

03 Установка Docker на нашем стенде

Переключитесь в режим суперпользователя `root`, для упрощения обучения в данном курсе команды будут запускаться от учетной записи `root`, если не оговорено иное.

Переходим собственно к установке.

1. Устанавливаем `yum-utils`, для того чтобы импортировать далее репозиторий:

```
yum install -y yum-utils
```

2. Добавляем репозиторий с официального URL:

```
yum-config-manager \
  --add-repo \
  https://download.docker.com/linux/centos/docker-ce.repo
```

Если вы не доверяете файлу репозитория, можете скачать его и посмотреть содержимое в терминале, а потом вставить в файл `/etc/yum.repos.d/docker-ce.repo`

3. Устанавливаем пакеты нужной нам версии:

```
yum install docker-ce-19.03.12-3.el7 docker-ce-cli-19.03.12-3.el7 containerd.io
```

Мы специально указываем определённые версии пакетов, чтобы в недалёком будущем новый релиз случайно не сломал практики в этом курсе.

После установки в RHEL/CentOS служба не запускается автоматически, поэтому мы добавим её в автозапуск и стартуем одной командой:

4. Запускаем docker engine:

```
systemctl enable docker --now
```

Вот и всё! Docker установлен.

Можете проверить версию:

```
docker version
```

И запустить для теста контейнер `hello-world`:

```
docker run hello-world
```

04.1 Немного практики. Запускаем свой первый контейнер

1. Выполните следующую команду в консоли:

```
docker run -t -i debian:jessie bash
```

Обратите внимание, что все параметры для запуска контейнера (-t -i) указываются до имени образа.

Всё, что находится после образа является командами и параметрами внутри контейнера.

После выполнения данной команды, Вы попадете в консоль контейнера:

```
docker run -t -i debian:jessie bash
Unable to find image 'debian:jessie' locally
jessie: Pulling from library/debian
bf295113f40d: Pull complete
Digest: sha256:201b887113d1190ed7811152af4c14f46a7ce8612dfdaee540fc353f5135d25a
Status: Downloaded newer image for debian:jessie
root@fe8a995c3727:/#
```

2. Выполните внутри контейнера команду:

```
cat /etc/os-release
```

Скопируйте и отправьте ее результат в качестве ответа.

P.S. Для выхода из контейнера наберите команду *exit* или нажмите Ctrl-D.

04.2 Запускаем второй контейнер

1. Выполните следующую команду в консоли:

```
docker container run -d -p 8080:80 nginx:1.13
```

После выполнения данной команды убедитесь, что контейнер запущен:

```
docker container ls
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS         NAMES
3c281f47fb9a   nginx:1.13     "nginx -g 'daemon of..." 5 seconds ago  Up 4 seconds
0.0.0.0:8080->80/tcp   mystifying_chaplygin
```


2. С локальной машины попробуйте обратиться на localhost:8080:

```
curl localhost:8080
```

3. Посмотрите в логи контейнера:

Обратите внимание, что вместо плэйсхолдера <ИМЯ КОНТЕЙНЕРА> Вам нужно подставить реальное имя вашего контейнера из команды *docker ps*

```
docker container logs <ИМЯ КОНТЕЙНЕРА>
```

Скопируйте и отправьте одну строку из лога контейнера как решение для данного шага.

4. После выполнения задания остановите и удалите контейнер одной командой:

```
docker container rm -f <ИМЯ КОНТЕЙНЕРА>
```

04.3 Запускаем третий контейнер

1. Сохраните на вашем стенде с Docker файл default.conf с содержимым

```
server {  
    listen      80 default_server;  
    server_name _;  
  
    location / {  
        return 200 '$hostname\n';  
    }  
}
```

2. Запустите контейнер с сохранённым файлом:

```
docker container run -d -p 8080:80 -v "$PWD/default.conf":/etc/nginx/conf.d/default.conf  
nginx:1.13
```

После выполнения данной команды убедитесь, что контейнер запущен:

```
docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS NAMES				
e0911beb600a	nginx:1.13	"nginx -g 'daemon of...'"	3 seconds ago	Up 1 second
0.0.0.0:8080->80/tcp recursing_driscoll				

3. С локальной машины попробуйте обратиться на localhost:8080:

```
curl localhost:8080
```

Скопируйте и отправьте ответ этой команды в качестве решения данного шага.

4. После выполнения задания остановите и удалите контейнер:

```
docker rm -f <ИМЯ КОНТЕЙНЕРА>
```

04.4 Самостоятельная работа

Запустите контейнер из образа *slurmio/hello-nginx:latest*, спроксируйте локальный порт *8080* в *80* порт контейнера, при запуске задайте переменной окружения *USERNAME* в контейнере значение с вашим именем.

Чтобы при старте контейнера задавать значения переменных окружения, добавьте к команде *docker run* ключ *-e*.

Пример:

```
docker run -e 'VARIABLE=value' nginx:1.12
```

Обратитесь на спроксированный порт, скопируйте ответ контейнера и отправьте его как решение для этого шага.

Не забудьте удалить контейнер после окончания работы, чтобы не держать занятым локальный порт 8080, он нам еще понадобится.

05.1 Запуск собственного приложения в Docker

Давайте повторим то, что было сделано в видео-уроке. Вам потребуется клонировать репозиторий <https://gitlab.slurm.io/edu/docker-2020>. В нем нам нужна директория *5.own_app*

1. Создайт и активируйте python окружение.
2. Создайте скрипт с содержимым *5.own_app/hello.py*, с именем *hello.py*
3. Установите flask

```
pip install flask
```
4. Сформируйте файл *requirements.txt* в директории со скриптом.
5. Создайте файл *Dockerfile* в директории со скриптом, с содержимым *5.own_app/Dockerfile*

6. Соберите образ
`docker build -t myapp:1.0 .`
7. Запустите приложение
`docker run -d -p 5000:5000 myapp:1.0`
8. Проверьте, что приложение работает
`curl 127.0.0.1:5000`

05.2

1. Создайте bash-скрипт с именем `hello.sh`, который будет выводить строку "Hello world!".
2. Создайте `Dockerfile` для запуска скрипта `hello.sh`, в качестве основного образа используйте `ubuntu`
3. Создайте образ из `Dockerfile`, полученного на шаге 2, и запустите из него контейнер
4. Введите вывод команды `docker ps -a --no-trunc`, столбец `COMMAND`

06.1 Запускаем простой `docker-compose` проект

1. Склонируйте репозиторий с практикой на ваш стенд и перейдите в директорию `6.docker-compose`:

```
git clone git@gitlab.slurm.io:edu/docker-2020.git
cd docker-2020/6.docker-compose
```

2. Выполните команду для запуска проекта:

```
docker-compose up -d
```

3. С локальной машины попробуйте обратиться на `localhost:8000`:

```
curl localhost:8000/polls/
```

4. Скопируйте ответ приложения и отправьте в качестве решения в данном шаге.
5. После выполнения задания остановите проект с помощью команды:

```
docker-compose down
```

06.2 Самостоятельная работа

1. Склонируйте репозиторий с практикой на ваш стенд и перейдите в директорию 6.docker-compose_practice:

```
git clone git@gitlab.slurm.io:edu/docker-2020.git
cd docker-2020/6.docker-compose_practice
```

2. Самостоятельно напишите Dockerfile и docker-compose.yml файлы для сборки и запуска приложения в директории.

- В Dockerfile должны устанавливаться зависимости из файла requirements.txt (*pip install -r requirements.txt*)
- Само приложение должно запускаться командой *python setup.py && flask run --host=0.0.0.0 --port 8000*
- Приложение должно быть доступно на 8000 порту
- Приложение должно работать от имени пользователя *app*
- Рабочей директорией в контейнере должен быть путь */app*
- В Dockerfile также необходимо передавать дополнительную переменную для работы приложения - *FLASK_APP=src/app.py*
- В docker-compose.yml файле должно быть два сервиса - db (postgres) и app.
- Приложение должно собираться автоматически при выполнении команды *docker-compose up*
- Приложение должно запускаться из локальных файлов проекта (то есть нужно монтировать директорию с кодом в рабочую директорию контейнера с приложением)
- Для работы приложения ему нужно передать следующие переменные:
 - DB_NAME
 - DB_USER
 - DB_PASSWORD
 - DB_HOST
 - FLASK_ENV: development

3. Приложение должно уметь сохранять пользователей с *lastName* и *firstName* и выводить данные пользователя по запросу с id.

Примеры запросов:

```
curl localhost:8000/api/v1/users -X POST -d '{"firstName":"John","lastName":"Cena"}' --header "Content-Type: application/json"
```

```
curl localhost:8000/api/v1/users/1
```

4. Создайте тестового пользователя, скопируйте ответ приложения и отправьте в качестве решения данного шага.

07.1 Подготовка к деплою

1. Создание variables в Gitlab

Для доступа из Gitlab на стенд нам необходимо добавить в Gitlab переменную, в которой будет содержаться IP-адрес вашего хоста с Docker, а также приватный ключ с вашего стенда.

- Переходим в gitlab

Для этого открываем в браузере свой форк xpaste. <Ваш номер логина> меняем на свой номер студента:

`https://gitlab.slurm.io/s<Ваш номер логина>/xpaste`

- Добавляем переменные

Для этого в левом меню находим Settings -> CI/CD -> Variables и нажимаем Expand. В поле Name вводим имя первой переменной:

`PRODUCTION_SERVER_IP`

В поле value вводим адрес, у каждого студента он свой. Если вы знаете, как посмотреть IP-адрес хоста - впишите его сюда, а если нет - посмотрите в личном кабинете вашу подсеть (например, 172.15.5.0) и вместо нуля подставьте число 20.

Вторая переменная, которая нам нужна - это

`SSH_PRIVATE_KEY`

Чтобы её узнать, на стенде введите команду

```
cat ~/.ssh/id_rsa
```

И скопируйте вывод полностью в значение переменной в Gitlab.

Далее не забудьте нажать Save variables для сохранения переменных.

2. Подготовка стенда с Docker

Для работы приложению нужна как минимум база данных PostgreSQL. В целях обучения мы запустим её прямо в контейнере на том же стенде.

Вам понадобятся такие команды:

```
docker network create --driver bridge xpaste-net
```



```
docker run --name xpaste-postgres --network xpaste-net -e
POSTGRES_PASSWORD=postgres -e POSTGRES_DB=xpaste -d postgres:9.6
```

На этом всё, можно переходить к следующему шагу.

07.2 CI/CD и деплой

В этой части мы добавляем описание шагов CI/CD - это то, что нужно для автоматизации деплоя приложения из Gitlab на хост с Docker. Деплой производится посредством выполнения на стенде команд с gitlab-runner по SSH.

1. Подготавливаем ci/cd

Для этого скопируем .gitlab-ci.yml в проект xpaste, выполнив команду:

```
cp ~/docker-2020/7.ci-cd/1.2.deploy/.gitlab-ci.yml ~/xpaste/
```

В нём ничего править не нужно, все значения будут использоваться из переменных в Gitlab.

Сохраняем все изменения и пушим их в репозиторий. Для этого необходимо выполнить команды:

```
cd ~/xpaste
git add .
git commit -am "add deploy stage"
git push
```

2. Проверка результата

Для проверки результата необходимо перейти в gitlab в раздел ci/cd -> pipelines форка проекта xpaste. Можно воспользоваться прямой ссылкой:

<https://gitlab.slurm.io/sXXXXXX/xpaste/pipelines>. sXXXXXX необходимо заменить на номер своего студента.

В результате все 5 job'ов должны закончиться без ошибок.

3.Открываем приложение в браузере

В браузере открываем url: <http://xpaste.s<Ваш номер логина>.edu.slurm.io>. <Ваш номер логина> необходимо заменить на номер своего студента. Открывать лучше в режиме инкогнито, так как современные браузеры без спроса подставляют <https://> и потом это запоминают как единственно верный вариант.

Всё готово! Вы только что автоматизировали сборку, тестирование и деплой приложения при помощи Docker и Gitlab.

8.1

1. Посмотрите какие Ns есть в текущей системе
`lsns`
2. Создадим форк от проца в ns pid:
`unshare -p -f --mount-proc=/proc`
3. Проверьте NS теперь, вы находитесь в новом контейнере
4. Во втором окне, проверьте нсы, что mnt и pid были задействованы.
5. Во втором окне выполните команду
`nsenter -t pid-container --pid --net --mount --ipc --uts sh`
6. Проверьте, что pid 1 в контейнере отличается от хостовой машины
`ps aux`

+13

9.1

1. Запустите 2 контейнера alpine:
`docker run -dit --name alpine1 alpine ash`

`docker run -dit --name alpine2 alpine ash`
2. Проверьте, что контейнеры запустились:
`docker container ls`
3. Проверьте, что к bridge присоединились эти контейнеры:
`docker network inspect bridge`
4. Зайдите в контейнер
`docker attach alpine1`
5. Проверьте, что второй контейнер доступен из первого.
6. Создайте свою сеть bridge
`docker network create --driver bridge alpine-net`
7. Запустите alpine3 в новой сети
`docker run -dit --name alpine3 --network alpine-net alpine ash`
8. Запустите alpine4 и присоедините новую сеть.
`docker run -dit --name alpine4 alpine ash`

`docker network connect alpine-net alpine4`
9. Изучите новую сеть, попингуйте из alpine3 остальные контейнеры.

10.1

С помощью знаний, полученных на этом уроке, на стенде проделайте следующее:

1. Переключите драйвер логирования на "syslog"
2. Включите режим отладки
3. Запустите контейнер из образа **hello-world** и узнайте его ID
4. Посмотрите его логи с помощью команды
`grep <ID вашего контейнера> /var/log/messages | grep debug`
5. Введите в поле ответа вывод команды выше

11.1

Репозиторий с скриптом очистки <https://rpms.southbridge.ru/>, пакет `docker_image_prune`.
Скрипт так же можно найти в репо курса `docker-2020/11.registry/docker_image_prune.sh`

Запустите ваш регистри локально, командой

```
docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

Напишите количество слоев у образа `ubuntu:18.04`

12.1

На стенде выполните следующие действия:

1. Запустите контейнер из образа **ubuntu:16.04** с пустым томом **testvol**, смонтированным в **/var/log**.
2. Остановите этот контейнер.
3. С помощью команды
`docker system df -v`
узнайте, сколько информации туда попало.
4. В качестве ответа введите объем данных из колонки **SIZE** для тома **testvol**.

Можете прибратся за собой - удалите том `testvol` командой

```
docker volume prune
```

не забыв перед этим удалить контейнер, к которому был привязан этот том.

13.1

Оптимизируйте Dockerfile из репо `docker-2020/13.best_practices`. Какой образ вы возьмете за основу?

14.1 Самостоятельная работа

1. Склонируйте репозиторий с практикой на стенд и перейдите в директорию `14.docker_practice`:

```
git clone git@gitlab.slurm.io:edu/docker-2020.git
cd docker-2020/14.docker_practice
```

2. Самостоятельно напишите Dockerfile для сборки приложения в директории, используя все известные вам способы оптимизации для Python.

- В Dockerfile должны устанавливаться зависимости из файла `requirements.txt` (*`pip install -r requirements.txt`*)
- Само приложение должно запускаться командой *`python app.py`*
- Приложение должно быть доступно на *8000* порту
- Приложение должно работать от имени пользователя *app*
- Рабочей директорией в контейнере должен быть путь */app*

Постарайтесь добиться, чтобы размер вашего образа с готовым к работе приложением не превышал 150 мб.

3. Для проверки работы приложения в контейнере выполните команду:

```
curl '127.0.0.1:8000/sum?a=83&b=12'
```

Приложение выведет сумму параметров *a* и *b*.