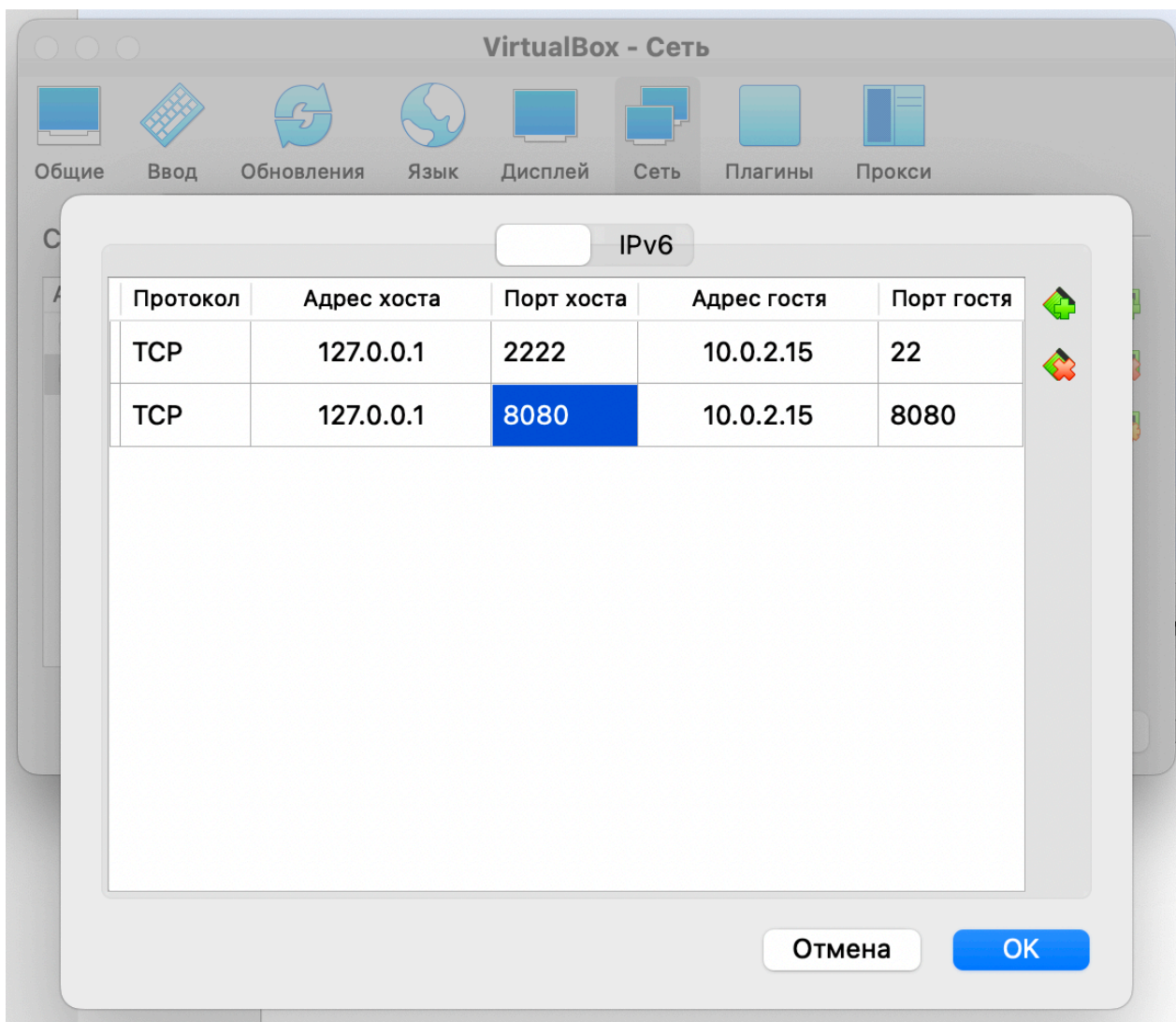
```
docker volume rm test
```

[Очистить лишние тома:](#)

```
docker volume prune
```

1.10. Контейнер Adminer

Настроим дополнительный проброс портов (если кто-то настроил bridge - то просто идем на IP виртуальной машины):



Запустим образ [Adminer](#) (скачается автоматически). Для того, чтобы попасть из виртуальки внутрь контейнера на порт 8080 укажем ключик `-p HostPort:ContainerPort`:

```
docker run -d -p 8080:8080 --name adminer adminer
```

Подключимся в браузере на хосте к `http://127.0.0.1:8080/`:

Language: English

Adminer 4.8.1

Login

System	MySQL
Server	db
Username	
Password	
Database	

Login ☐ Permanent login

Adminer (бывший phpMinAdmin) — это легковесный инструмент администрирования [MySQL](#), [PostgreSQL](#), [SQLite](#), [MS SQL](#) и [Oracle](#). Проект родился как «облегчённый» вариант [phpMyAdmin](#). Распространяется в форме одиночного [PHP](#)-файла размером около 380 KB, который является результатом компиляции исходных php- и js-файлов с помощью специального PHP-скрипта. Т.о. контейнер с ним содержит php-сервер и один php-скрипт.

Однако как бы мы не пытались подключиться к базе - ничего не выйдет. Контейнеры не связаны по сети.

1.10. Сети - `docker network`

Для начала попробуем связать контейнеры простым способом, завершим предыдущий контейнер Adminer и запустим новый, с параметром `--link Container:AliasName`.

```
docker rm -f adminer
docker run -d -p 8080:8080 --link db1:mysql --name adminer adminer
```

Теперь подключимся к базе по ее `Alias` из контейнера adminer:

Login - mysql - Adminer

127.0.0.1:8080/?server=mysql

Language: English

Adminer 4.8.1

Login

Logout successful. Thanks for using Adminer, consider [donating](#).

System	MySQL
Server	mysql
Username	root
Password
Database	testdb

☐ Permanent login

Database: testdb - mysql - Adminer

127.0.0.1:8080/?server=mysql&username=root&db=testdb

Language: English

MySQL » mysql » Database: testdb

Adminer 4.8.1

DB: testdb

[SQL command](#) [Import](#) [Export](#) [Create table](#)

No tables.

Database: testdb

[Alter database](#) [Database schema](#) [Privileges](#)

Tables and views

No tables.

[Create table](#) [Create view](#)

Routines

[Create procedure](#) [Create function](#)

Events

[Create event](#)

Неплохо, но такой способ считается устаревшим. К тому же, это может быть не всегда удобно. Теперь создадим новую сеть:

```
docker network create cluster
```

Кстати в man-странице по этой команде есть много интересного про параметры:

```
man docker-network-create
```

Specifying advanced options

When you create a network, Engine creates a non-overlapping subnet for the network by default. This subnet is not a subdivision of an existing network. It is purely for ip-addressing purposes. You can override this default and specify subnet values directly using the `--subnet` option. On a **bridge** network you can only create a single subnet:

```
$ docker network create -d bridge --subnet=192.168.0.0/16 br0
```

Additionally, you also specify the `--gateway` `--ip-range` and `--aux-address` options.

```
$ docker network create \
  --driver=bridge \
  --subnet=172.28.0.0/16 \
  --ip-range=172.28.5.0/24 \
  --gateway=172.28.5.254 \
  br0
```

If you omit the `--gateway` flag the Engine selects one for you from inside a preferred pool. For **overlay** networks and for network driver plugins that support it you can create multiple subnetworks.

```
$ docker network create -d overlay \
  --subnet=192.168.0.0/16 \
  --subnet=192.170.0.0/16 \
  --gateway=192.168.0.100 \
  --gateway=192.170.0.100 \
  --ip-range=192.168.1.0/24 \
  --aux-address="my-router=192.168.1.5" --aux-address="my-switch=192.168.1.6" \
  --aux-address="my-printer=192.170.1.5" --aux-address="my-nas=192.170.1.6" \
  my-multihost-network
```

Be sure that your subnetworks do not overlap. If they do, the network create fails and Engine returns an error.

Network internal mode

By default, when you connect a container to an **overlay** network, Docker also connects a bridge network to it to provide external connectivity. If you want to create an externally isolated **overlay** network, you can specify the `--internal` option.

Network ingress mode

You can create the network which will be used to provide the routing-mesh in the swarm cluster. You do so by specifying `--ingress` when creating the network. Only one ingress network can be created at the time. The network can be removed only if no services depend on it. Any option available when creating an overlay network

Manual page docker-network-create(1) line 50 (press h for help or q to quit)

Проверим как создалась сеть с параметрами по умолчанию:

```
[mak@overmind:~$ docker network create cluster
ed17695fdaced7541b802a9e70a5e4ea6f52e6466385dad2a366925a32fea965]
[mak@overmind:~$ docker network ls]
NETWORK ID        NAME        DRIVER    SCOPE
56c7c79652dd     bridge     bridge    local
ed17695fdace     cluster    bridge    local
e9be1a44af63     host       host      local
132c2c8ded6c     none      null      local
[mak@overmind:~$ docker network inspect cluster]
[
  {
    "Name": "cluster",
    "Id": "ed17695fdaced7541b802a9e70a5e4ea6f52e6466385dad2a366925a32fea965",
    "Created": "2022-09-22T10:02:58.901756599Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
    "Options": {},
    "Labels": {}
  }
]
```

Проверим в какой сети работают сейчас Adminer и MySQL:

```
docker inspect db1
docker inspect adminer | egrep "IPAddress|Gateway|IPPrefixLen"
```

```
[mak@overmind:~]$ docker inspect adminer | egrep "IPAddress|Gateway|IPPrefixLen"
    "SecondaryIPAddresses": null,
    "Gateway": "172.17.0.1",
    "IPAddress": "172.17.0.3",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "Gateway": "172.17.0.1",
    "IPAddress": "172.17.0.3",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
```

Теперь пересоздадим контейнеры СУБД и Adminer в этой сети:

```
docker rm -f db1
docker rm -f adminer

# Добавим к предыдущей команде запуска: --net NetName
docker run --rm -d \
  -v mysql:/var/lib/mysql \
  -v mysql_config:/etc/mysql \
  --name db1 \
  -e MYSQL_ROOT_PASSWORD=password \
  --net cluster \
  mysql

# А теперь аналогично для Adminer
docker run -d -p 8080:8080 --net cluster --name adminer adminer
```

Проверим их IP-адреса:

```
docker inspect db1 | egrep "IPAddress|Gateway|IPPrefixLen"
docker inspect adminer | egrep "IPAddress|Gateway|IPPrefixLen"
```

Попробуем подключиться к базе по IP-адресу:

Login - 172.18.0.2 - Adminer x +

← → 127.0.0.1:8080/?server=172.18.0.2 ☆ ↻

Language: English ▾

Adminer 4.8.1

Login

Logout successful. Thanks for using Adminer, consider [donating](#).

System	MySQL ▾
Server	172.18.0.2
Username	root
Password
Database	testdb

Login ☐ Permanent login

Кроме того, docker предоставляет dns-записи внутри своих сетей, поэтому мы можем обратиться по имени:

Login - 172.18.0.2 - Adminer x +

← → 127.0.0.1:8080/?server=172.18.0.2 ☆ ↻

Language: English ▾

Adminer 4.8.1

Login

Logout successful. Thanks for using Adminer, consider [donating](#).

System	MySQL ▾
Server	db1
Username	root
Password
Database	testdb

Login ☐ Permanent login

Для диагностики сетей есть [полезный образ](#), подробное применение рассматривается по ссылке. Попробуем запустить его в интерактивном режиме и проверить сеть контейнеров с помощью `nmap`:

```
docker run -it --net cluster nicolaka/netshoot
```



```

1cb6452f30ec: Pull complete
cc37e878581f: Pull complete
4f4fb700ef54: Pull complete
b01a5ec78db0: Pull complete
123e924da2d1: Pull complete
367cf0ba0067: Pull complete
5a3733011111: Pull complete
6adf3d98c1c3: Pull complete
77c79836780d: Pull complete
4aafb6d72b2b: Pull complete
Digest: sha256:aeafd567d7f7f1edb5127ec311599bb2b8a9c0fb31d7a53e9cff26af6d29fd4e
Status: Downloaded newer image for nicolaka/netshoot:latest
      dP      dP      dP
      88      88      88
88d888b. .d8888b. d8888P .d8888b. 88d888b. .d8888b. .d8888b. d8888P
88' `88 88oooo8 88 Y8ooooo. 88' `88 88' `88 88' `88 88
88 88 88. ... 88 88 88 88 88. .88 88. .88 88
dP dP `88888P' dP `88888P' dP dP `88888P' `88888P' dP

```

Welcome to Netshoot! (github.com/nicolaka/netshoot)

```
[ 91391eb86c98 [?]~[?]
```

```
[ 91391eb86c98 [?]~[?] nmap db1
```

```

Starting Nmap 7.92 ( https://nmap.org ) at 2022-09-22 10:22 UTC
Nmap scan report for db1 (172.18.0.2)
Host is up (0.0000090s latency).
rDNS record for 172.18.0.2: db1.cluster
Not shown: 999 closed tcp ports (reset)
PORT      STATE SERVICE
3306/tcp  open  mysql
MAC Address: 02:42:AC:12:00:02 (Unknown)

```

Nmap done: 1 IP address (1 host up) scanned in 0.11 seconds

```
[ 91391eb86c98 [?]~[?] nmap adminer
```

```

Starting Nmap 7.92 ( https://nmap.org ) at 2022-09-22 10:23 UTC
Nmap scan report for adminer (172.18.0.3)
Host is up (0.0000080s latency).
rDNS record for 172.18.0.3: adminer.cluster
Not shown: 999 closed tcp ports (reset)
PORT      STATE SERVICE
8080/tcp  open  http-proxy
MAC Address: 02:42:AC:12:00:03 (Unknown)

```

Nmap done: 1 IP address (1 host up) scanned in 0.09 seconds

```
91391eb86c98 [?]~[?]
```

Теперь выйдем и грохнем все контейнеры:

```
docker stop $(docker ps -a -q)
```

Часть 2. Продвинутая работа с Docker

2.1. Настройка базы данных

```
docker run --rm -d \
  -v mysql:/var/lib/mysql \
  -v mysql_config:/etc/mysql \
  --name mysql \
  -e MYSQL_ROOT_PASSWORD=password \
  -e MYSQL_DATABASE=blog \
  --net cluster \
  mysql
```

Подключимся к рабочей базе и создадим таблицу

```
docker exec -it mysql mysql -p
# Вводим пароль из переменной MYSQL_ROOT_PASSWORD, который мы сами задали
```

Создадим табличку **posts** в базе **blog**

```
CREATE TABLE blog.posts (
  id INT NOT NULL AUTO_INCREMENT,
  title varchar(255),
  created TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (id)
);
```

Выйдем из контейнера с помощью ctrl + D

2.2. Запускаем Adminer

Запустим Adminer в той же сети

```
docker run -d -p 8080:8080 --net cluster --name adminer adminer
```

Подключимся к <http://127.0.0.1:8080/> (или IP вашей ВМ, если у вас бридж-сеть).

В открывшемся интерфейсе подключимся к базе, указав:

Поле	Значение	Примечание
Сервер	mysql	имя контейнера с СУБД
Имя пользователя	root	
Пароль	password	
База данных	blog	

Если все прошло успешно, вы увидите нашу таблицу `posts`.

Перейдем в нее и добавим несколько записей с помощью "Новая запись".

Можно указывать только поле `title`, т.к. остальные поля генерятся автоматически при создании объекта.

При желании можно создать записи и с помощью SQL через контейнер с `mysql` или через SQL-запрос в Adminer:

```
INSERT INTO blog.posts (title)
VALUES ("First Post"), ("Second Post"), ("Third Post");
```

2.3. Запускаем свой сервис

Загрузим себе материалы лабораторной:

```
git clone https://gitlab.com/tlakatlekutl/devops-lab2 && cd devops-lab2
```

В каталоге `code/` находится пример примитивного API сервиса на Golang. Для того чтобы начать с ним работать, необходимо собрать образ. Для этого воспользуемся следующей командой:

```
docker build -t step1 -f step1.Dockerfile code/
```

где:

- опция `-t` задает имя образа, иначе имя будет равно ID образа
- опция `-f` указывает путь к Dockerfile, если ее не указывать докет будет искать файл Dockerfile в текущем каталоге
- `code/` - это контекст для сборки образа, все файлы, которые должны быть доступны во время сборки.

Запустим наш сервис (собранный ранее образ `step1`), передадим ему параметры подключения к базе через переменные окружения (с помощью опции `-e`), а также подключим его к контейнеру `mysql`.

```
docker run --net cluster -p 8000:8000 -e MYSQL_HOST=mysql -e MYSQL_USER=root -e MYSQL_PASSWORD=password step1
```

Проверим работу нашего приложения выполнив команду `curl`, в ответе должны увидеть json массив с созданными постами.

```
curl localhost:8000/posts
```

2.4. Подробнее про сборку образа

Рассмотрим структуру Dockerfile на примере `step1.Dockerfile`.

В начале любого образа должен быть указан базовый образ, на основе которого собирается новый. В нашем случае `FROM golang:1.19.1`, самый базовый возможный образ `FROM scratch`, в нем нет ничего.

`COPY` копирует файл из контекста (папки, которую вы указываете при `docker build`) в указанное место в образе. Команда создает новый слой в образе;

`RUN` команда - запускает команду в образе, создает новый слой;

Остальные операнды либо создают пустой слой, либо напрямую меняют метаданные в образе.

`EXPOSE` показывает докеру, какие порты слушает приложение;

`CMD` задает команду запуска в контейнере;

`WORKDIR` меняет рабочую директорию;

`USER` меняет пользователя, под которым идет работа;

`ENV` задает переменные окружения;

и т.д. Подробнее можно ознакомиться в [официальной документации](#).

Взглянем на получившиеся слои у образа:

```
docker history step1
```

Как было показано на лекции, каждый получившийся слой - это архив с измененными файлами.

2.5. Оптимизируем сборку

Для удобства вынесем переменные окружения в отдельный файл, например `config.env`:

```
MYSQL_DB=blog
MYSQL_HOST=mysql
MYSQL_USER=root
MYSQL_PASS=password
```

Соберем образ step2:

```
docker build -t step2 -f step2.Dockerfile code/
```

Запустим образ step2 и проверим, что он работает, аналогично предыдущему:

```
docker run --net cluster -p 8000:8000 --env-file=config.env step2
```

Задание:

Сравните образы step1 и step2 (их Dockerfile и history) и опишите различия.

2.6. Многоэтапная сборка

Вы могли заметить, что для работы нашего сервиса необходим только исполняемый файл, тогда возникает вопрос "А зачем нам тащить с собой сборочные зависимости?" Да не нужны они, поэтому существует многоэтапная сборка: вы собираете исполняемый файл в отдельном образе, а затем копируете результат в эксплуатируемый образ.

Соберем, запустим и протестируем образ step3

```
docker build -t step3 -f step3.Dockerfile code/  
docker run --net cluster -p 8000:8000 --env-file=config.env step3
```

Задание:

Взгляните на step3.Dockerfile, в чем принципиальное отличие от предыдущих докерфайлов? Оцените docker history.

2.7. Делимся образом - `docker push`

Для ознакомления:

```
# Авторизируемся где-нибудь на приватном registry  
docker login  
  
# Смотрим что за образ мы хотим загрузить  
docker images  
  
# Тэгаем  
docker tag XXX:VERSION registry-gitlab.com/user_or_group/XXX:VERSION  
  
# Загружаем  
docker push registry-gitlab.com/user_or_group/XXX:VERSION
```

Контрольные вопросы

1. Что такое и зачем нужен Docker? Альтернативные системы?
2. Как получить Docker-образ, что это такое?
3. Как запустить контейнер? Как получить доступ к его портам?
4. Как просмотреть логи контейнера?
5. Как сохранить данные внутри контейнера между его перезапусками?
6. Как подключить контейнеры к одной сети? Какие есть альтернативные варианты?

7. Почему контейнеры могут обращаться между собой по имени (хэшу, если его нет)?
8. Что такое метки (docker tag)?
9. Как удалить ненужные образа и контейнеры?
10. Как запустить что-то внутри работающего контейнера?
11. Как узнать, какие файлы изменяет программа внутри контейнера?
12. Когда происходит завершение контейнера? Как сделать?
13. Перезапустите сборку собранного образа, оцените время пересборки, объясните причины.
14. К какому числу слоев стремиться в образе, правила оптимизации?
15. Опишите базовые команды Dockerfile, что они делают, где смотреть документацию?
16. Что такое контекст сборки, как его оптимизировать?