Команды в linux

Ldd /bin/bash – посмотреть ссылки на необходимые библиотеки

chroot gb – команда для смены корневого каталога системы

ps – покажет PID запущенный процессов

top – покажет PID всех запущенных процессов

gcc exemple1.c -o example1.sh – компилируем файл, т.е. файл exemple1.c компилируем в exemple1.sh

./example1.sh – запускаем файл на исполнение

uname – покажет название операционной системы

uname -n – покажет название образа

uname –r – покажет имя ядра системы

uname –m – покажет архитектуру системы

uname –a – покажет всю информацию о системе

$$ - обозначается PID процесса, который выполняется нашей оболочкой

lxc-create -n test123 -t ubuntu -f /usr/share/doc/lxc/exemples/lxc-veth.conf – установка контейнера lxc

lxc-start -d -n test123 – запуск контейнера lxc

lxc-attach -n test123 – подключение к контейнеру lxc

lxc-ls –f – отобразить состояние запущенных контейнеров

lxc-stop -n test123 – остановка контейнера

lxc-destroy -n test123 – удалить контейнеры

**Seminar01**

mkdir ~/testfolder – создает папку в домашней директории

Для смены корневого каталога необходимо:

1. mkdir ~/testfolder
2. mkdir ~/testfolder/bin
3. cp /bin/bash ~/testfolder/bin
4. mkdir ~/testfolder/lib ~/testfolder/lib64
5. cp /lib/x86\_64-linux-gnu/libtinfo.so.6 ~/testfolder/lib
6. cp /lib/x86\_64-linux-gnu/libc.so.6 ~/testfolder/lib
7. cp /lib64/ld-linux-x86-64.so.2 ~/testfolder/lib64/
8. sudo chroot ~/testfolder – команда смены корневого каталога

Для того, чтобы работала команда ls в новом каталоге, необходимо:

1. Вернутся в основной каталог командой exit
2. cp /bin/ls ~/testfolder/bin/
3. cp /lib/x86\_64-linux-gnu/libselinux.so.1 ~/testfolder/lib/
4. cp /lib/x86\_64-linux-gnu/libpcre2-8.so.0 ~/testfolder/lib/
5. sudo chroot ~/testfolder

Все команда ls работает

ps aux – просмотр запущенных процессов

sudo ip netns add testns – команда для создания пространства имен для сети

sudo ip netns exec testns bash – команда для входа в изолированное пространство сети

sudo unshare --net --pid --fork --mount-proc /bin/bash – команда для создания пространства имен процессов (unshare – команда для создания изолированных пространства имен; --net – для изоляции сетевого пространства имен; --pid – для изоляции пространства по процессам; --fork – для изоляции по памяти; --mount-proc – для изоляции процессов)

*От преподавателя:*

Изолированное выполнение процессов с помощью chroot и пространства имен в Linux Подготовка к использованию chroot:

1. Для начала, создадим каталог "testfolder" в домашнем каталоге пользователя и скопируем необходимые исполняемые файлы и библиотеки в эту папку:
2. mkdir ~/testfolder mkdir ~/testfolder/bin
3. cp /bin/bash ~/testfolder/bin mkdir ~/testfolder/lib ~/testfolder/lib64
4. cp /lib/x86\_64-linux-gnu/libtinfo.so.6 ~/testfolder/lib
5. cp /lib/x86\_64-linux-gnu/libc.so.6 ~/testfolder/lib cp /lib64/ld-linux-x86-64.so.2 ~/testfolder/lib64/
6. Запуск команды chroot: Запустим команду chroot для изменения корневой папки нашей текущей среды: sudo chroot ~/testfolder

Примечание: Убедитесь, что вы являетесь владельцем папки "testfolder" и имеете права на выполнение команды chroot. Изменение корневой папки:

Теперь мы находимся в изолированной среде с корнем, отличным от основной файловой системы. Оболочка интерпретатора Bash запущена в этой изолированной среде. ls В этот момент вы можете заметить ошибку: bash: ls: command not found

Это произошло потому, что необходимо также скопировать исполняемые файлы, такие как "ls", и связанные с ними библиотеки в созданный каталог. Добавление дополнительных файлов: Для решения проблемы с отсутствием команды "ls" и других, продолжим копировать необходимые файлы:

1. cp /bin/ls ~/testfolder/bin/
2. cp /lib/x86\_64-linux-gnu/libselinux.so.1 ~/testfolder/lib/
3. cp /lib/x86\_64-linux-gnu/libpcre2-8.so.0 ~/testfolder/lib/
4. Подготовка для запуска: Теперь мы можем выполнить следующие команды для убедиться, что все необходимые файлы скопированы: ldd ~/testfolder/bin/ls

Использование chroot с дополнительными файлами:

1. Попробуем снова войти в изолированную среду через chroot: chroot ~/testfolder Теперь вы должны видеть изменение приветствия оболочки, что означает успешное выполнение команды chroot.
2. Выполните: \_ ls / Вы увидите, что теперь команда "ls" работает. Пространство имен: Однако стоит отметить, что использование chroot имеет недостатки, такие как необходимость дублирования всех необходимых файлов.

Вместо этого, можно рассмотреть использование механизма пространства имен. Пространство имен в Linux обеспечивает изоляцию процессов друг от друга, и может быть более эффективным способом управления изоляцией и ресурсами. Вышеупомянутые инструменты, такие как Docker, также предоставляют более надежные и гибкие средства для изоляции процессов и управления ресурсами. Пример использования сетевого пространства имен: Создание Пространства Имен для Сети:

1. Воспользуемся командой ip для создания сетевого пространства имен. Давайте создадим пространство имен с именем "testns": \_ ip netns add testns Это создаст изолированное сетевое окружение, похожее на свитч, к которому можно подключить процессы.
2. Запуск Процесса в Пространстве Имен: Используя команду ip, мы можем выполнить процесс в созданном пространстве имен: \_ ip netns exec testns bash Это подобно подключению процесса к изолированному свитчу, где процесс работает в собственной виртуальной сетевой среде.
3. Изоляция и Проверка: Внутри изолированной среды мы можем выполнить команды, такие как ip a, чтобы увидеть сетевые настройки. Однако, поскольку в этой среде нет реальных сетевых ресурсов, мы можем увидеть только виртуальные настройки. Даже в этой изолированной среде, мы по-прежнему можем использовать команду ps aux, чтобы увидеть процессы, ограниченные только пространством имен.
4. Просмотр Процессов: Выполнив команду ps aux, мы можем увидеть список всех процессов в текущем пространстве имен. Однако они будут ограничены только к процессам, которые работают в данной изолированной области.

Часть 2: Более Глубокая Изоляция Применяя дополнительные параметры, мы можем углубить уровень изоляции:

1. Изоляция по Процессам и Файловой Системе: \_\_ unshare --net --pid --fork --mount-proc /bin/bash

ps aux unshare Утилита которая позволяет это разграничивать - --net — ограничевает сетевое пространство имен ip a -mount-proc — разграничивает процессы ps aux --fork — изолирует память --pid — изолирует дерево процессов Формально мы внутри контейнера ls ls / ps aux``

**Lesson03**

Установка Docker выполняется по инструкции на сайте <https://docs.docker.com/engine/install/ubuntu/>

docker --help - выведет все доступные команды

docker --version – выведет версию docker

docker info – выведет всю информацию о docker

docker pull ubuntu – получит образ, но не установит его. Для выполнение команды нужно быть root

docker images – покажет полученные образы

docker rmi ubuntu:22.04 – удалит указанный образ

docker rmi $(docker images -aq) – удалит все полученные образы. Не получится удалить образ, который запущен.

docker images –aq – выводит id образов

docker rmi $(docker images -aq) --force – в случае если система не может удалить какой либо образ, его можно удалить добавив --force.

docker run -it ubuntu bash – запустит образ ubuntu и в нем выполнить команду bash, т.е. запустит оболочку (командный интерпритатор). –it говорит о запуске контейнера в интерактивном режиме

docker run ubuntu date – пакажет дату запущенную из контейнера

docker run --publish 8080:80 nginx – запуск nginx

docker run -d --publish 8080:80 nginx – запуск nginx в фоновом режиме

docker rm имя контейнера – удаление контейнера

docker rm $(docker ps -a -q) – удалит не запущенные контейнеры

ps -a – выведет все доступные в системе контейнеры

ps -q – выведет id запущенные контейнеры

ps -a -q – выведет id всех доступных контейнеров

docker ps –a – покажет id запущенного контейнера

docker stop id процесса – останавливает контейнер с указанным id. После повторяем команду docker rm $(docker ps -a -q).

docker system df – покажет место занимаемое контейнерами

docker system prune –af – удалит контейнеры и образы, которые остановленные и не используются

docker run -d ubuntu sleep 15 – запустит контейнер, который будет работать указанное время sleep 15 (15 секунд)

**Lesson04**

apt install cowsay – установка приложения cowsay

/usr/games/cowsay "GeekBrains" – запуск приложения

cat > Dockerfile – создаем Dockerfile с содержанием:

FROM ubuntu:24.04

RUN apt-get update

RUN apt-get install -y cowsay

RUN ln -s /usr/games/cowsay /usr/bin/cowsay

CMD ["cowsay"]

docker build -t cowsaytest . – запускаем файл Dockerfile на исполнение

docker run -it cowsaytest bash – запускаем созданный контейнер

hostname – покажет hostname

cowsay "Hi" – запускаем приложение в контейнере



docker run cowsaytest cowsay "GeekBrains" – запускаем приложение без захода в контейнер

Оптимизируем контейнер для уменьшения веса. Объединим команды и теперь Dockerfile выглядит:

FROM ubuntu:24.04

RUN apt-get update && \

apt-get install -y cowsay && \

ln -s /usr/games/cowsay /usr/bin/cowsay && \

rm -rf /var/lib/apt/lists/\*

CMD ["cowsay"]

Уменьшился его вес

**Seminar02**

sudo apt-get install lxc debootstrap bridge-utils lxc-templates – установка контейнера lxc

sudo apt-get install lxd-installer – установка lxd

sudo lxd initt –инициирование lxd

sudo apt install snapd

sudo snap install core

sudo lxc storage list

sudo apt update

sudo lxc-create -n test123 -t Ubuntu – создания контейнера

sudo lxc-start -n test123 – запускаем контейнер

sudo lxc-attach -n test123 – заходим в контейнер

free –m – проверяем память

exit – вход из контейнера

sudo lxc-stop -n test123 –останавливаем контейнер

sudo nano /var/lib/lxc/test123/config – открываем файл конфигурации

Добавляем в конец файла строку lxc.cgroup2.memory.max = 256M и снова запускаем контейнер, заходим в контейнер и вводим команду free –m и видим ограничение памяти



* У кого не меняется память, попробуйте запустить контейнер с указанием пути к конфигу:  
  sudo lxc-start -n test123 -f /var/lib/lxc/test123/config  
  Мне помогло

Начало формы

Конец формы

udo apt **update**

apt-**get** install lxc debootstrap bridge-utils lxc-templates

apt-**get** install lxd-installer

lxd init(Здесь просто нажимаем на Enter что уствновились значения по умолчанию)

Проверяем

lxc storage list

sudo apt **update**

Создаем

lxc-**create** -n test123 -t ubuntu *--создаем контейнер*

lxc-**start** -n test123 *-- запускаем*

lxc-attach -n teat123 *-- войдем в него*

**free** -m —проверяем пямаять

exit *-- выходим*

lxc-stop -n test123 - -закрываем

nano /var/lib/lxc/test123/config-открываем

lxc.rootfs.path = dir:/var/lib/lxc/test1234/rootfs — путь

lxc.uts.name = test1234 *-- имя*

Network configuration — Конфегурация сети

.

.

.

lxc.cgroup2.memory.max = 256M *-- ограничиваем(В режиме Вставка делаем запись)*

lxc-**start** -n test123 *-- запускаем*

lxc-attach -n teat123 *-- войдем в него*

**free** -m *-- проверяем пямаять*

!!!Видим что наше ограничение работает!!

sudo lxc-stop -n HW02 – останавливаем контейнер

sudo lxc-destroy -n HW02 – удаляем контейнер

**Lesson05**

docker run --name some-mysql -e MYSQL\_ROOT\_PASSWORD=test123 -d mysql:8.0.31 - запускаем docker

docker run --name myphp -d --link some-mysql:db -p 8081:80 phpmyadmin/phpmyadmin

Создадим файл nano compose.yaml со следующим содержанием:

version: ‘3.9’

services:

db:

image: mariadb:10.10.2

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: 12345

adminer:

image: adminer:4.8.1

restart: always

ports:

- 6080:8080

docker-compose build – собирает сервисы, которые прописаны в конфигурационных файлах

docker-compose up – разворачивает и запускает проекта в фоновом режиме

docker-compose up -d – запуск проекта в режиме демон

docker-compose start – запускает ранее остановленные проекты

docker-compose down – останавливает проект и удаляет все сущности проекта

docker-compose stop – просто останавливает проект

docker-compose logs –f <service\_name> - позволяет собрать логи сервиса

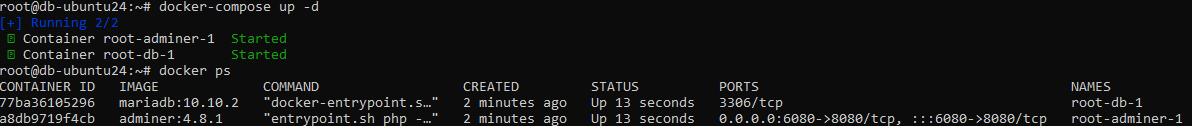
docker-compose exec – выполняет команду в сервисе не заходя в контейнер

docker-compose images – выводит список образов

Для выполнения команды docker-compose up сначала sudo apt install -y docker.io, потом sudo snap install docker и только после этого выполняем docker-compose up.

docker-compose up -d – запускаем в режиме демон

Проверяем по процессам



docker container inspect root-adminer-1 – проверяем работу контейнера

docker container inspect root-db-1 – также проверяем второй контейнер

**Seminar03**

sudo apt update

sudo apt install apt-transport-https ca-certificates curl software-properties-common – установка пакетов для использования репозитория

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg – добавление gpg ключа для репозитория

echo "deb [signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null – добавляет докер в список источников пакетов

sudo apt update

sudo apt install docker-ce – устанавливаем docker-ce

sudo usermod -aG docker $USER – чтобы докер не запрашивал пароль с sudo

newgrp docker – перезагрузка докер

docker –version – покажет версию докера

docker run -it -h GB --name gb-test ubuntu:22.10 – запуск контейнера в интерактивном режиме, т.е. будем работать внутри контейнера. Если образа нет, то он скачивается с репозитория.

Если появляется ошибка docker: Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?.  
See 'docker run --help'., то необходимо запустить команду sudo systemctl start docker, которая запускает контейнер в ручную. И потом запусить sudo systemctl enable docker чтобы потом докер запускался автоматически.

docker ps –a – посмотреть все контейнеры

docker stop gb-test – остановка контейнера

docker rm gb-test – удаление нашего контейнера по имени

docker rm b017a80903c0 - удаление нашего контейнера по id

docker rm $(docker ps -aq) --force – удалит все даже остановленные контейнеры. В скобках указано, что возвращать при удалении. -aq – покажет id даже остановленных контейнеров.

docker images – покажет все скаченные образы

mkdir /example – создали папку

touch /example/passwords.txt – создали файл

echo "123test" >> /example/passwords.txt – записали текст

cat /example/passwords.txt – проверили содержимое

Если выйти из контейнера (остановить его), то созданный файл сохранится

docker start gb-test – запускаем контейнер

docker exec -it gb-test bash – заходим в контейнер

sudo mkdir /test

sudo mkdir /test/folder

docker run -it -h GB --name gb-test -v /test/folder:/otherway ubuntu:22.10 – запускаем контейнер в интерактивном режиме, -v /test/folder – связываем папку folder с папкой otherway контейнера. : - указывается до двоеточия папку в системе, после двоеточия папку в контейнере. При запуске контейнера папка otherway автоматически создастся

echo "$HOSTNAME" >> /otherway/test.txt – создаем файл и записываем в него имя хоста, т.е. в нашем случае GB

ls / - покажет файлы в корне

От преподавателя

Обновите списки пакетов:  
\_\_  
sudo apt update  
Установите пакеты, которые позволят использовать репозиторий по HTTPS:  
\_\_  
sudo apt install apt-transport-https ca-certificates curl software-properties-common  
Добавьте официальный GPG-ключ Docker:  
\_\_  
curl -fsSL <https://download.docker.com/linux/ubuntu/gpg> | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg  
Для других дистрибутивов, замените URL на соответствующий.  
\_\_  
Добавьте репозиторий Docker к списку источников пакетов:  
\_\_  
echo "deb [signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] <https://download.docker.com/linux/ubuntu> $(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null  
Обновите список пакетов, чтобы включить информацию о пакетах Docker из добавленного репозитория:  
\_\_  
sudo apt update  
Установите Docker:  
\_\_  
sudo apt install docker-ce  
Добавьте вашего пользователя в группу docker, чтобы избежать использования sudo для запуска Docker команд:  
\_\_  
sudo usermod -aG docker $USER  
Перезагрузите систему или запустите следующую команду, чтобы применить изменения в текущем сеансе:  
\_\_  
newgrp docker  
Теперь вы должны быть готовы использовать Docker через терминал. Вы можете проверить его работу, выполнив команду:  
\_\_  
docker --version  
2 -тестируем.

3 - тестируем команды.  
\_\_  
Создание и запуск контейнеров:

docker run: Запускает контейнер из образа.  
docker start: Запускает остановленный контейнер.  
docker stop: Останавливает работающий контейнер.  
docker restart: Перезапускает контейнер.  
docker exec: Выполняет команду внутри запущенного контейнера.  
Управление контейнерами:  
docker rm $(docker ps -aq): удалит все остановленные контейнеры  
\_\_

docker ps: Просмотр списка запущенных контейнеров.  
docker ps -a: Просмотр списка всех контейнеров (включая остановленные).  
docker rm: Удаляет контейнер.  
docker logs: Просмотр логов контейнера.  
Работа с образами:  
\_\_  
docker images: Просмотр списка образов.  
docker pull: Загрузка образа с Docker Hub.  
docker build: Сборка образа из Dockerfile.  
docker rmi: Удаляет образ.  
\_\_  
4- Хранение данных в контейнерах Docker: Руководство с пояснениями  
Часть-1  
```

В данной методичке мы рассмотрим примеры хранения файлов в контейнерах Docker. Параллельно с теоретическими пояснениями, студенты также будут выполнять практические задания для лучшего понимания материала.  
\_\_  
Введение:  
Мы уже ознакомились с основами работы с Docker, умеем запускать контейнеры и управлять параметрами. Сейчас давайте разберемся с тем, как можно хранить данные в контейнерах. Это критически важно для инженеров, работающих с Docker, так как хранение данных - ключевой аспект работы.  
\_\_  
Задачи:  
\_\_  
Для начала давайте запустим контейнер из образа Ubuntu и войдем в него:  
\_\_  
docker run -it -h GB --name gb-test ubuntu:22.10  
Посмотрим содержимое корневой директории:  
\_\_  
ls -l /  
Создадим новую директорию в корне:  
\_\_  
mkdir /example  
Создадим файл "passwords.txt" и добавим в него какие-либо данные (представим, что это данные сайта или базы данных). Но что делать, если у нас нет редактора? Продолжим.  
\_\_  
touch /example/passwords.txt  
echo "123test" >> /example/passwords.txt  
Объяснение:  
Мы создали директорию и файл внутри контейнера Ubuntu.  
\_\_  
Задача:  
Давайте попробуем остановить контейнер и затем запустить его снова. Сохранятся ли наши данные?  
\_\_  
docker stop gb-test  
docker start gb-test  
docker exec -it gb-test bash  
cat /example/passwords.txt  
Объяснение:  
Наши данные сохранятся, так как мы не пересоздавали контейнер.  
\_\_  
Задача:  
Удалим контейнер и создадим его заново, используя те же команды:  
\_\_  
docker stop gb-test  
docker rm gb-test  
docker run -it -h GB --name gb-test ubuntu:22.10  
Объяснение:  
В этот раз наши данные будут утеряны, так как контейнер был удален.  
\_\_  
Задача:  
Рассмотрим наиболее интересный вариант - использование внешнего хранилища. Создадим директорию и подмонтируем ее к контейнеру:  
\_\_  
mkdir /test/folder  
docker run -it -h GB --name gb-test -v /test/folder:/otherway ubuntu:22.10  
Объяснение:  
Мы создали директорию и подмонтировали ее в контейнер, что позволило нам сохранить данные.  
\_\_  
Задача:  
Добавим данные в подмонтированную директорию:

echo "$HOSTNAME" >> /otherway/test.txt  
Объяснение:  
Мы добавили данные в подмонтированную директорию.  
\_\_  
Задача:  
Проверим доступность данных с локальной системы:  
\_\_  
cat /test/folder/test.txt  
Задача:  
Удалим контейнер и создадим его снова, подмонтировав директорию:  
\_\_  
docker rm gb-test  
docker run -it -h GB --name gb-test -v /test/folder:/otherway ubuntu:22.10  
Объяснение:  
Мы видим, что данные по-прежнему доступны.  
\_\_  
Заключение:  
Самый надежный способ хранения данных в контейнерах - использование внешних хранилищ. Важно избегать хранения важных данных внутри контейнеров, чтобы предотвратить потерю информации.  
```

Часть-2 Хранение данных в контейнерах Docker: Практическое руководство  
```

В этой части мы рассмотрим практические примеры хранения данных в контейнерах Docker. Мы также рассмотрим случаи использования монтирования папок и файлов\_\_ в контейнерах.

Задачи:  
\_\_  
Создайте папку, которую мы будем готовы смонтировать в контейнер:  
\_\_  
mkdir ~/docker-mount-example  
В этой папке создайте файл test.txt и наполните его данными:  
\_\_  
echo "This is the host test.txt file" > ~/docker-mount-example/test.txt  
В домашней директории создайте файл test.txt, который также понадобится для монтирования в контейнер, но с другим содержимым:  
\_\_  
echo "This is the root test.txt file" > ~/test.txt  
Создайте контейнер из образа ubuntu:22.10 и задайте ему имя и hostname:  
\_\_  
docker run -it -h GB --name gb-test ubuntu:22.10  
Смонтируйте ранее созданную папку с хоста в контейнер:  
\_\_  
docker run -it -h GB --name gb-test -v ~/docker-mount-example:/container-mount ubuntu:22.10  
Смонтируйте созданный ранее текстовый файл из домашней директории внутрь смонтированной папки в контейнере:  
\_\_  
docker run -it -h GB --name gb-test -v ~/docker-mount-example:/container-mount -v ~/test.txt:/container-mount/test.txt ubuntu:22.10  
Посмотрите содержимое текстового файла в контейнере:  
\_\_  
cat /container-mount/test.txt

**Seminar04**

mkdir task

nano app.py

Содержание файла app.py:

def add(x, y):  
return x + y  
  
def subtract(x, y):  
return x - y  
  
def multiply(x, y):  
return x \* y  
  
def divide(x, y):  
if y == 0:  
return "Ошибка: деление на ноль"  
return x / y  
  
def menu():  
print("Выберите операцию:")  
print("1. Сложение")  
print("2. Вычитание")  
print("3. Умножение")  
print("4. Деление")  
print("5. Выйти")  
  
while True:  
menu()  
choice = input("Введите номер операции (1/2/3/4/5): ")  
  
if choice in ('1', '2', '3', '4'):  
num1 = float(input("Введите первое число: "))  
num2 = float(input("Введите второе число: "))  
  
if choice == '1':  
print("Результат:", add(num1, num2))  
elif choice == '2':  
print("Результат:", subtract(num1, num2))  
elif choice == '3':  
print("Результат:", multiply(num1, num2))  
elif choice == '4':  
print("Результат:", divide(num1, num2))  
elif choice == '5':  
print("До свидания!")  
break  
else:  
print("Неверный ввод. Попробуйте снова.")

nano Dockerfile, следующего содержания:

FROM ubuntu:24.04

RUN apt-get update && apt-get install -y python3 – Сначала с помощью apt-get update получаем самую последнюю версию, а потом ее устанавливаем с помощью apt-get install. Т.к. режим запуска не интерактивный, поэтому ставим -y это согласие на установку python3.

WORKDIR /app – устанавливаем рабочую директорию

COPY app.py /app/ - копируем наш файл в созданную папку app

CMD ["python3", "app.py"] – запускаем наш файл, т.е. CMD будет запускаться, когда будет исполняться наш файл. CMD будет выполнена, если не указана другая команда.

docker build -t my-docker-image . – создаем образ, «.» указывает, что образ находится там, где мы работаем.

docker run -it my-docker-image – запустим созданный образ

При выходе из калькулятора (нажатие цифры 5) происходит выход из контейнера, т.к. кроме калькулятора других запущенных процессов в контейнере нет.

Для удержания контейнера в памяти, можно добавить в файл Dockerfile еще одно строку и тогда он будет иметь вид:

FROM ubuntu:24.04

# Обновляем пакеты и устанавливаем Python 3

RUN apt-get update && apt-get install -y python3

# Копируем файл "app.py" в директорию "/app/"

COPY app.py /app/

# Устанавливаем рабочую директорию

WORKDIR /app

# Определяем команду, которая будет выполнена при запуске контейнера

CMD ["python3", "app.py"]

# Добавляем команду для удержания контейнера активным

CMD ["sh", "-c", "python3 app.py & tail -f /dev/null"]

ENTRYPOINT – команда, которая будет всегда запускаться при запуске контейнера

# Используем ENTRYPOINT и CMD  
ENTRYPOINT ["python3", "app.py"]  
CMD ["--default-arg1", "--default-arg2"] – т.е. при запуске контейнера будет всегда запускаться ["python3", "app.py"], а ["--default-arg1", "--default-arg2"] нужна для определения аргументов по умолчанию, т.е. если в CMD указано ["--default-arg1", "--default-arg2"], то по умолчанию будут запускаться ["python3", "app.py"].

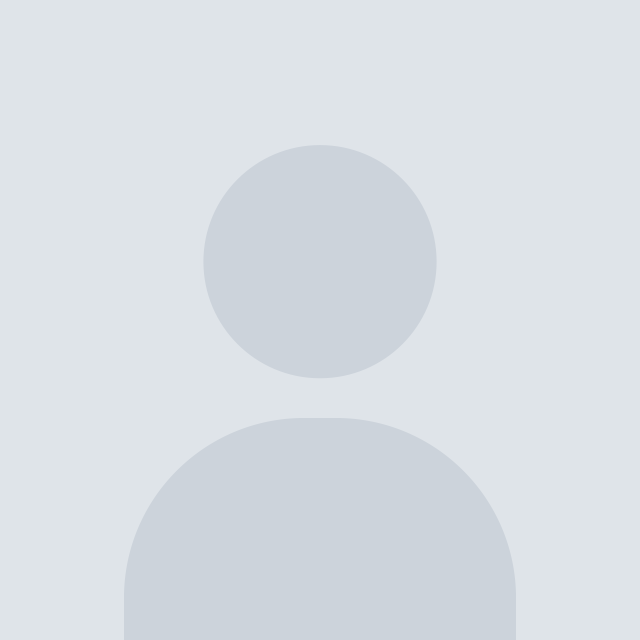
От преподавателя:

* **FROM** ubuntu:20.04
* Используйте инструкцию RUN, чтобы выполнить команды внутри контейнера во время сборки образа. Например, для обновления пакетов в Ubuntu и установки Python 3, вы можете написать:
* RUN apt-**get** **update** && apt-**get** install -y python3
* Используйте инструкцию **COPY** или **ADD**, чтобы скопировать файлы или директории из вашей локальной системы внутрь контейнера. Например, чтобы скопировать файл "app.py" из текущей директории внутрь контейнера в директорию "/app/", вы можете написать:
* **COPY** app.py /app/
* Установите рабочую директорию с помощью инструкции WORKDIR. Это указывает контейнеру, в какой директории выполнять команды по умолчанию. Например:
* WORKDIR /app
* Определите команду, которая будет выполняться при запуске контейнера, с помощью инструкции CMD. Эта команда будет выполнена, если при запуске контейнера не указана другая команда. Например, чтобы запустить приложение Python 3, вы можете написать:
* CMD ["python3", "app.py"]
* При необходимости, вы можете также использовать инструкцию ENTRYPOINT для определения точки входа, которая будет выполняться при запуске контейнера. Эта инструкция обычно используется, чтобы определить исполняемую программу, которая будет запущена, и может комбинироваться с CMD.
* После того как вы создали Dockerfile, вы можете собрать Docker-образ с помощью команды docker build. Например:
* docker build -t my-docker-image .
* После сборки образа, вы можете создать контейнер на его основе с помощью команды docker run. Например:
* docker run -it my-docker-image
* Это запустит контейнер, и он выполнит команду, указанную в инструкции CMD или ENTRYPOINT.

Начало формы

Конец формы

[ОтветитьСпасибо1](https://gb.ru/lessons/451714)

* [](https://gb.ru/users/teachers/c91db438-d6d3-4b86-bb39-69062f9cf6e2)

[Леонид Бадеев](https://gb.ru/users/teachers/c91db438-d6d3-4b86-bb39-69062f9cf6e2)  •  **Преподаватель**  •  вчера в 20:36

**def** **add**(x, y):

**return** x + y

**def** **subtract**(x, y):

**return** x - y

**def** **multiply**(x, y):

**return** x \* y

**def** **divide**(x, y):

**if** y == 0:

**return** "Ошибка: деление на ноль"

**return** x / y

**def** **menu**():

print("Выберите операцию:")

print("1. Сложение")

print("2. Вычитание")

print("3. Умножение")

print("4. Деление")

print("5. Выйти")

**while** True:

menu()

choice = input("Введите номер операции (1/2/3/4/5): ")

**if** choice **in** ('1', '2', '3', '4'):

num1 = float(input("Введите первое число: "))

num2 = float(input("Введите второе число: "))

**if** choice == '1':

print("Результат:", add(num1, num2))

**elif** choice == '2':

print("Результат:", subtract(num1, num2))

**elif** choice == '3':

print("Результат:", multiply(num1, num2))

**elif** choice == '4':

print("Результат:", divide(num1, num2))

**elif** choice == '5':

print("До свидания!")

**break**

**else**:

print("Неверный ввод. Попробуйте снова.")