МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение   
высшего образования

«КРЫМСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ им. В. И. ВЕРНАДСКОГО»

ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ

Кафедра компьютерной инженерии и моделирования

**ОТЧЕТ ПО ПРАКТИЧЕСКОМУ ЗАДАНИЮ №1**

**«Организация удалённого доступа в OC Linux.»**

Практическая работа

по дисциплине «Современные технологии программирования»

студента 1 курса группы ПИ-б-о-231(2)

Покидько Максим Сергеевича

направления подготовки 09.03.04 «Программная инженерия»

|  |  |  |
| --- | --- | --- |
|  |  |  |

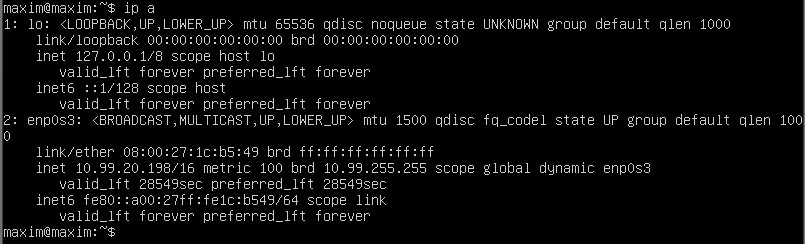
Симферополь, 2024

**Цель**: ознакомиться с базовыми возможностями Docker и Podman. Приобрести опыт и навыки контейнеризации приложений.

**Ход выполнения задания:**

**Основы Docker**

1. Для начала нужно определить название сетевого интерфейса, через который машина выходит в интернет. Введу команду ip a и найду название интерфейса:



Сетевой интерфейс называется enp0s3.

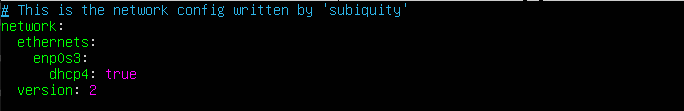


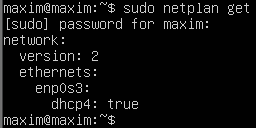
2. Чтобы установить для данного интерфейса dns можно воспользоваться 2-мя вариантами: временным и постоянным. Я воспользуюсь постоянным.

Постоянный вариант. Для этого нужно воспользоваться утилитой netplan - это основной способ настройки сети в современных дистрибутивах Ubuntu. Все конфигурационные файлы netplan хранят в директории "/etc/netplan".

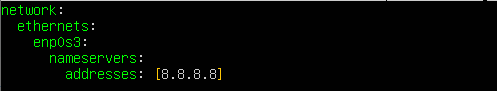
Содержимое каталога /etc/netplan – файл 00-installer-config.yaml



Содержимое этого файла:

Теперь посмотрим актуальные настройки:  


Вывод тот же.

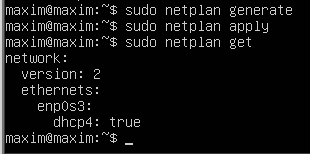
Создадим в каталоге "/etc/netplan" файл с названием "01-dns.yaml. В нём пропишем: 

Пропишем эти команды:

-sudo netplan generate

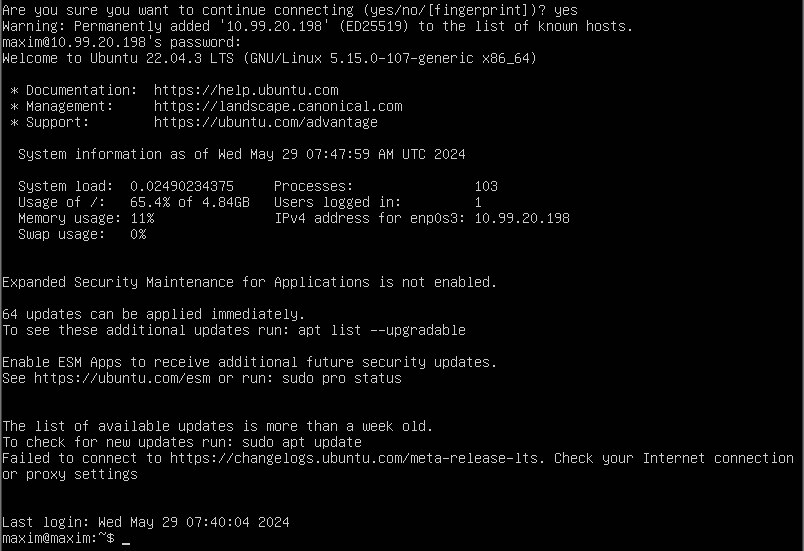
-sudo netplan apply

-sudo netplan get



**Установка компонентов Docker Engine**

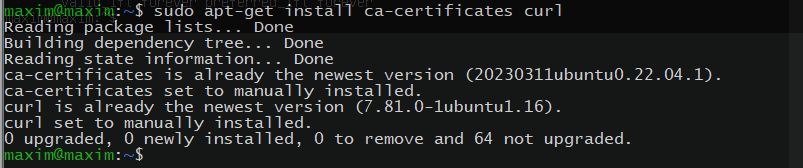
3. Подключусь к машине по ssh для удобства.



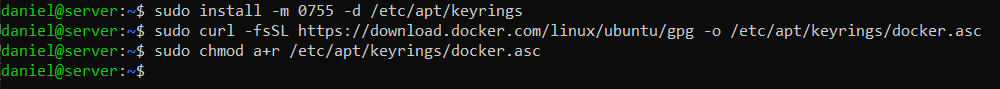
4. Обновим индексы пакетов.

5. Теперь перейдём к установке.

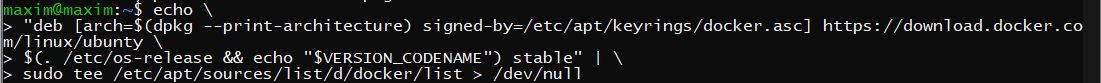
Установим некоторые необходимые пакеты:



Добавим официальный GPG-ключ для доступа к репозиторию Docker:

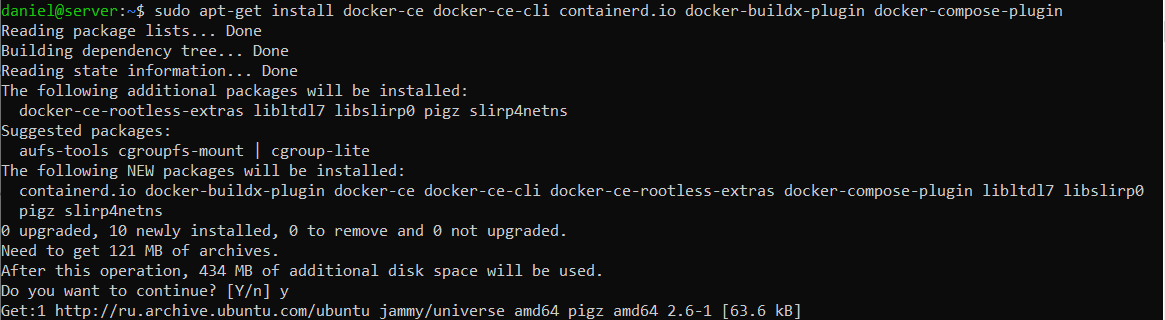


Настроим репозиторий:



После добавления нового репозитория обновим индексы пакетов

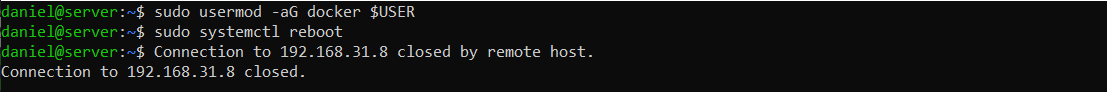
Теперь установим docker и docker compose:



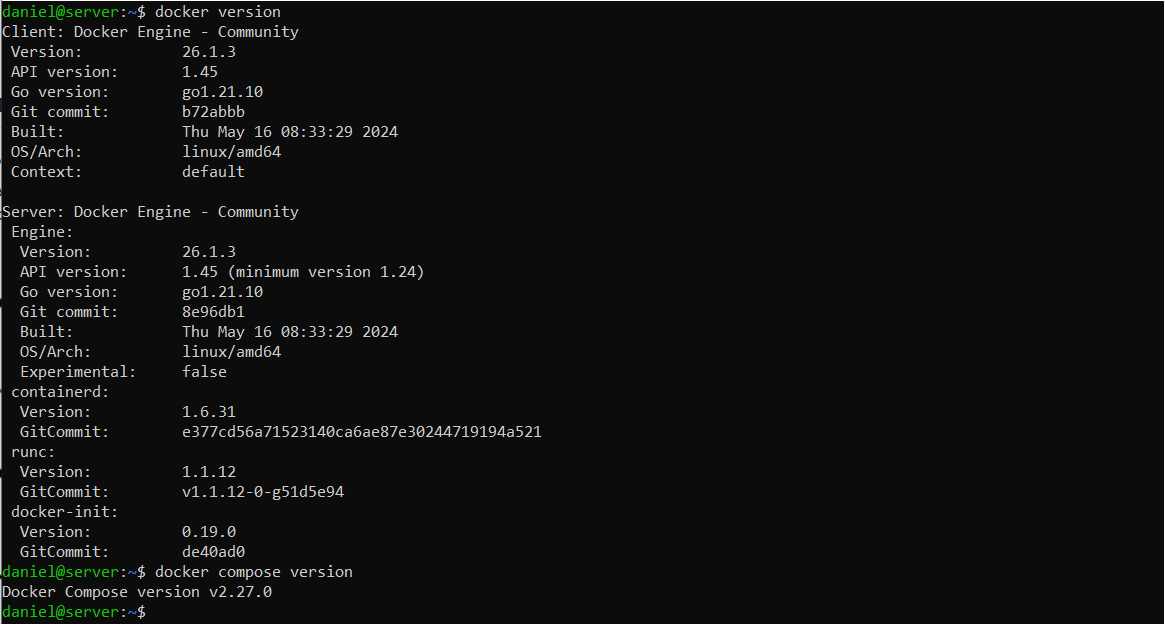
Добавим текущего пользователя в группу docker:



Теперь перезапустим сервер:

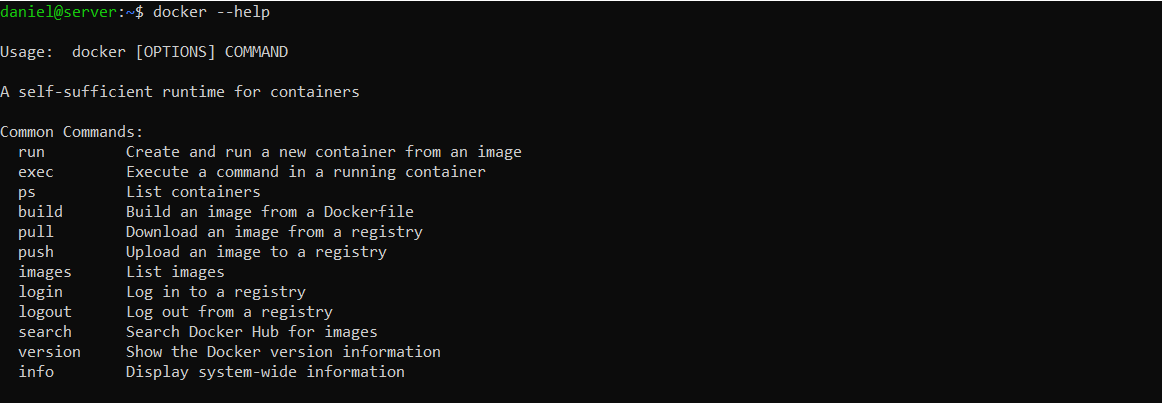


Проверим, что установка прошла успешно, посмотрев версию docker:



**Начинаем работать с Docker**

6. Выполним команду docker –help:



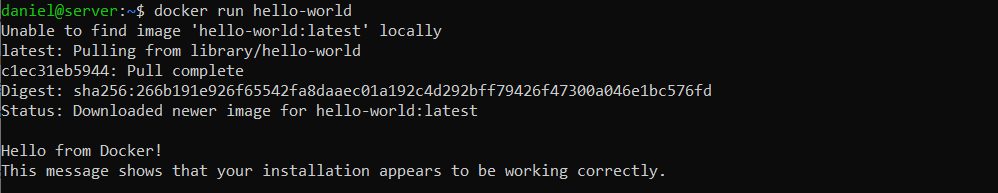
И так далее…

7. Посмотрим список образов:

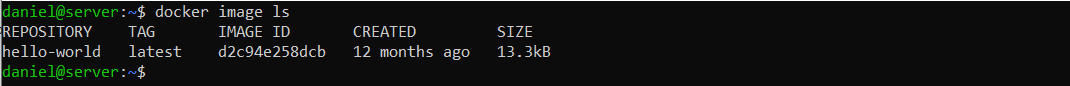


Данные пустые, т.к. пока что у меня нет ни одного образа.

8. Создадим новый контейнер из образа с именем hello-world. Докер скачает его с репозитория.

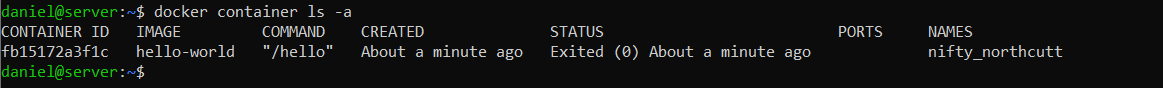


9. Теперь проверим, есть ли он в списке образов:



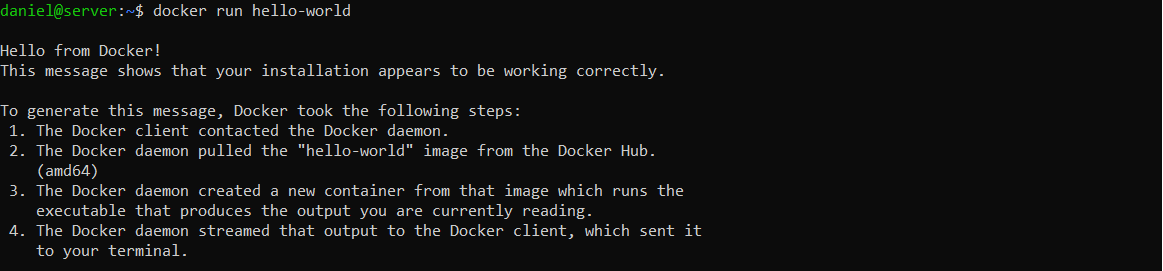
Вот и он.

10. Посмотрим список контейнеров, которые когда-либо были запущены:



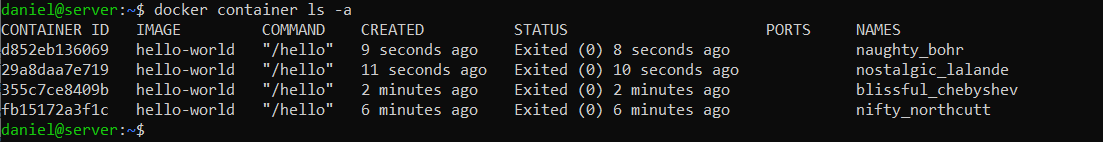
В списке есть hello-world.

11. Ещё раз запустим этот контейнер:

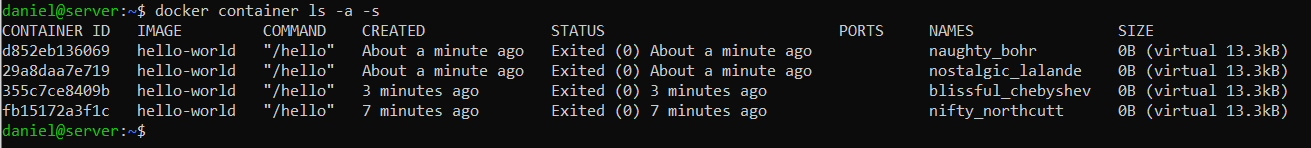


На этот раз он не упомянул о его отсутствии, потому что он уже есть на машине.

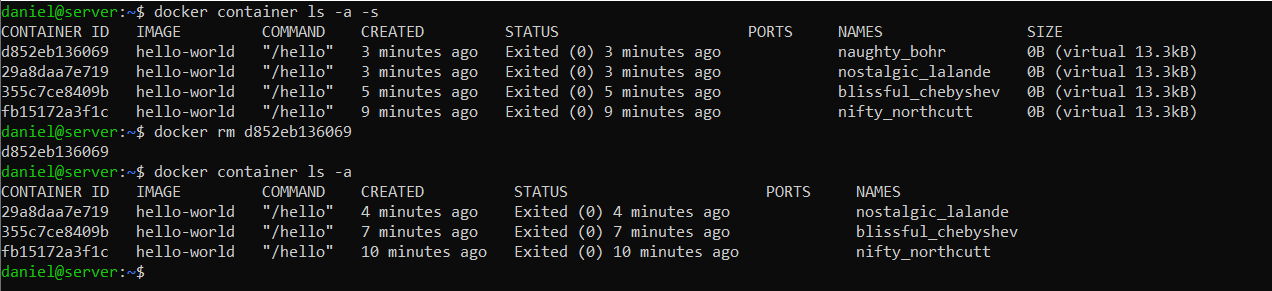
Запустим ещё пару раз и посмотрим список запущенных контейнеров:



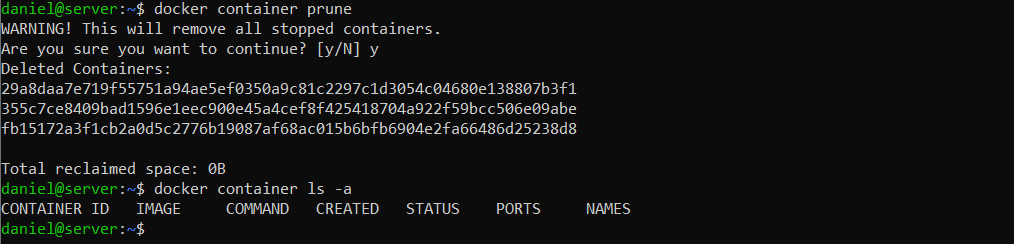
12. Добавим к команде опцию –s, чтобы посмотреть размеры самих контейнеров:



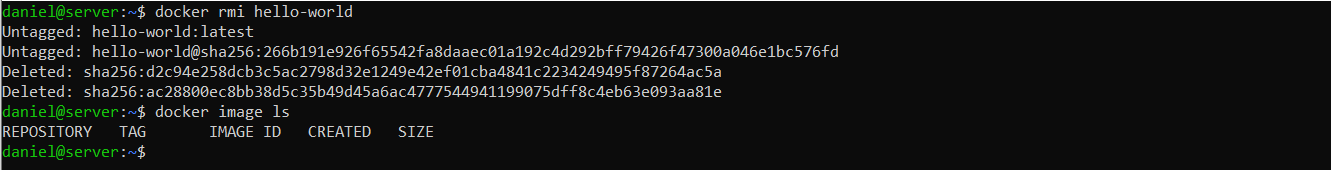
13. В результате работы у вас может накопится довольно большое количество контейнеров. И хотя каждый из них занимает мало места на диске полезно уметь их удалять. Удалю последний контейнер через его id:



14. Можно удалить все остановленные контейнеры сразу:



15. Образ hello-world больше не понадобится, поэтому можно его удалить:

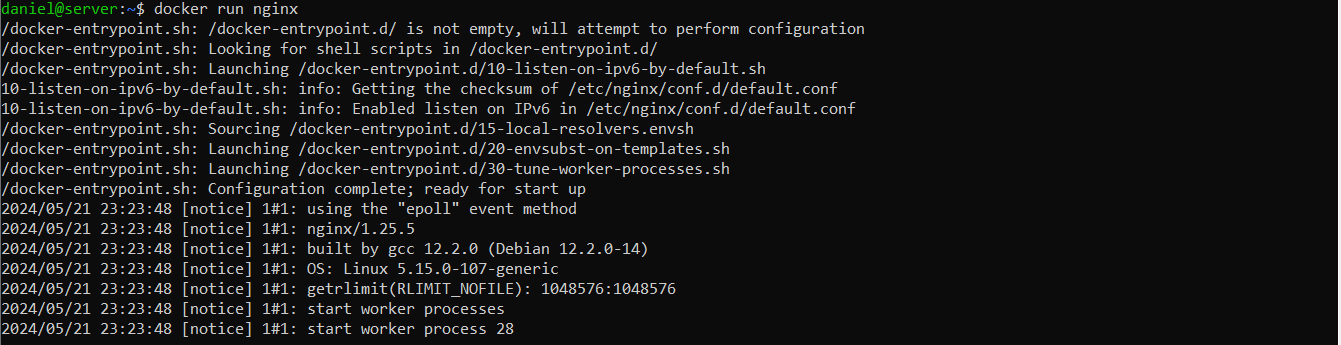


**Способы запуска контейнера**

16. Скачаем образ nginx, не запуская его:



17. Запустим контейнер из этого образа без параметров:



Эта команда запустит контейнер в режиме по умолчанию, который называется attached mode. Это означает, что стандартный ввод (stdin), стандартный вывод (stdout) и стандартный вывод ошибок (stderr) контейнера подключены к сеансу текущей оболочки. Т.е. любой вывод из контейнера немедленно выводится на наш терминал, а любой ввод с терминала отправляется в контейнер. Однако мы не можем полноценно взаимодействовать с контейнером, потому что внутри контейнера не запущена командная оболочка.

18. Т.к. весь наш ввод отправляется в контейнер, то мы не можем работать с хостовой системой. Остановим контейнер при помощи комбинации клавиш Ctrl + C.

19. Выполним у себя в терминале команду tty.



На экран вывелся путь к виртуальному устройству, с которым ассоциирован терминал.

20. Попросим контейнер выполнить команду tty:

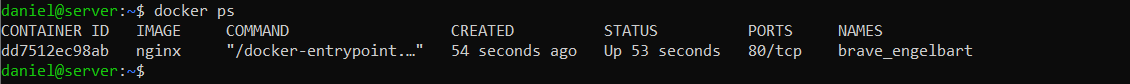


Внутри контейнера нет запущенного терминала.

21. Чтобы решить проблему захвата ввода-вывода нашего терминала контейнером запустим новый контейнер из образа "nginx:

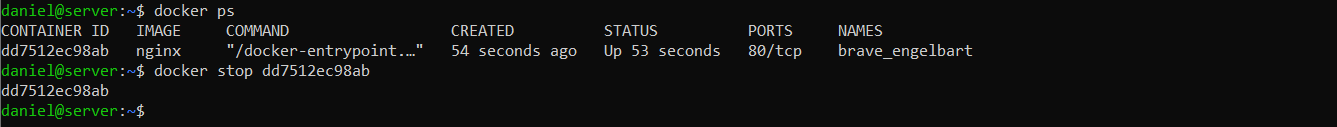


22. Выполним команду:



Команда показывает только работающие контейнеры. Сейчас это nginx.

23. Остановим контейнер, используя его id:



24. Выполним команду:

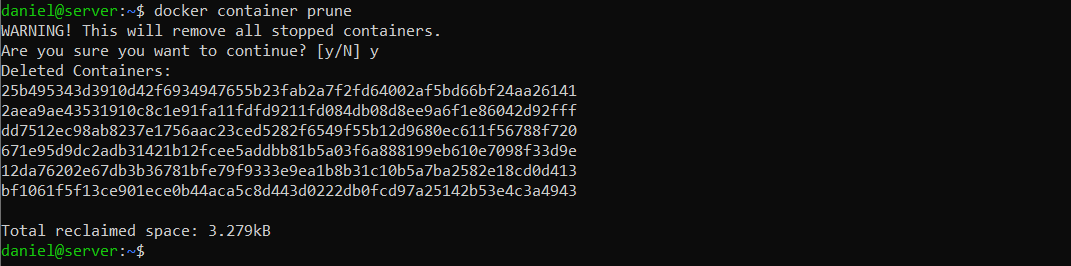


25. По виду, отличий от запуска без параметров нет, но их мы увидим далее. Остановим контейнер.

26. Убедимся, что при запуске контейнера в интерактивном режиме в нём есть терминал:

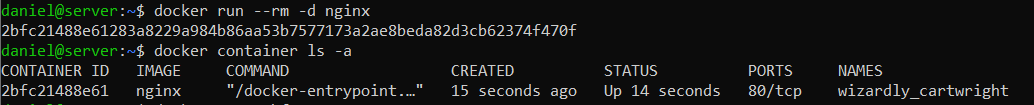


27. Удалим все созданные контейнеры:

1

28. В процессе работы у нас постоянно накапливаются ненужные контейнеры и их приходится постоянно удалять. Чтобы попросить контейнер автоматически удалиться после остановки, можно добавить к команде запуска опцию --rm.

Запустим контейнер:



29. Теперь остановим его и проверим список контейнеров:



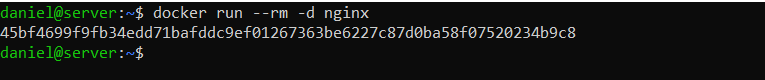
Список пуст.

**Взаимодействие с контейнером**

По умолчанию все docker контейнеры запускаются изолированными друг от друга и от хостовой системы. Поэтому далее рассмотрим как получить возможность "общаться" с контейнером.

**Выполнение команд внутри запущенного контейнера**

30. Запустим новый контейнер nginx:

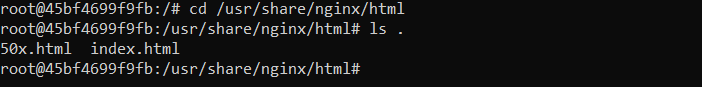


31. Выполним команду:



В результате я попал в контейнер. Последующие действия буду совершать внутри него.

32. Перейду в каталог "/usr/share/nginx/html" и посмотрю его содержимое.



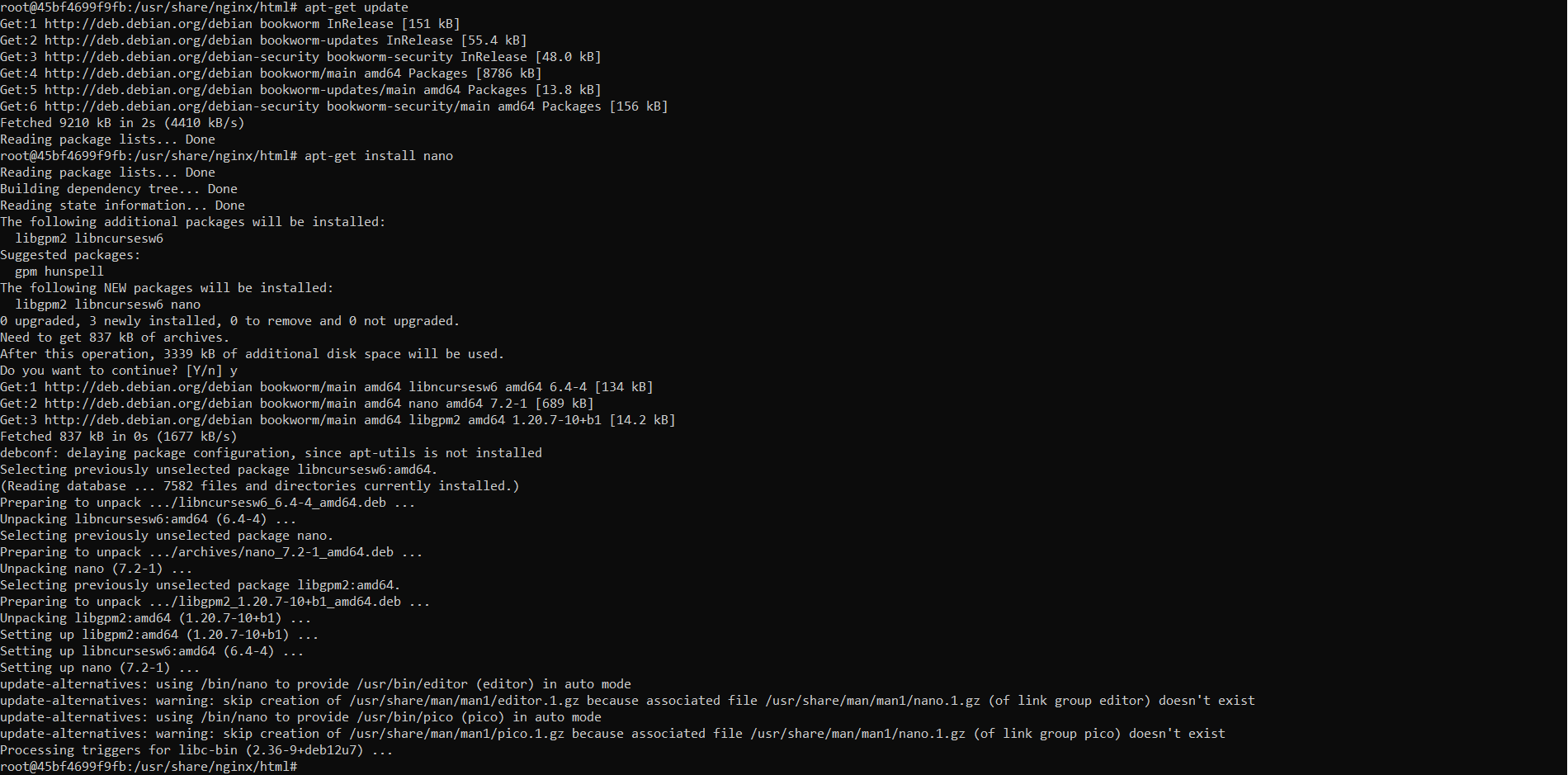
Тут лежит 2 файла – 50x.html и "index.html".

33. Попробую открыть второй с помощью nano:

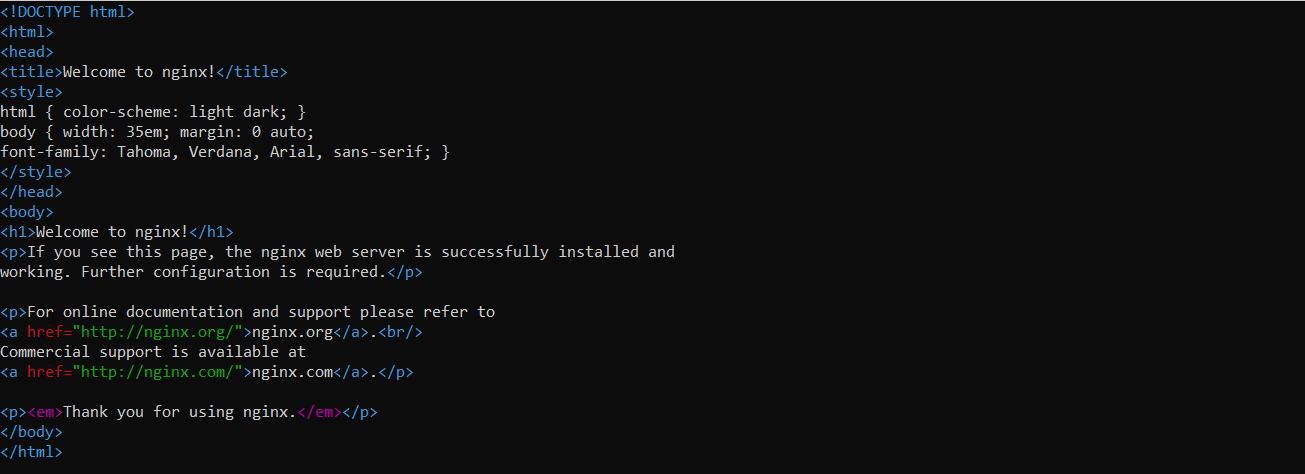


В контейнере не установлен nano.

34. Обновим индексы пакетов apt и установим nano:

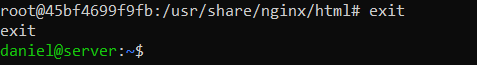


35. Теперь ещё раз попробую открыть файл:



Файл успешно открылся.

36. Выйду с контейнера.

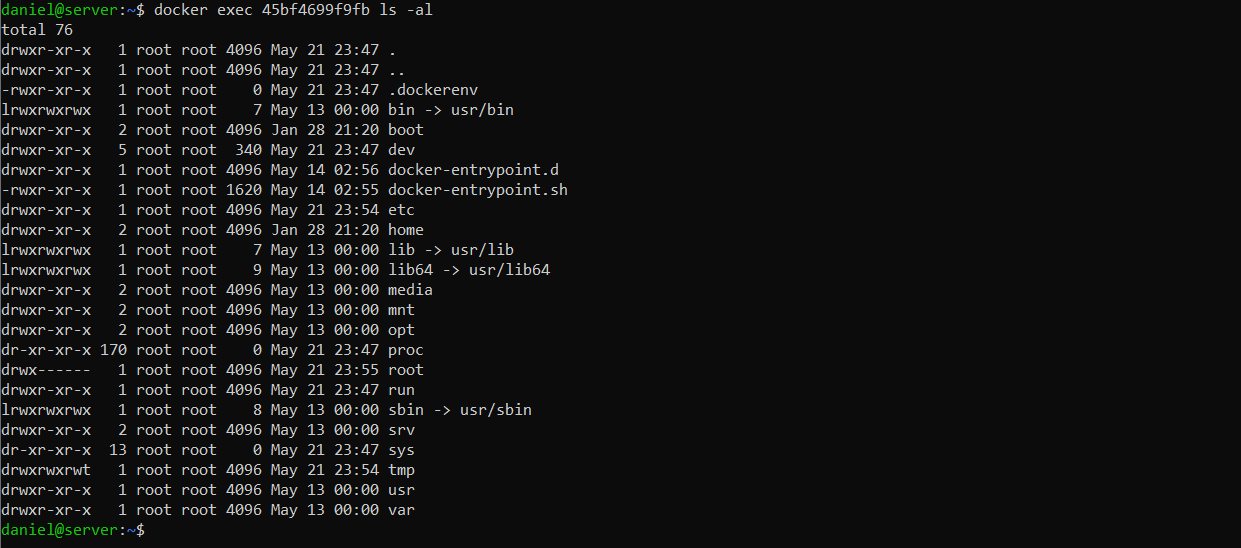


Контейнер продолжит работать дальше и при необходимости мы в любое время снова сможем к нему подключиться.

Команда exec не требует обязательного входа в контейнер для запуска команд, например:

docker exec {CONTAINER ID | CONTAINER NAME} ls -al

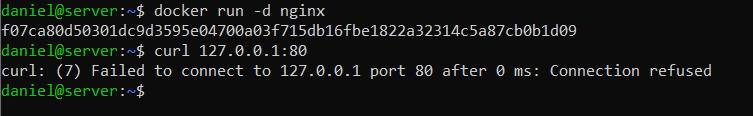
выполнит команду ls -al в контейнере и сразу отобразит её результат на экран.



**Сетевое взаимодействие**

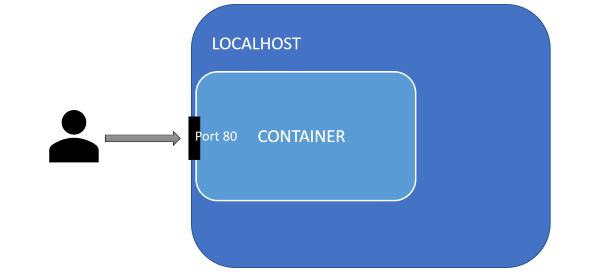
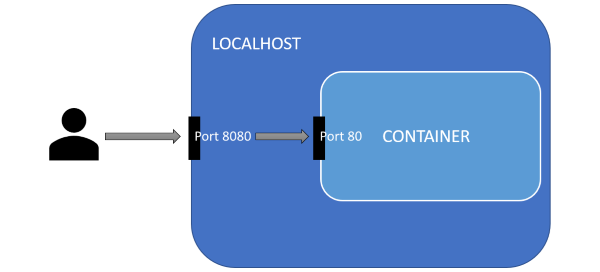
**Как общаться с одним контейнером**

37. Запущу новый контейнер с помощью –d и попробую получить ответ на GET-запрос:



Выдало ошибку, т.к. nginx хоть и слушает 80 порт но внутри своей изолированной от хоста сети.

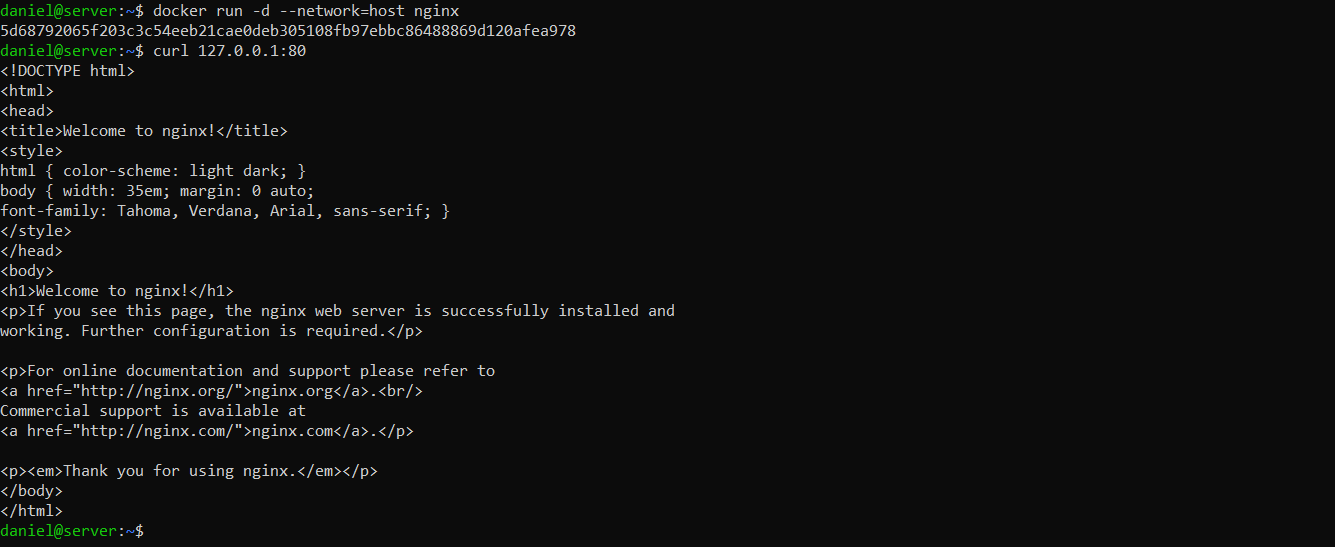
38. Чтобы решить эту проблему есть два варианта:

* подключить контейнер как приложение к хостовой сети;  
  [](https://github.com/VladimirChabanov/MPT_2023/blob/main/02_practice/task_03_subtasks/task_03.1_img/docker_ports_host.png)
* пробросить порты из хостовой сети в сеть контейнера.  
  [](https://github.com/VladimirChabanov/MPT_2023/blob/main/02_practice/task_03_subtasks/task_03.1_img/docker_ports_map.png)

39. Остановим контейнер и запустите новый, командой:

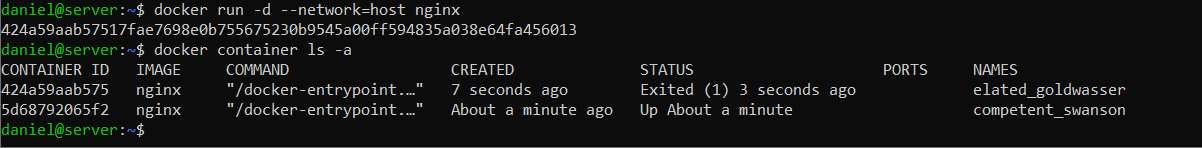


Проверим ответ:



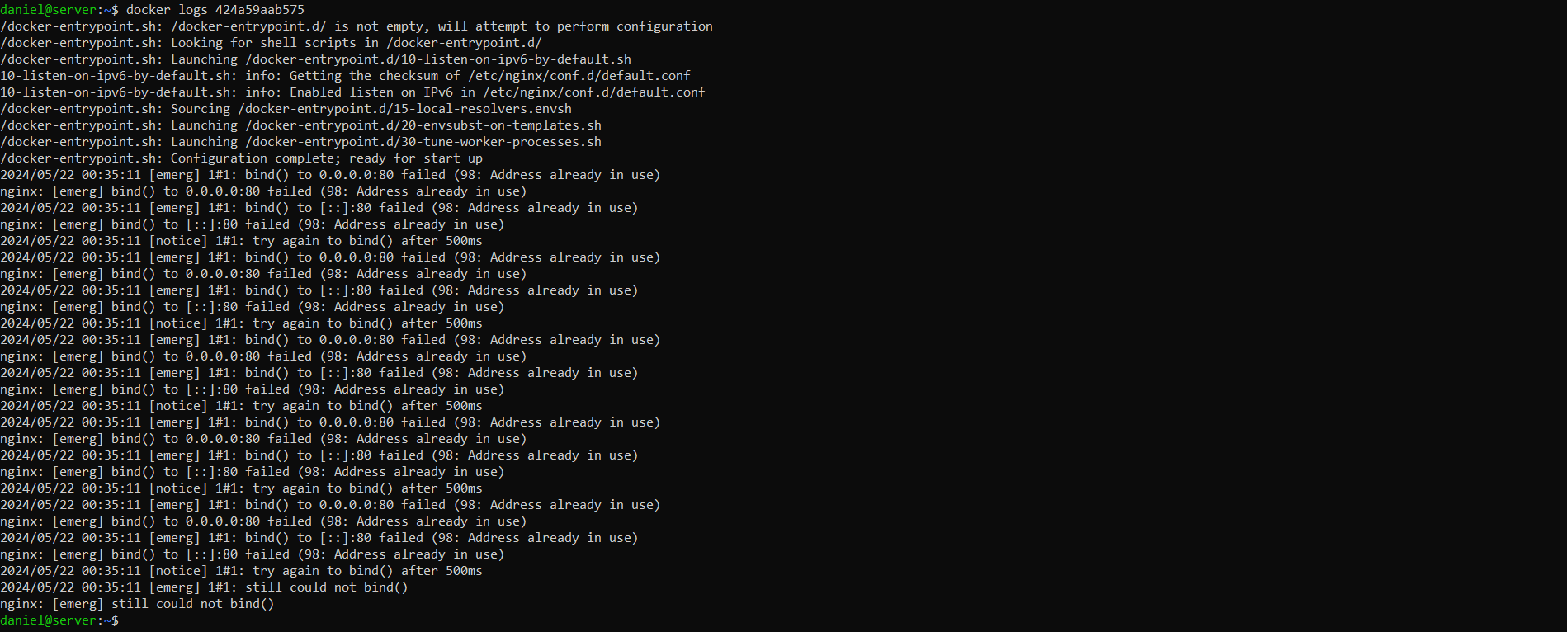
Теперь я получил в ответ код html страницы с приветствием от nginx.

40. Теперь запустим тем же способом ещё один контейнер и проверим список запущенных:



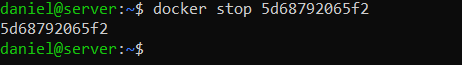
Запущен только первый, а второй почти сразу прекратил работу.

41. Чтобы понять в чём проблема воспользуемся командой:

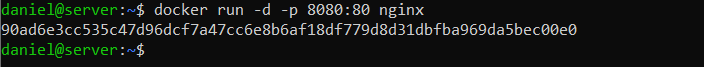


Контейнер несколько раз пробовал занять порт, но выдавало ошибку, потому что он уже используется.

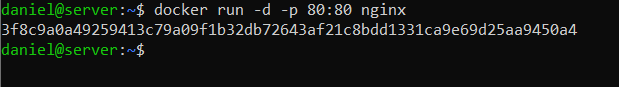
42. Остановим первый контейнер.

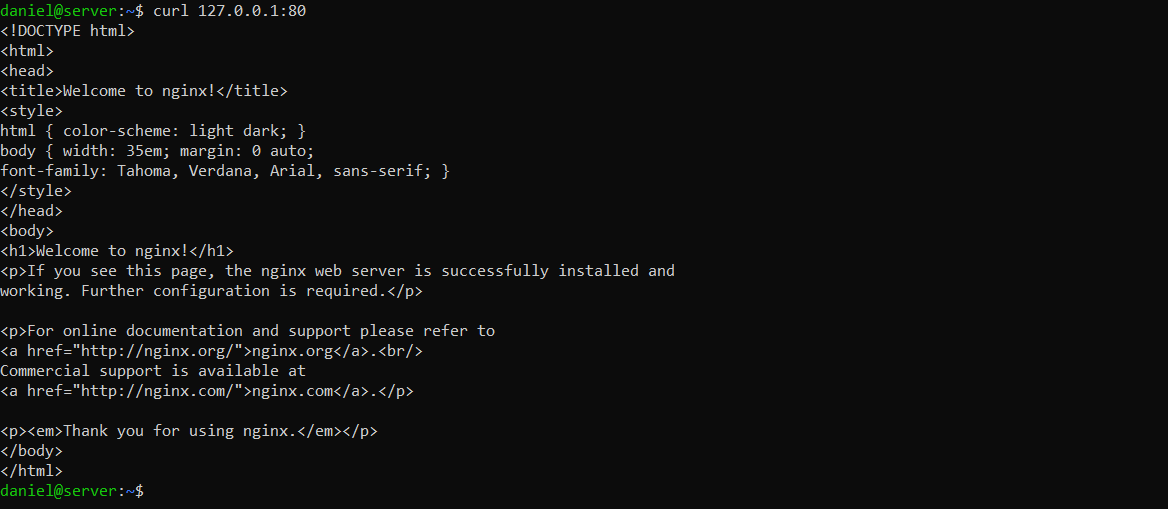


43. Более безопасным способом попасть внутрь сети контейнера является проброс портов. Введём команду:

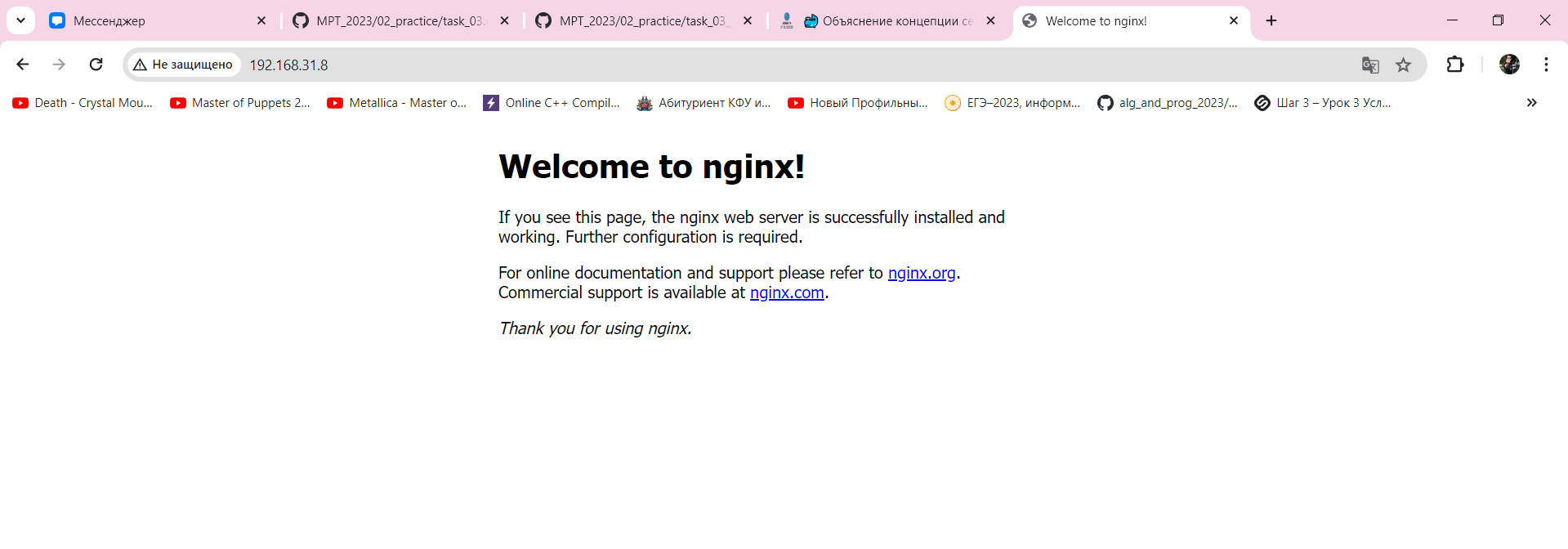


44. Запустим ещё один контейнер, но в качестве порта хоста будет 80:



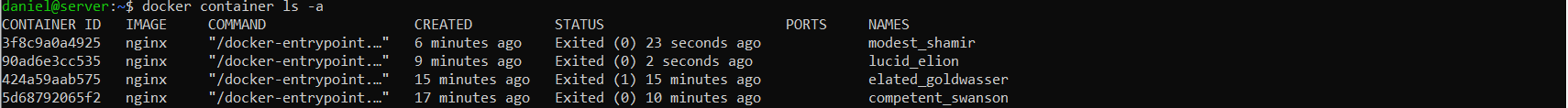


45. Введу в браузере хостовой машины ip виртуалки:



В результате открылась приветственная страница "Welcome to nginx!", что свидетельствует о том, что трафик по 80-му порту попадает через виртуальную машину в docker-контейнер и обратно.

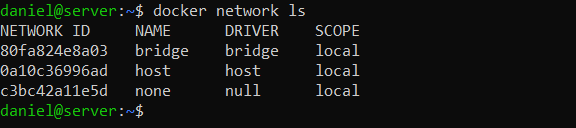
46. Остановим все запущенные контейнеры.



**Смотрим на сети**

В целом работа с сетями в docker не ограничивается пробросом портов.

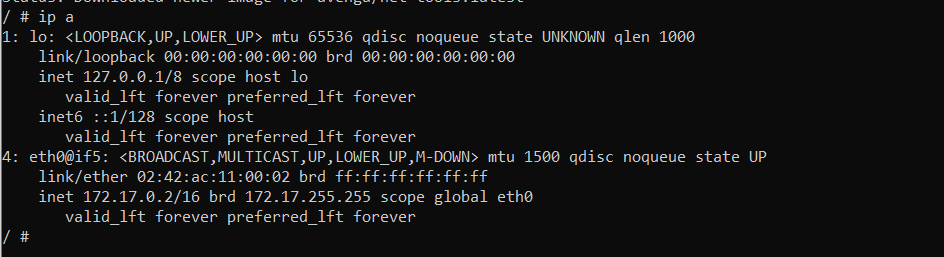
47. Посмотрим какие сети уже созданы докером:



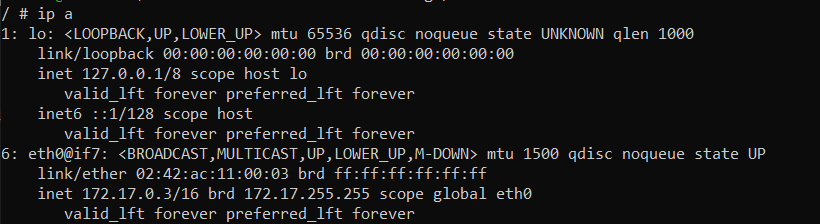
48. Запустим ещё одно окно с ssh-подключением к и в каждом окне запустим по одному контейнеру "avenga/net-tools":



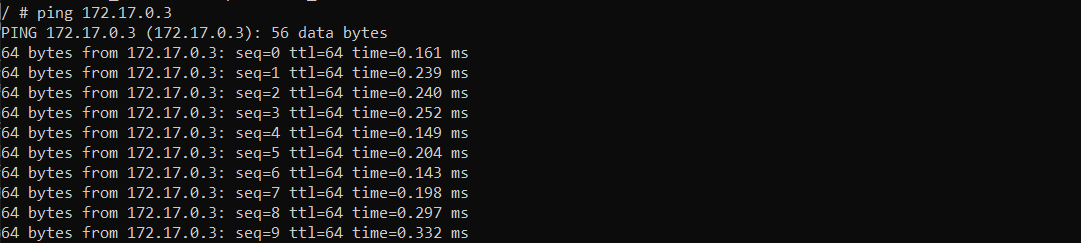
49. Посмотрим ip-адрес внутри контейнера one:



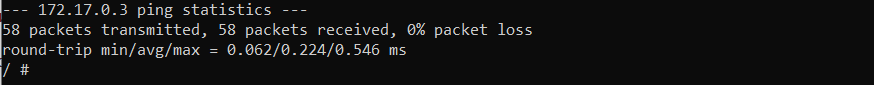
50. Теперь внутри two:



51. Пропингуем второй контейнер из первого:



Пинги идут – всё хорошо.



52. При подключении к сети, контейнеры получают IP адрес динамически, что затрудняет настройку их взаимодействия. Хотя и можно задать фиксированный IP (или свободный из диапазона) при помощи опции --ip есть способы и получше:

-Использовать опцию --link;

-Использовать встроенный dns docker-a.

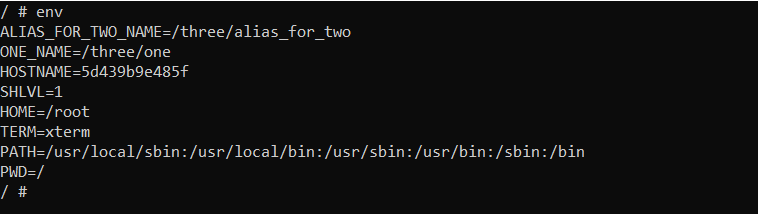
53. Для начала рассмотрим использование опции --link.

54. Ключ --link на данный момент считается устаревшим, поэтому не будем посвящать ему много времени. Данный ключ позволяет во время запуска контейнера подключить его к другим. Подключение состоит в том, что в запускаемый контейнер будет добавлен ряд переменных окружения и в его файл /etc/hosts будет добавлена новая запись (для каждого из подключаемых контейнеров). При этом, в тех контейнерах, к которым мы подключаемся никаких изменений не будет.

55. Запустим новый контейнер в новой ssh-сессии:

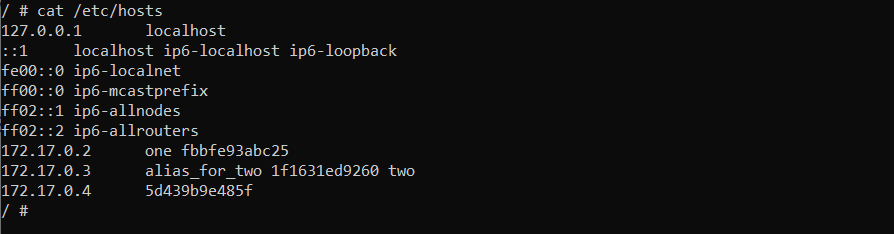


56. Получим список переменных окружения контейнера:

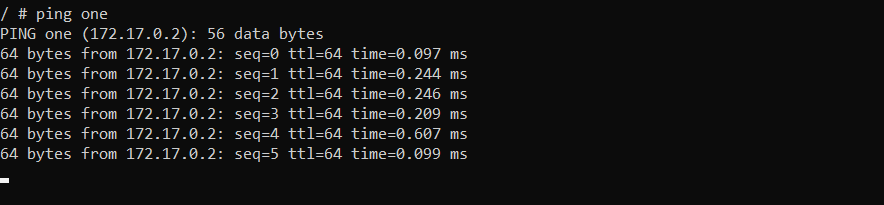


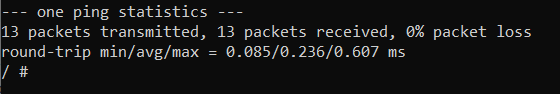
Среди них есть one и two(под именем alias\_for\_two).

57. Посмотрим содержимое файла /etc/hosts:



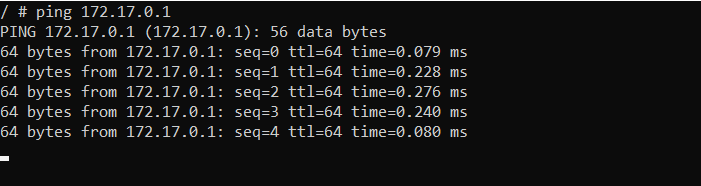
58. Пропингуем внутри контейнера three контейнер one по его имени:

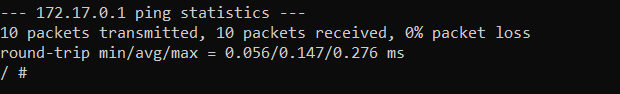




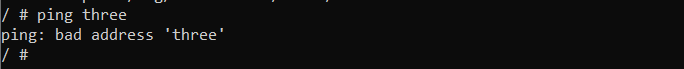
59. Теперь внутри первого контейнера пропингуем третий, сначала по ip, а потом по имени:

По ip:



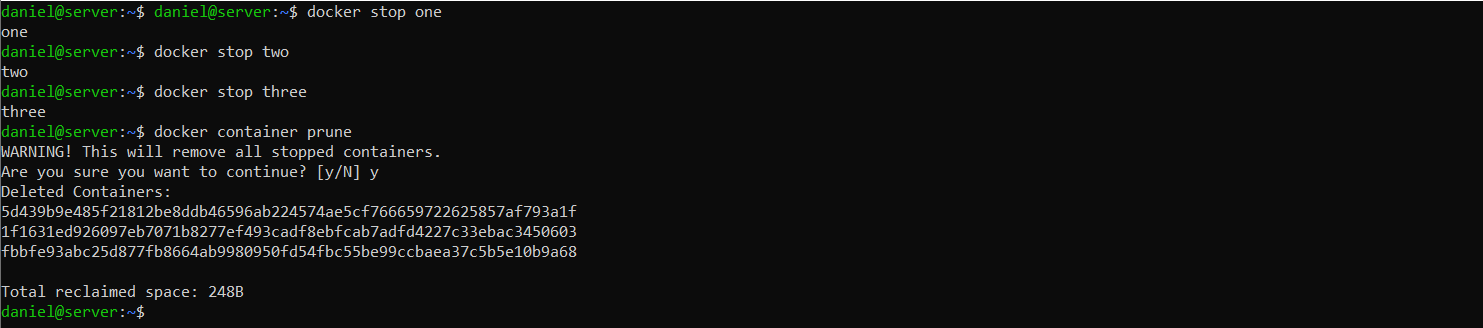


По имени:



Работает только по ip, так как --link действует только на третий контейнер.

60. Остановим и удалим все три контейнера

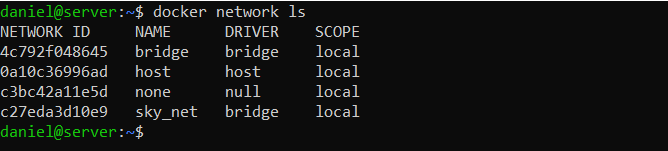


61. Более продвинутый вариант, позволяющий использовать имена контейнеров как их доменные имена в сети заключается в работе через встроенный dns docker-a. Но он не работает в сети по умолчанию, поэтому нужно будет создать свою.

62. Создадим новую сеть с настройками по умолчанию:



63. Проверим, появилась ли она в списке сетей:



Как видно, в качестве драйвера выбран bridge.

64. Запустим контейнер one:



65. В другой консоли запустим контейнер two:

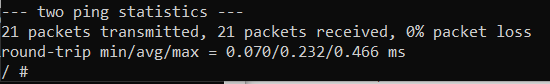
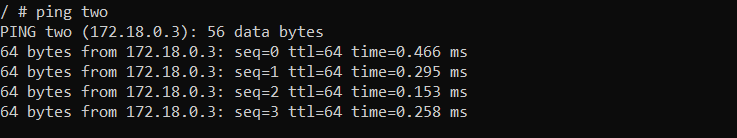


66. В третьей консоли запустим контейнер three, но без подключения к сети:

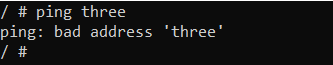


67. Проверим, что контейнер one пингует two по имени, а контейнер three нет:

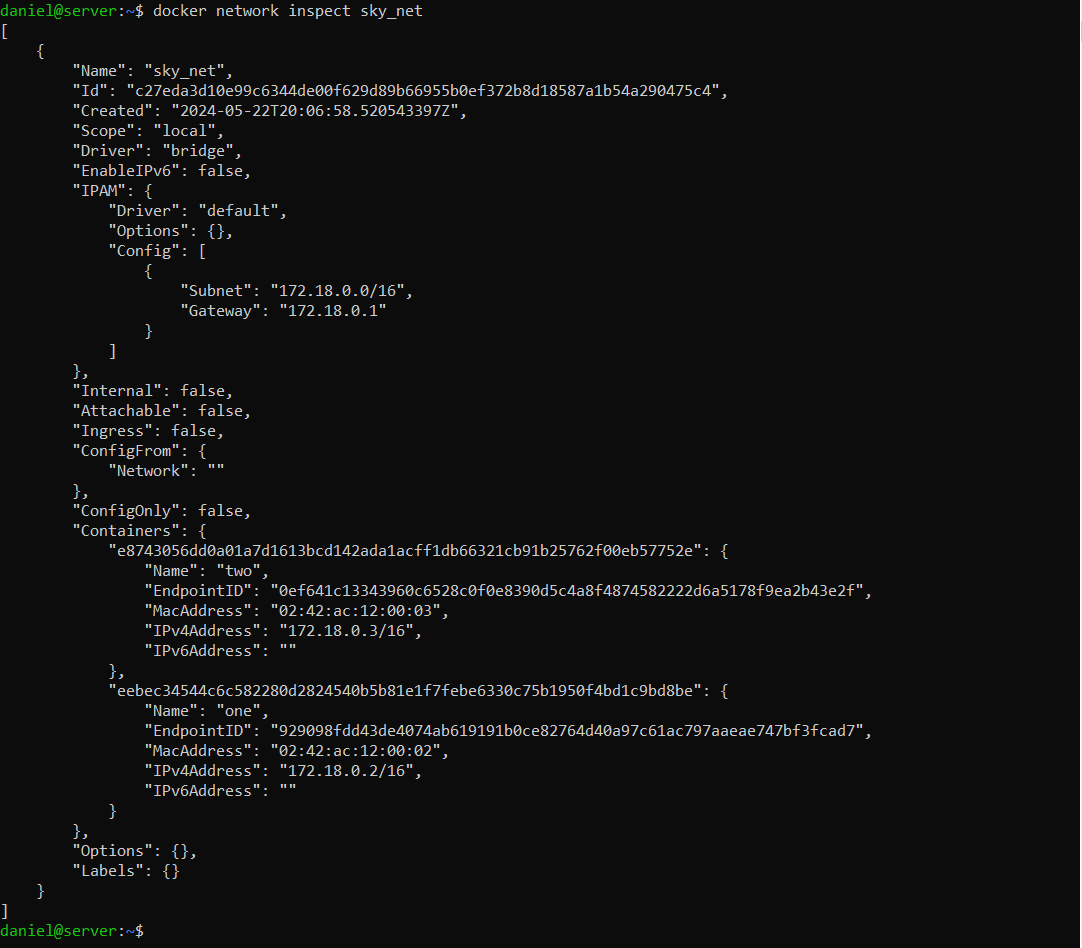
Пинг two:



Пинг three:



68. Подключимся с 4-той консоли и посмотрим информацию о сети sky\_net:



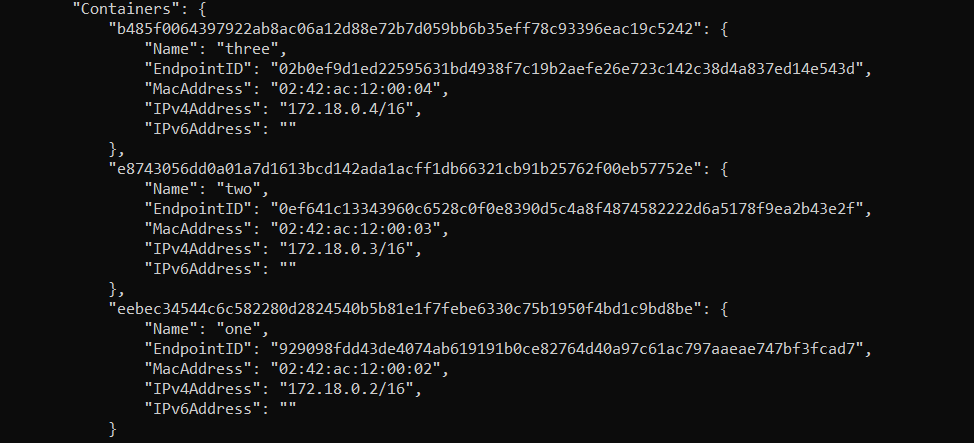
69. Найдём раздел Containers и увидим, что там присутствует информация только о контейнерах one и two.

70. Введём следующую команду:



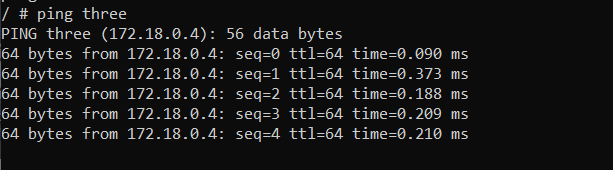
Таким образом мы добавили уже запущенный контейнер three в сеть sky\_net.

71. Посмотрим на изменения раздела Containers:



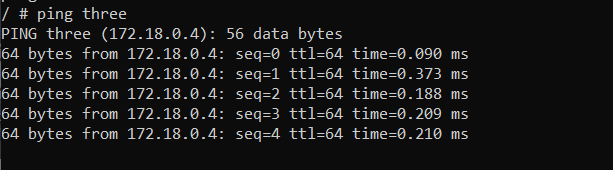
Теперь в списке есть и three.

72. Вернёмся в окно с one и пропингуем three ещё раз:



На этот раз пропинговать удалось.

73. Теперь в четвёртом окне отключим контейнер one от сети:



74. Проверим, отключило ли его, пропинговав с него другие контейнеры:

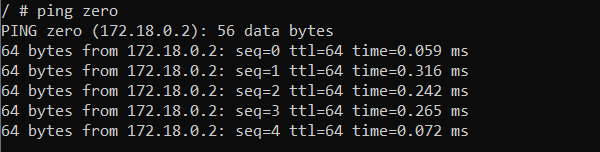
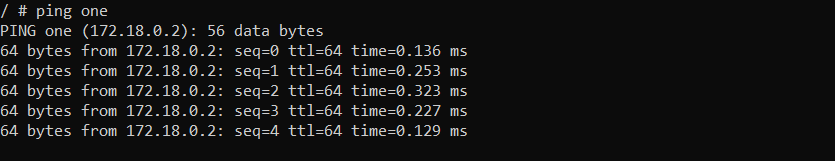


Обе попытки потерпели неудачу.

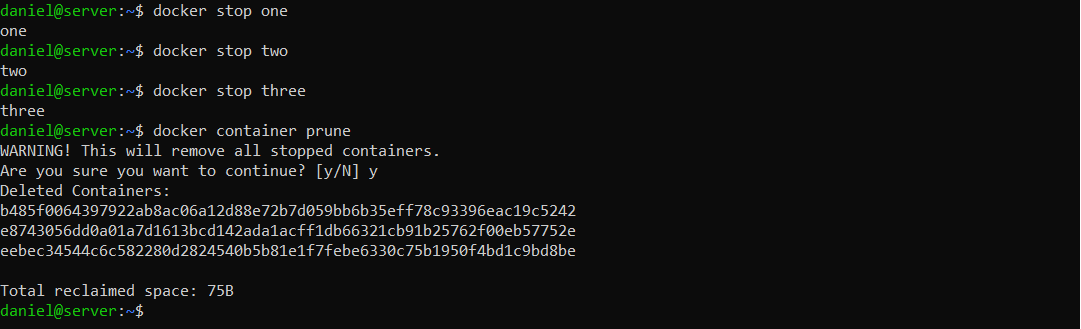
75. Вернёмся в 4-ое окно и подключим контейнер one под именем zero:



76. Теперь внутри контейнера two пропингуем контейнер one сначала под его имнем, а потом под именем zero:



77. Остановим и удалим все контейнеры:



78. Удалим сеть sky\_net:

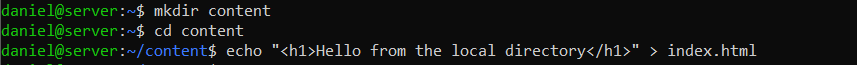


**Работа с данными**

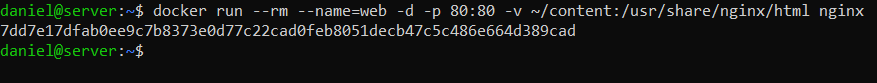
79. Для начала рассмотрим монтирование каталога внутрь контейнера.

80. Создам в домашнем каталоге пользователя папку "content" и внутри неё файл "index.html" с текстом:

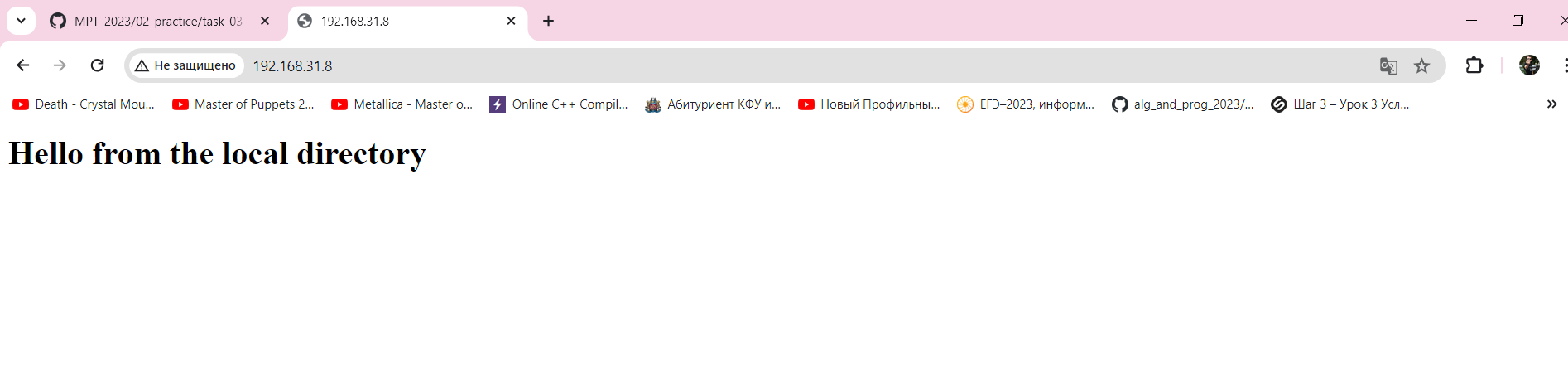
<h1>Hello from the local directory</h1>



81. Запустим контейнер "nginx" с именем "web" и монтируем локальный каталог "content" расположенный в домашнем каталоге пользователя в каталог /usr/share/nginx/html расположенный в контейнере.



82. В браузере в основной ОС введу ip-виртуальной машины в строку адреса.



83. Изменим содержимое файла "index.html" в локальной папке "content" на следующее:

<h1 style="color: red">Hello from the local directory</h1>:





84. Монтирование томов (volumes).

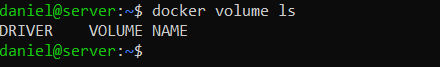
Хотя монтирование каталога выглядит вполне достаточным, чтобы решить описанную ранее проблему, у него есть ряд недостатков:

Нужно где-то создать каталог, который будет монтироваться. Нужно самому думать где это лучше сделать.

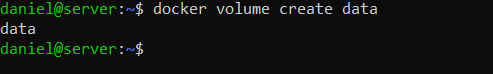
Если образ содержал какие-то файлы, то при монтировании каталога они исчезнут.

Внутри контейнера, вы часто работаете под пользователем root и способны получить доступ к любой подмонтированной директории не обращая внимание на права доступа. Например, если случайно подмонтировать в контейнер локальную папку "/etc" то через контейнер можно будет назначить своему пользователю любые права на хостовой системе.

85. Посмотрим список созданных томов:

 - список пуст.

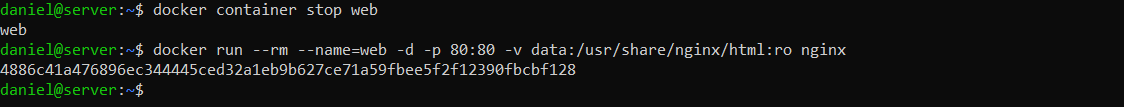
86. Создадим новый том:



87. Посмотрим информацию о томе:



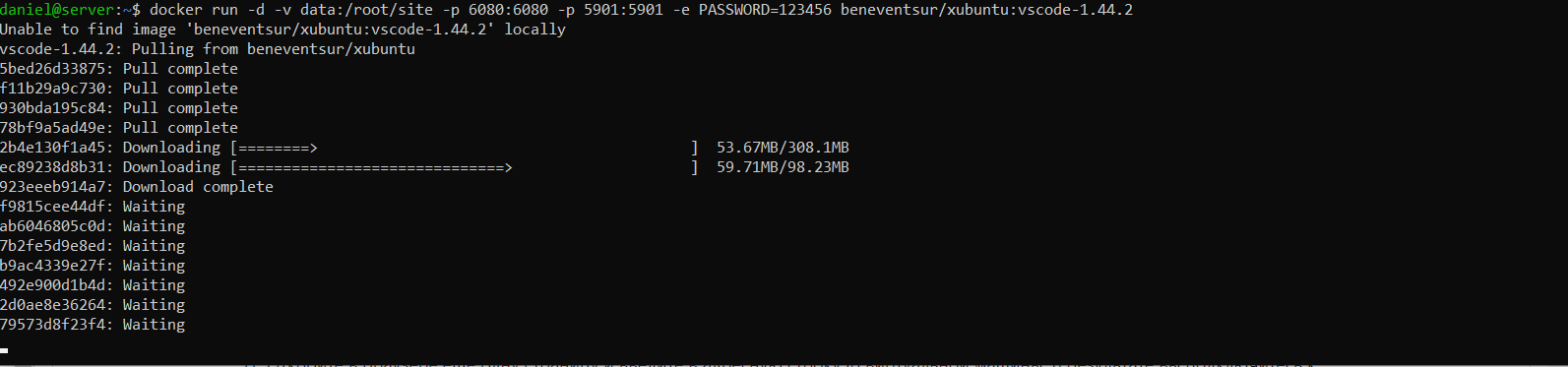
88. Остановим предыдущий контейнер и запустим новый:



89. Введём команду:

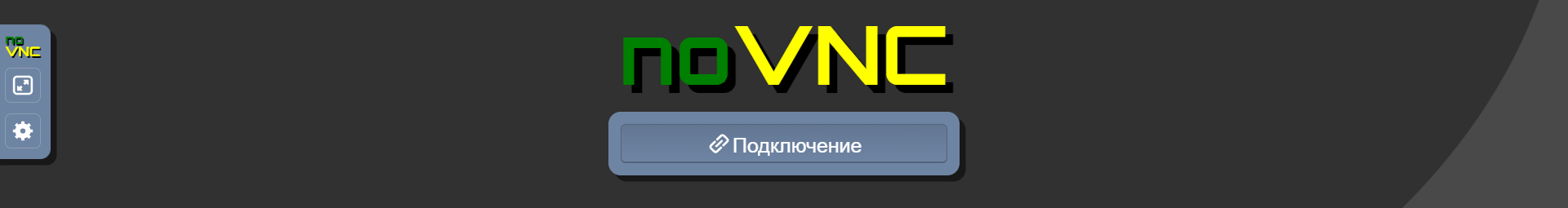


Здесь мы запускаем контейнер из образа в котором установлена среда разработки Visual Studio Code и монтируем к нему тот же том, что и к предыдущему.



90. Перейдём в браузер на основной ОС и введём в адресную строку:

http://{ip виртуальной машины}:6080/vnc.html

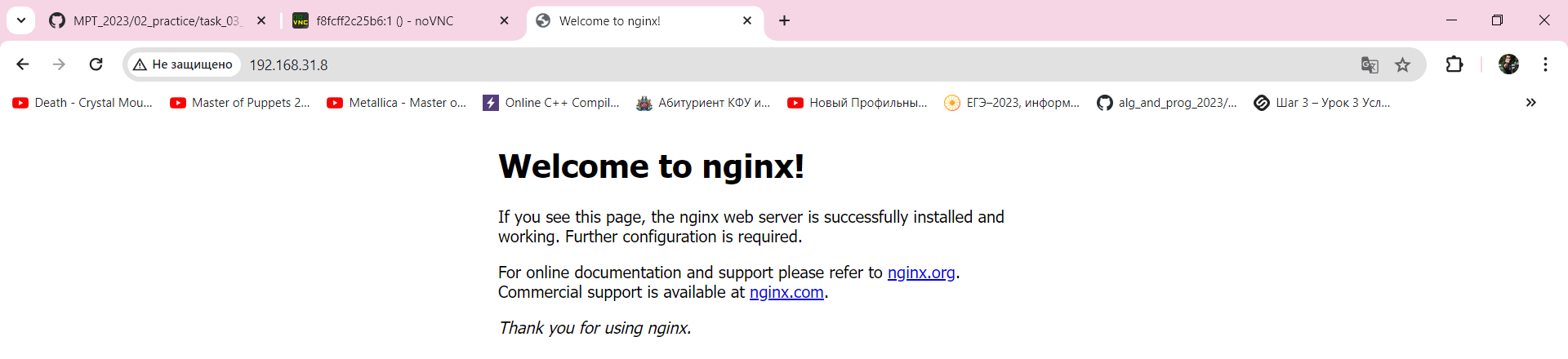


Нажму кнопку подключение и введу пароль.



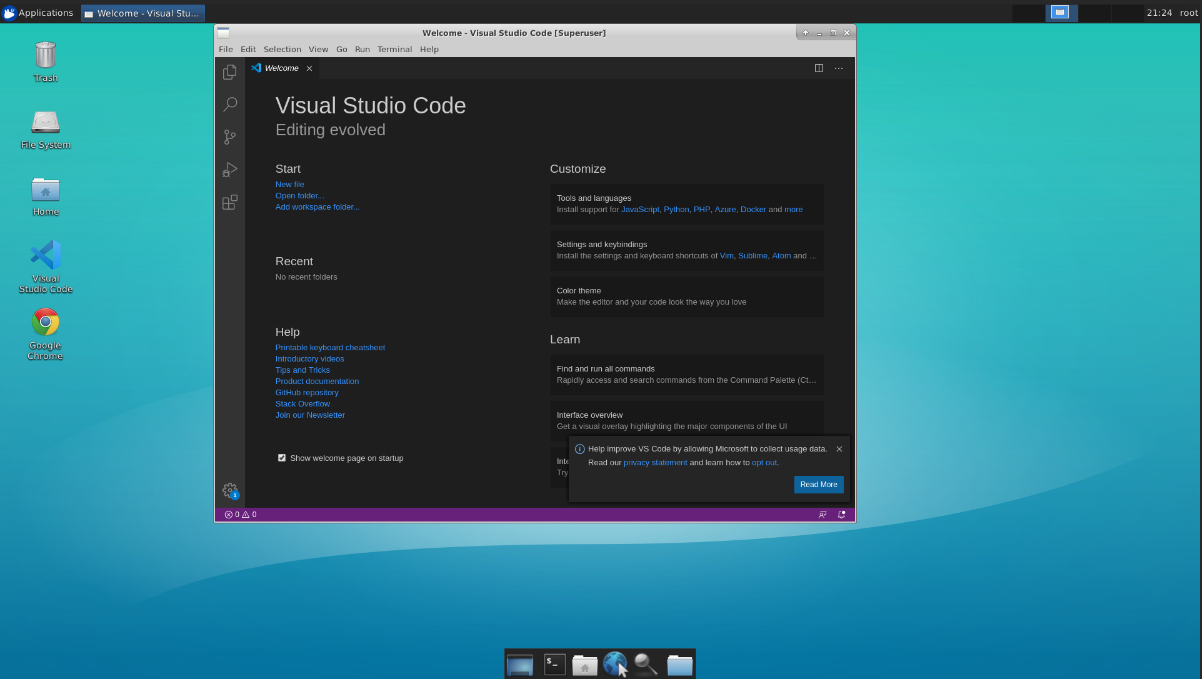
Открылся рабочий стол.

91. Обновим страницу с ip виртуалки в адресной строке:



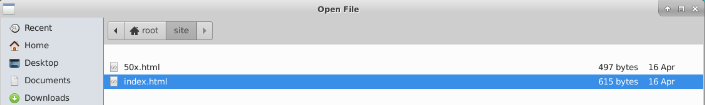
92. Выполним последовательность действий:

Запустим VScode:



В открывшемся окне нажму "File" -> "Open File...", а затем кнопку "Home" и в папке "site" выберу файл "index.html".

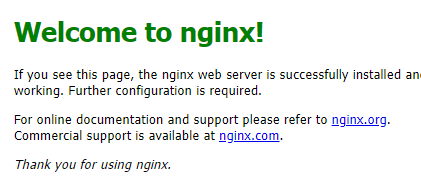
Этот файл появился в результате того, что сначала мы подключили том "data" к контейнеру с nginx и всё, что было у него в "/usr/share/nginx/html" попало в том;



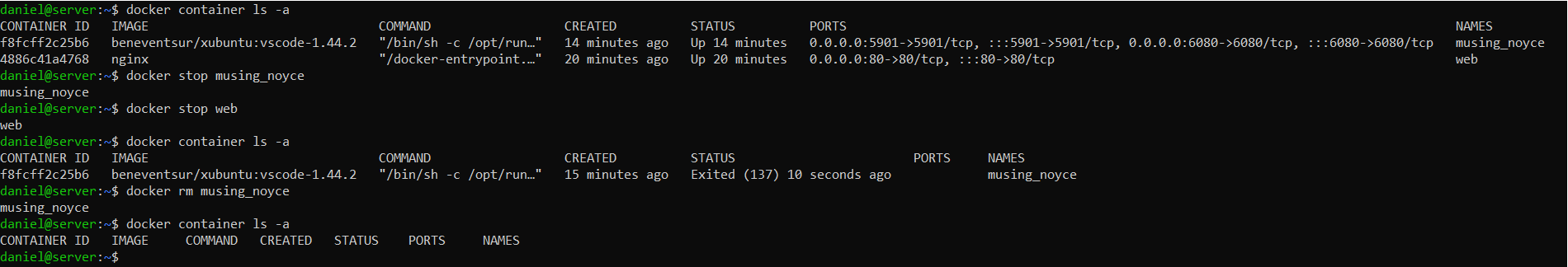
Изменим файл. Например, поменяем цвет приветственной надписи на зелёный:



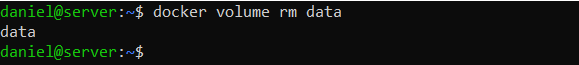
Перезагрузим ту страницу:



93. Закроем вкладки, остановим и удалим контейнеры.



94. Удалим том data:



95. Кроме опции -v можно, для монтирования, можно воспользоваться опцией --mount. В этом случае нужно заменить:

-v {локальный каталог или том}:{каталог в контейнере}

--mount type={bind-для каталогов или volume-для томов},source={локальный каталог или том},target={каталог в контейнере}

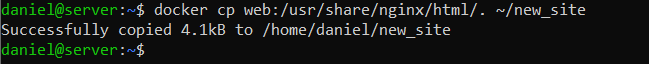
96. Монтирование tmpfs (похоже на том, только в оперативной памяти, а не на диске) в данной работе рассмотрено не будет, в виду её специфичности, и чтобы не усложнять работу.

**Прямое копирование файлов**

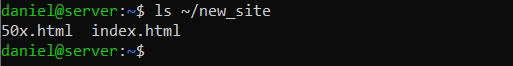
97. Запустим новый nginx контейнер с именем "web", в фоновом режиме, без монтирования томов и пробросив наружу 80-й порт.



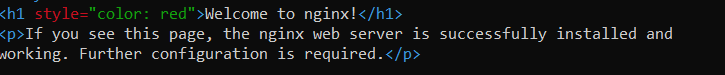
98. Введу команду:



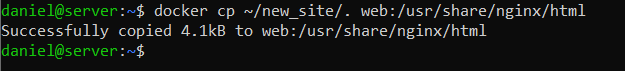
99. Проверим, что файлы появились в каталоге:



Изменю index.html, как обычно изменив цвет приветственной надписи:

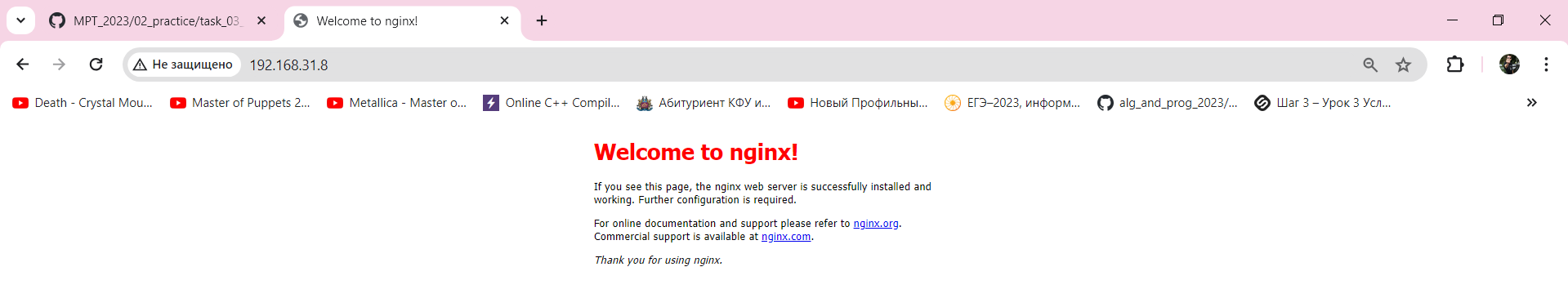


100. Введу команду:



Здесь всё ровно наоборот. Локальные файлы попадут в контейнер.

101. Откроем страницу в браузере:



Изменения сработали.

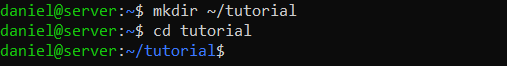
Остановим и удалим контейнер:



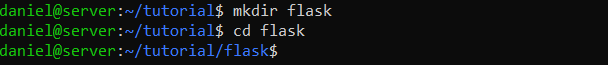
**Сборка своего образа**

**Настройка базового Flask приложения**

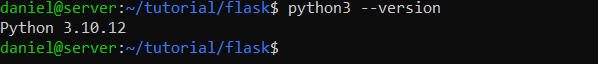
102. В домашнем каталоге создадим каталог tutorial и зайдём в него:



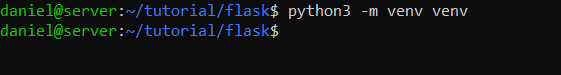
103. Здесь создадим каталог flask и зайдём в него. Тут будет flask-приложение:



104. Проверим, что в системе установлен интерпретатор python3 и его версию:



105. Создадим виртуальное окружение с именем "venv":

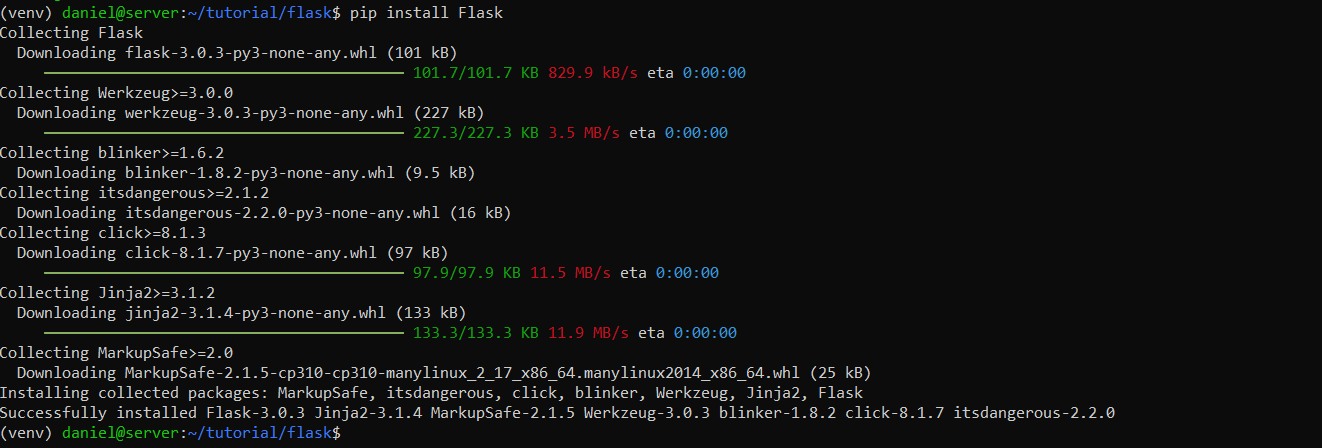


Мне пришлось сначала установить venv.

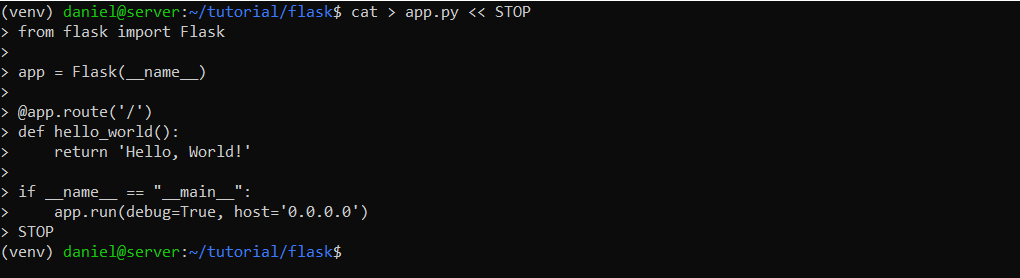
106. Активируем его:



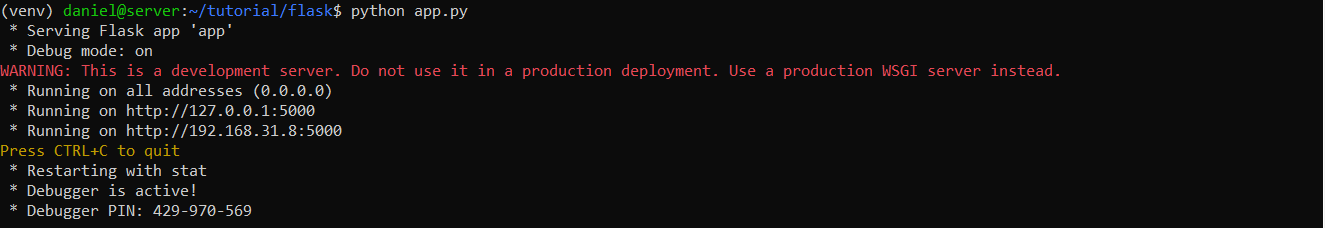
107. Установим пакет Flask с помощью pip:



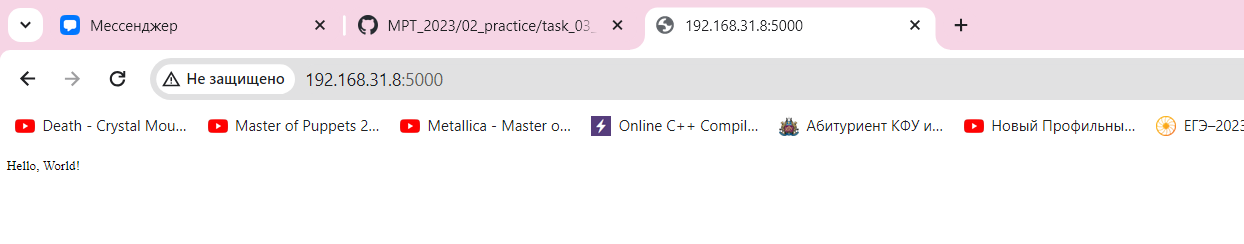
108. Создадим файл с именем app.py содержащий код из документации Flask для создания базового приложения:



109. Запустим этот файл:



Перейдём на страницу в браузере и откроем страницу с ip виртуалки:



110. Остановим приложение Ctrl+C:

111. Сохраним установленные пакеты (зависимости) в файл requirements.txt:



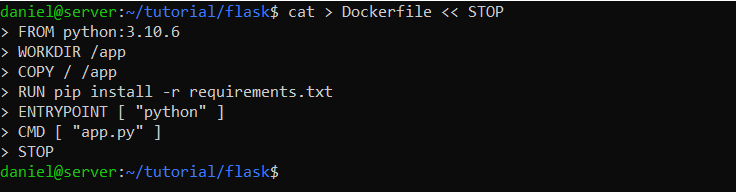
112. Деактивирую виртуальную среду:



**Упаковка приложения в docker-контейнер**

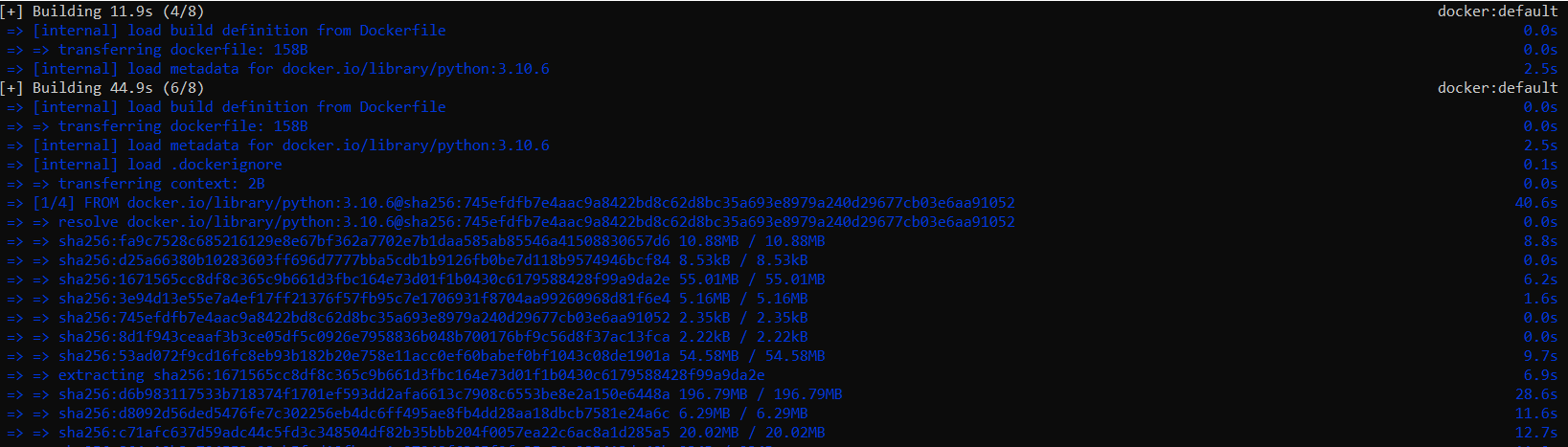
**Создаём Dockerfile**

113. Создадим файл с именем Dockerfile (nano Dockerfile):

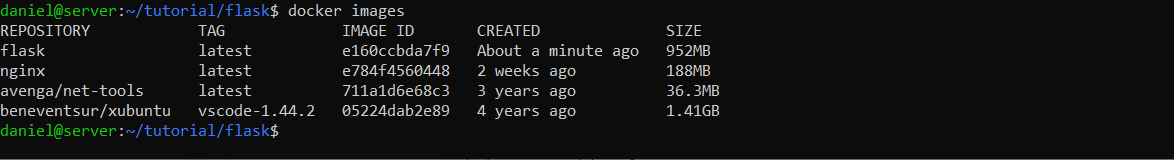


**Создаём образ**

114. Запустим сборку образа(команда docker build –t flask):

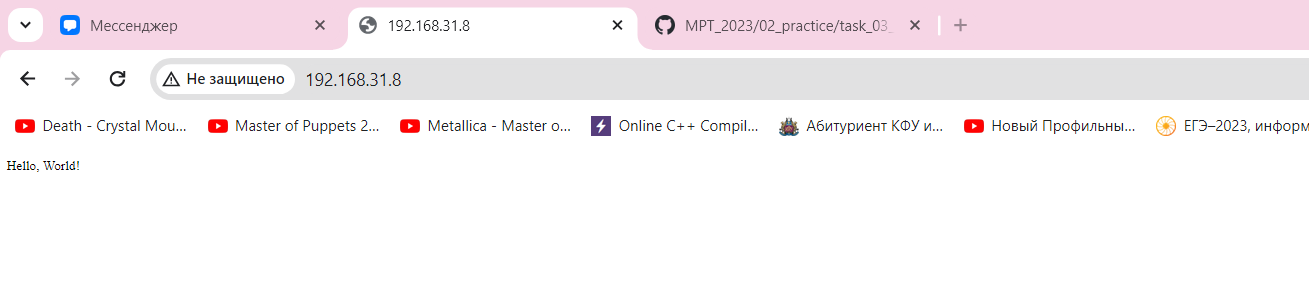


115. Посмотрим список образов:



flask присутствует.

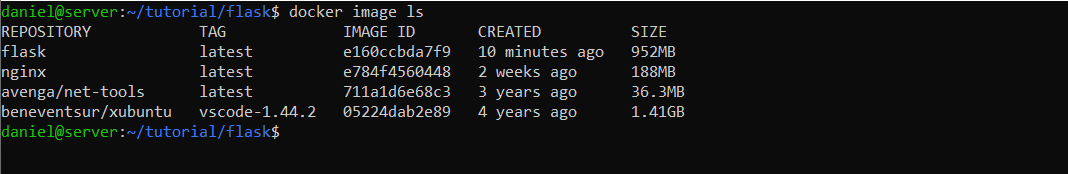
116. Запустим и проверим работу, после чего остановим:





**Уменьшаем размер образа**

117. Посмотрим размер образа:



952МБ, ОФИГЕТЬ!

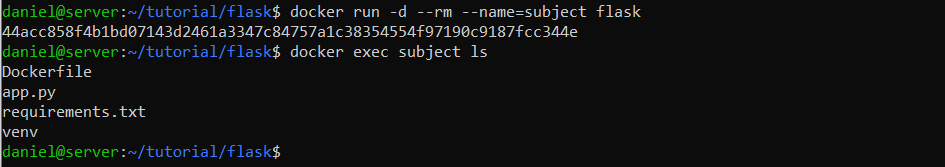
118. У нас есть 2 основных способа уменьшения веса образа:

-Не копировать в образ лишних файлов;

-Использовать более легковесный базовый образ.

119. Для начала избавимся от лишних файлов.

120. Запустим новый контейнер, а затем посмотрим, что у него внутри:



121. Оценим сколько места занимают эти:



122. Чтобы эти мусорные файлы не попали к образ при сборке можно поступить двумя способами:

-Во-первых, мы можем изменить Dockerfile и скопировать не все файлы из директории flask, а только те, которые нужны, т.е:

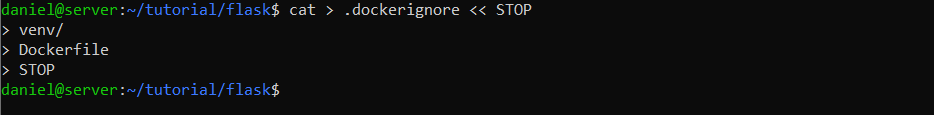
-COPY requirements.txt /app

-COPY app.py /app

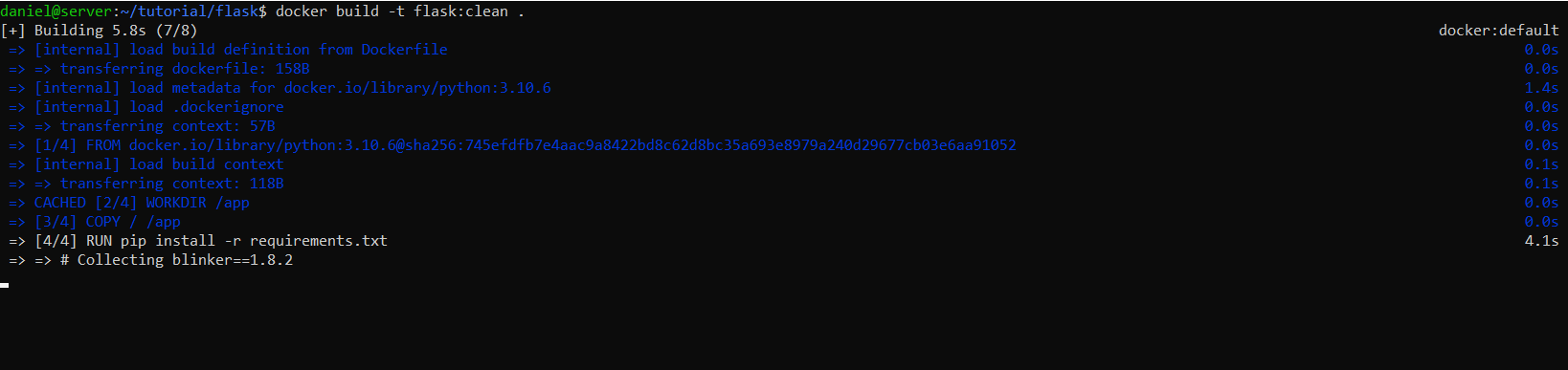
У нас очень маленькое приложение, и оно простое, но для больших проектов это может занять довольно много строк.

-2-й вариант: Docker предоставляет нам возможность добавить файл ".dockerignore", который работает как файл ".gitignore" для git. Когда Docker запустит COPY / /app при создании образа он проигнорирует файлы и каталоги перечисленные в ".dockerignore" и сам ".dockerignore" тоже.

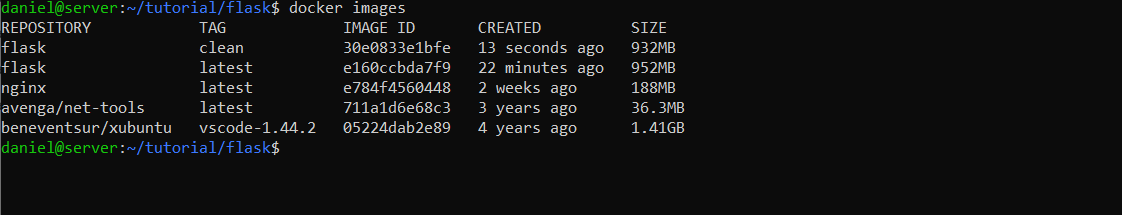
123. Создадим в каталоге flask файл ".dockerignore":



124. Пересоберём образ командой, но назовём его flask:clean:



125. Посмотрим список образов:



Теперь есть два образа flask с разными тегами.

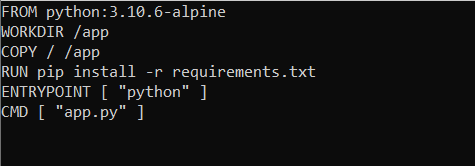
Сравним их размеры:

932 < 952

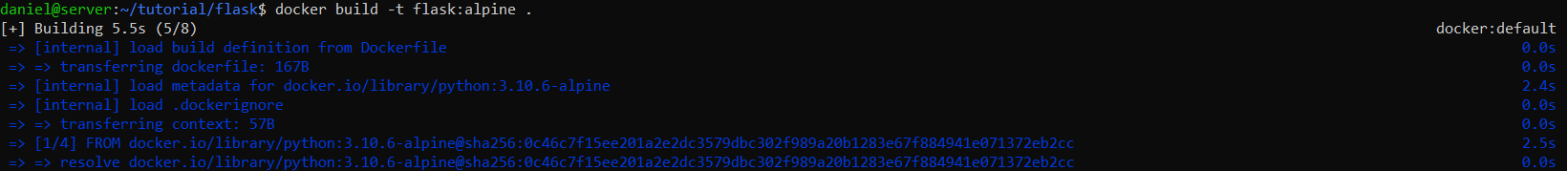
126. Теперь когда "мусора" в образе нет подумаем о замене базового образа для контейнера.

127. Откроем Dockerfile и замените первую строку:





Пересоберём образ с новым названием:



128. Посмотрим список образов:

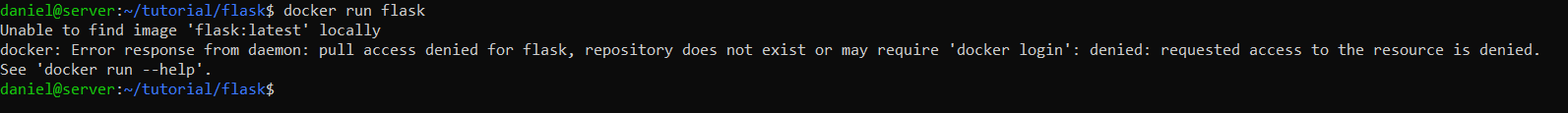


Размер нового образа – 60.4MB

129. Удалим образы flask c тегами latest, clean:



130. Теперь пытаясь запустить контейнер с образа flask, мы получим ошибку:



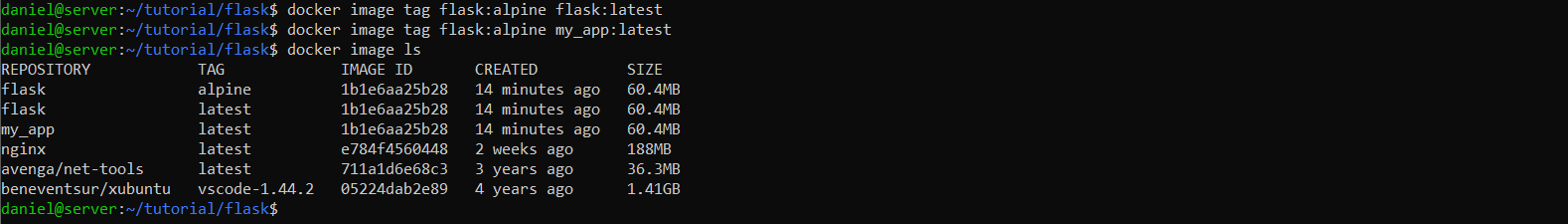
131. Выполним команды:

-docker image tag flask:alpine flask:latest

-docker image tag flask:alpine my\_app:latest

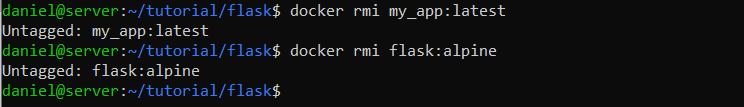
-docker image ls

и обратим внимание на IMAGE ID:



мы добавили 2 других имени для образа с нашим приложением.

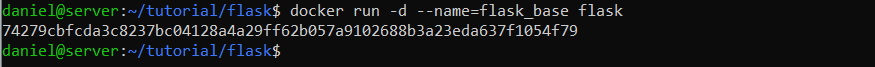
132. Удалим my\_app:latest и flask:alpine командой docker rmi:



Удалились только имена, а сам образ остался под именем flask:latest.

**Создание образа из контейнера**

133. Запустим новый контейнер в фоновом режиме:



134. Установим в этом контейнере модуль для работы с mysql - "mysql-connector-python":



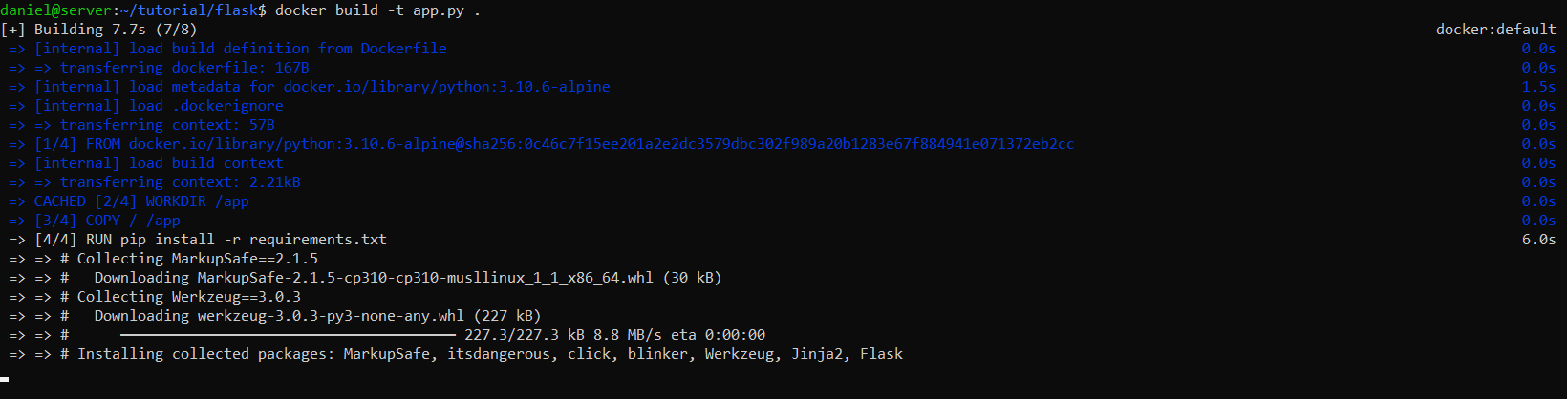
135. Откроем локальный файл "app.py" и замените его содержимое:



136. Остановим контейнер, а затем скопируем локальный файл app.py в контейнер:



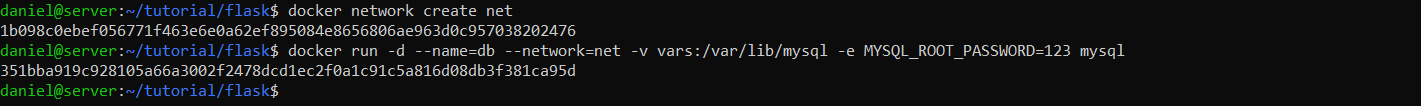
137. Мы изменили в контейнере всё, что хотели, теперь из него можно сделать образ (ну или просто запустить (docker start))



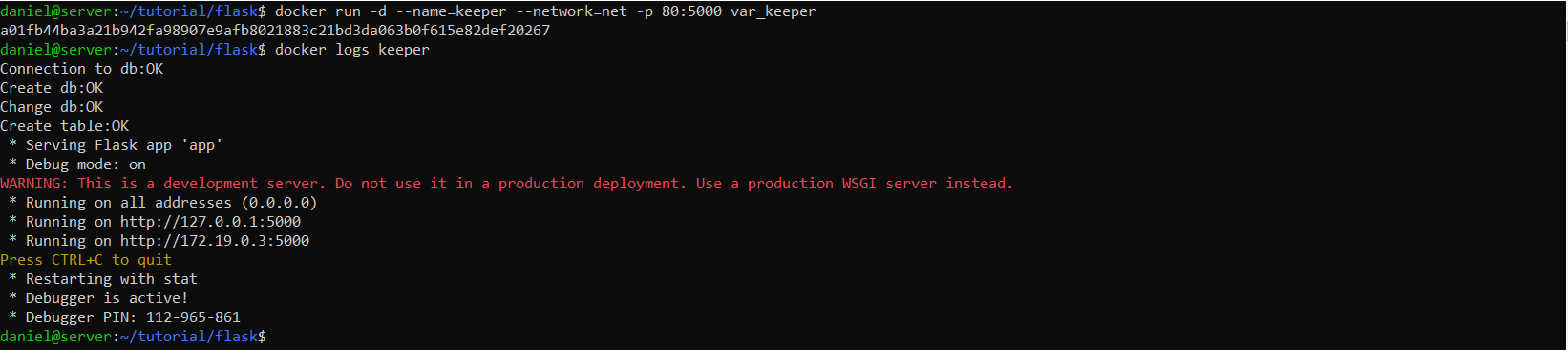
138. Проверим его наличие в списке:

139. Теперь проверим работоспособность получившегося образа, но для этого нам понадобиться контейнер с базой данный mysql.

140. Создадим сеть под названием "net", а затем запустим контейнер с mysql как указано на странице образа:



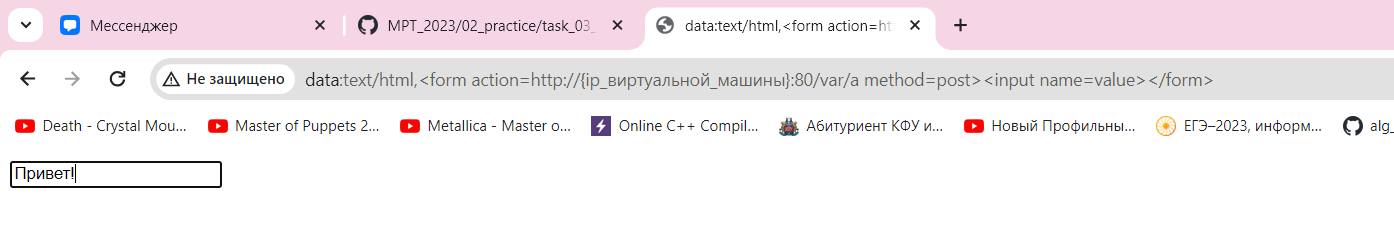
141. Теперь запустим контейнер из образа var\_keeper:latest и проверим, правильно ли всё работает:



142. Проверьте работоспособность контейнера при помощи браузера. Для отправки POST запроса можно ввести в строку адреса:

data:text/html,<form action=http://{ip\_виртуальной\_машины}:80/var/a method=post><input name=value></form>

и нажать Enter. На экране появится поле ввода. Введённое значение будет отправлено на /var/a и сервер запомнит новую переменную a:

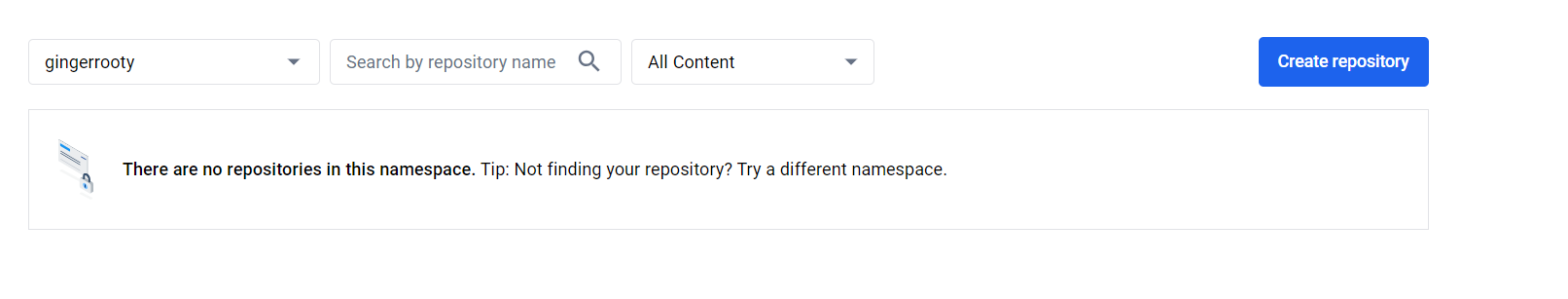


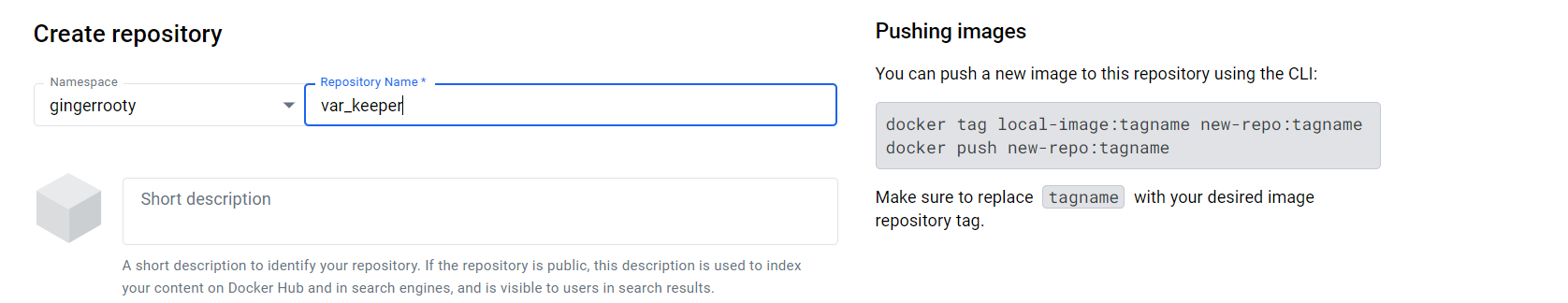
143. Затем отправим GET-запрос(по http://{ip\_виртуальной\_машины}/var/a) в браузере должно появиться то значение)

**Отправляем образ на DockerHub**

144. Зарегистрируемся на https://hub.docker.com и зайдём под своей учёткой.

145. Создадим репозиторий:

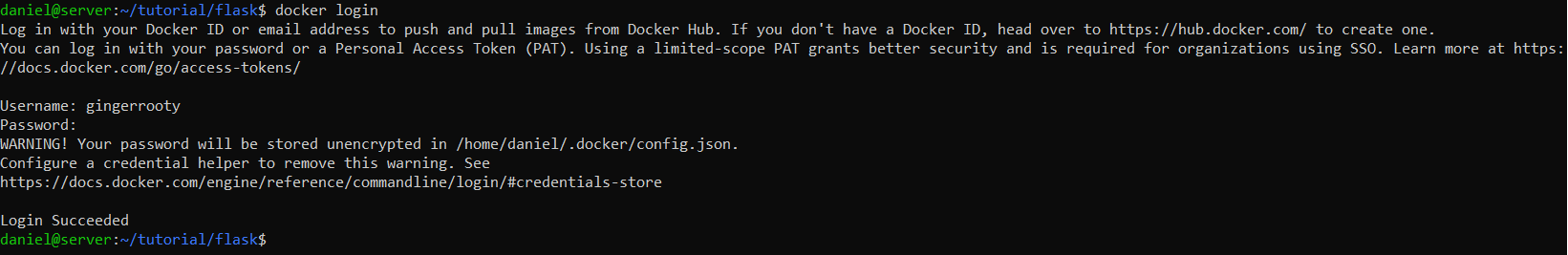




146. Перейду в терминал и введу команду:

-docker login

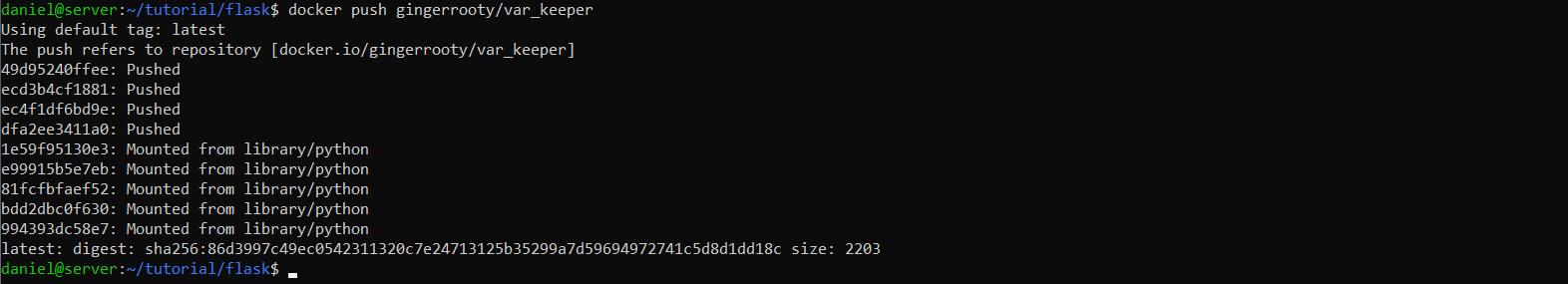
Введу имя пользователя и пароль, от DockerHub. Для того VPN пока не требуют.



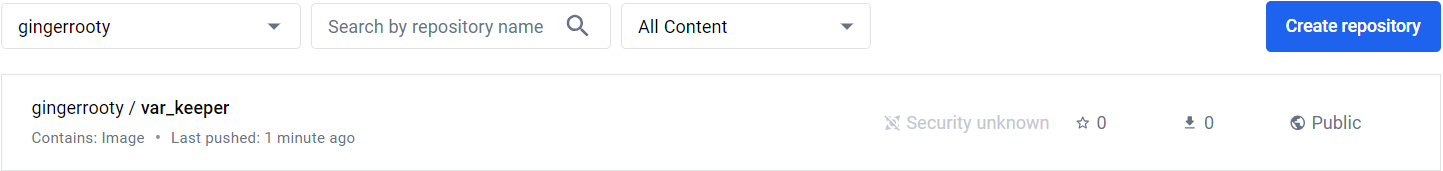
147. Добавлю образу var\_keeper ещё одно имя <имя пользователя DockerHub>/var\_keeper:



148. Отправлю образ на DockerHub:



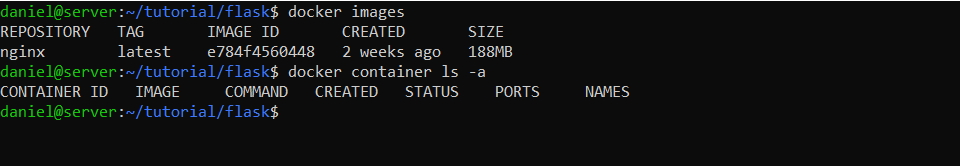
Полное название загруженного на DockerHub образа:



# Docker - многоконтейнерные приложения

# Начинаем работать с Docker compose

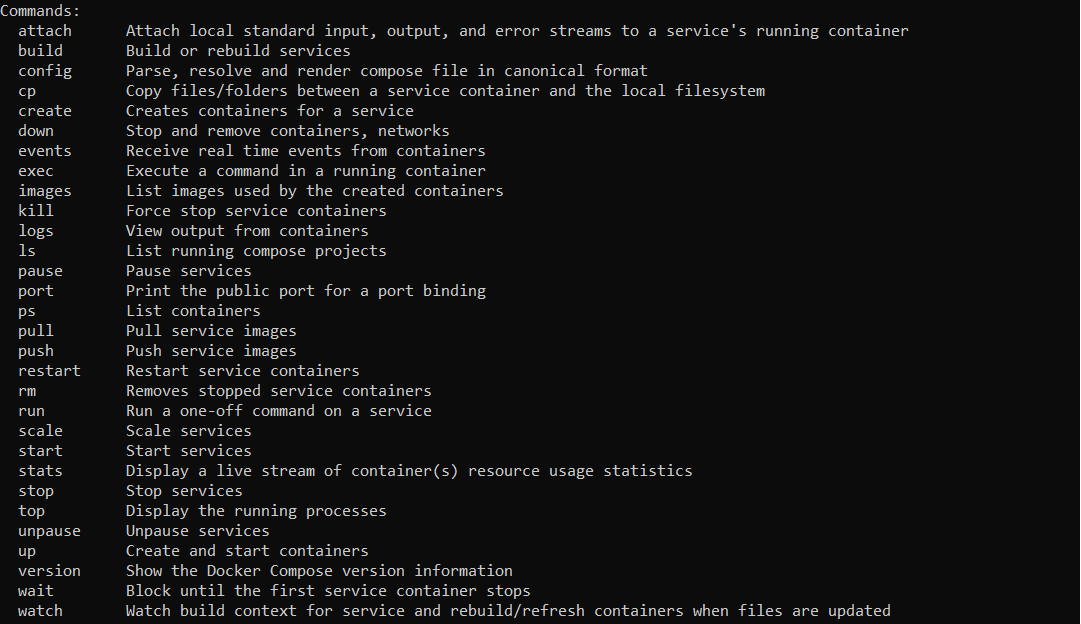
1. Для начала удалим все образы (кроме nginx) и все контейнеры которые у меня остались с прошлой работы.



Всё чистенько!

2. Выполним команду:

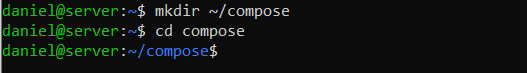




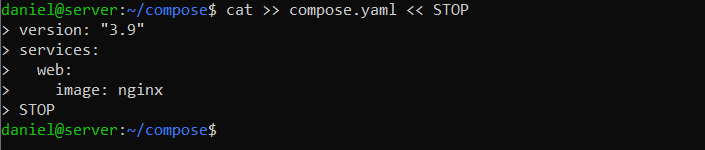
3. Изучим структуру формата yaml.



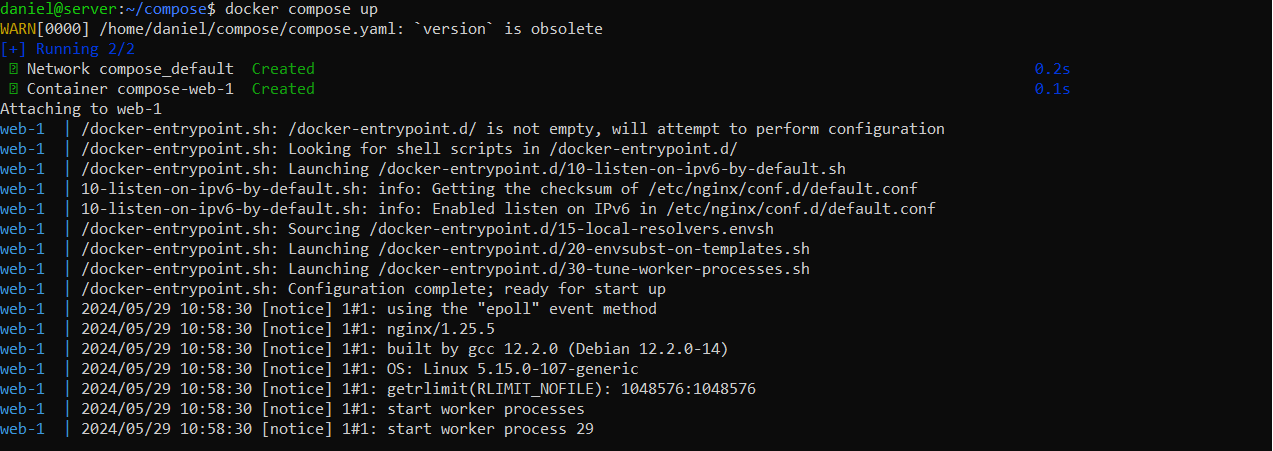
4. Создадим в ~ папку и зайдём в неё:



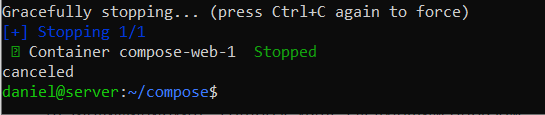
5. Создадим файл "compose.yaml", содержащий:

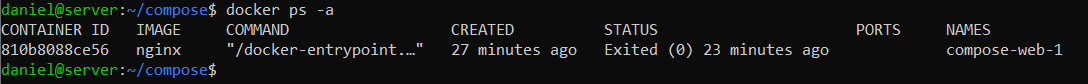


6. Запустим приложение файлом:

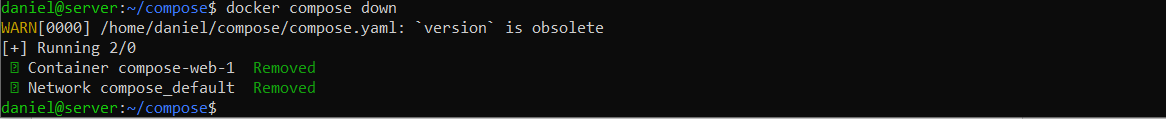


7. В данном случае мы запустили сервис "web" в attached mode, поэтому он захватил ввод и вывод нашего терминала. Остановим контейнер Ctrl+C и посмотрим список всех контейнеров docker ps –a.

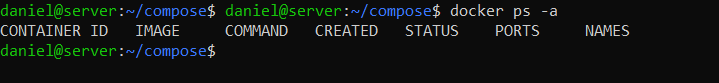




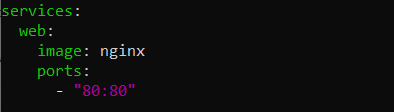
8. Выполним команду:



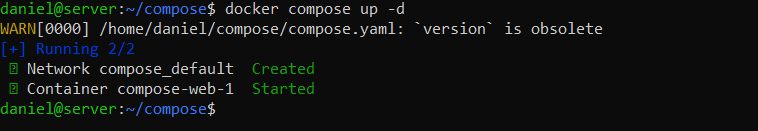
9. Проверим список контейнеров:



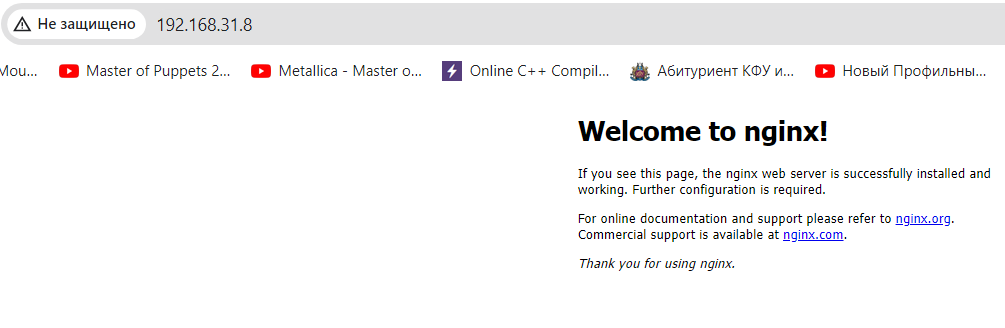
10. Модифицируем "compose.yaml":



и запустим приложение:



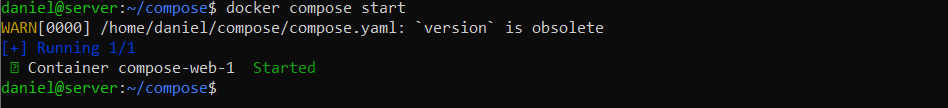
11. Перейду в браузер в основной ОС и введу в строку адреса ip виртуальной машины:



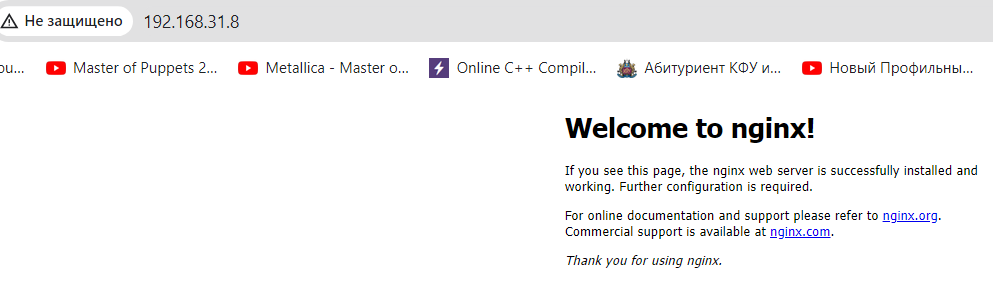
12. Остановим приложение:



13. Запустим приложение:



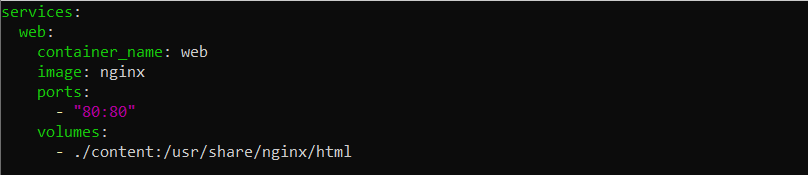
Эта команда выполнит start для всех контейнеров приложения. Через браузер по прежнему можно получить ответ от nginx.



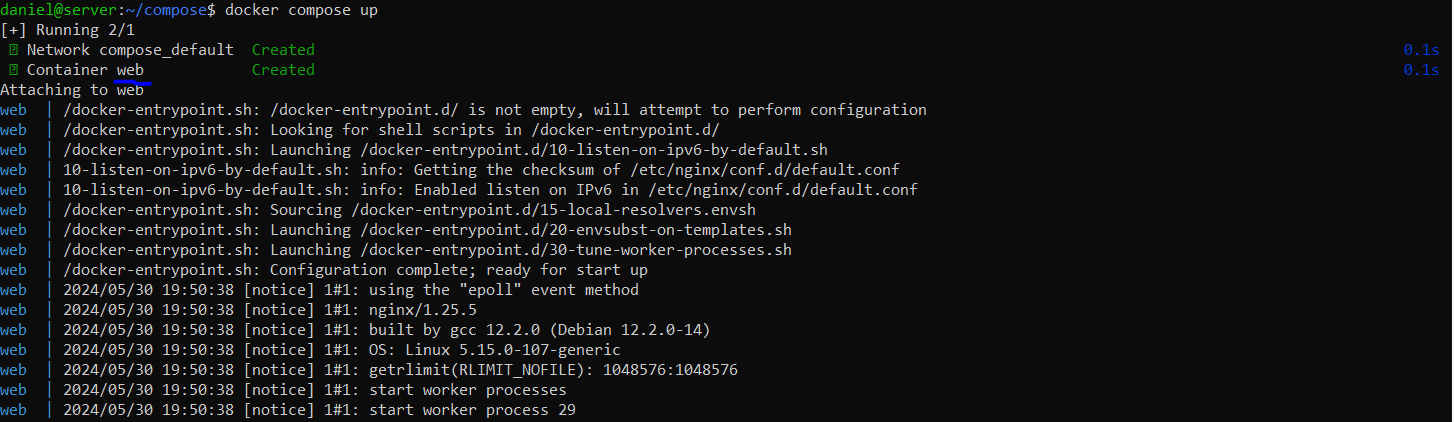
14. Остановим приложение:



15. Модифицируем "compose.yaml":



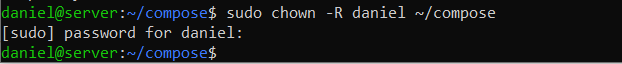
16. Запустим приложение:



Видим, что его название – web. А ещё в текущем каталоге появился каталог content:



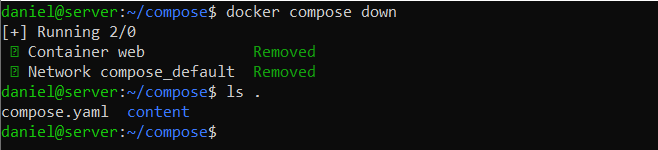
17. Поменяем владельца каталога compose других в нём на себя на себя:



Создадим в content index.html файл:

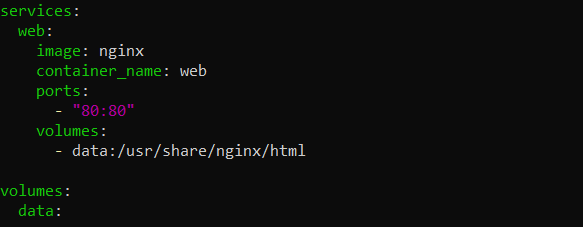


18. Остановим контейнер:

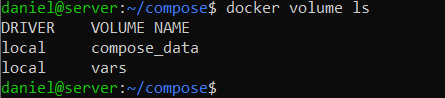


Каталог content всё ещё есть.

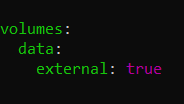
19. Модифицируем "compose.yaml":



20. Запустим приложение и посмотрим список томов:

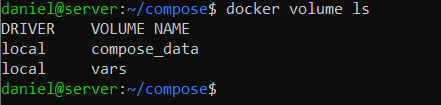


Чтобы намеренно использовать существующий том, нужно создавать его таким образом:

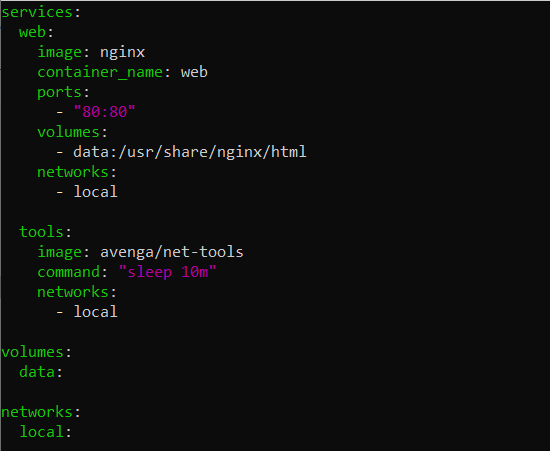


21. Остановим приложение и убедимся, что том по-прежнему существует:

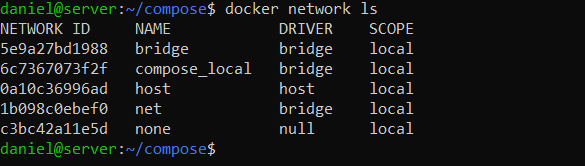




22. Модифицируем "compose.yaml":



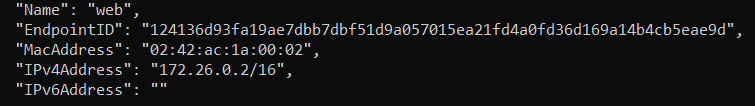
23. Запустим приложение и посмотрите список сетей:



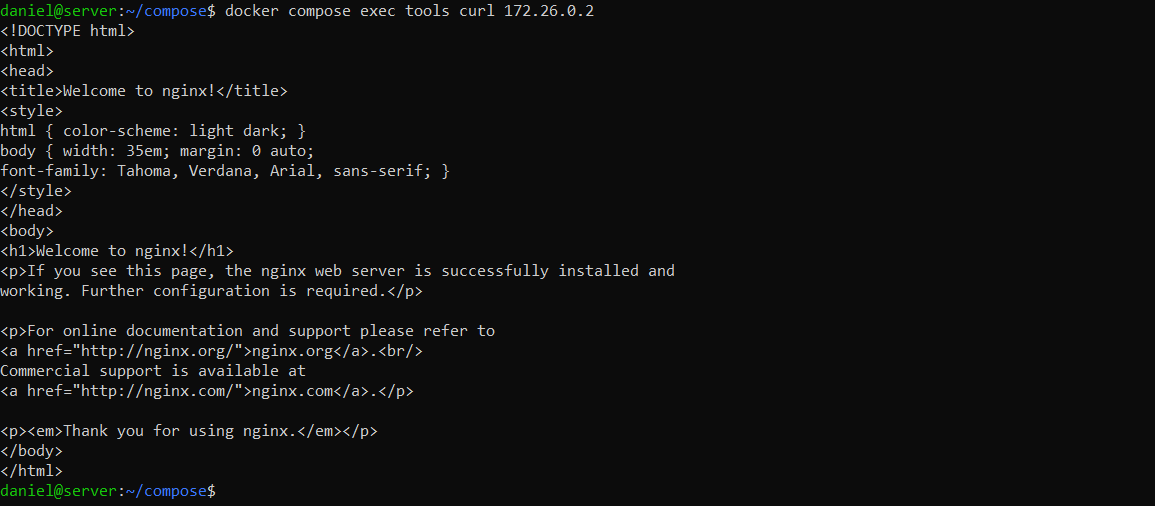
24. Выполним команду:



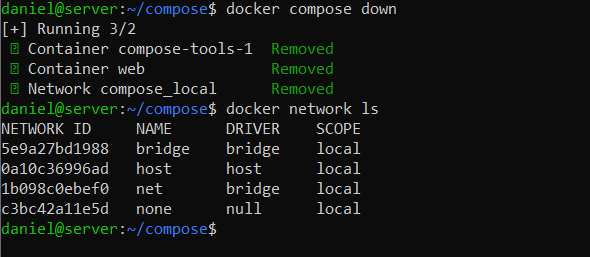
Найдём ip контейнера:



25. Выполним команду:



26. Остановим приложение и проверим, что сети "compose\_local" больше нет в списке.

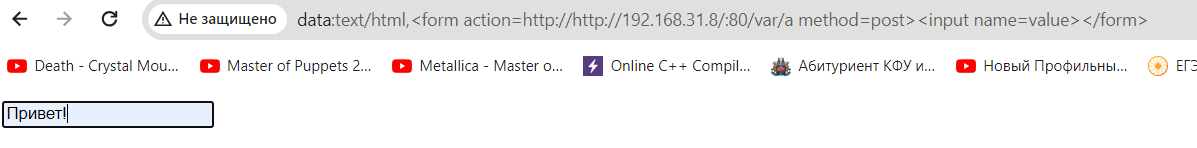


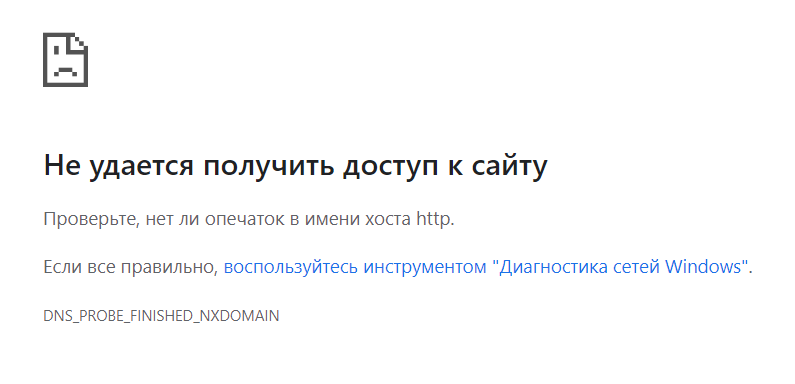
27. В прошлой работе я создавал приложение "var\_keeper" и загружал его на свой аккаунт в DockerHub. Используем этот образ и развернём приложение на локальной машине при помощи compose.

28. Модифицирую "compose.yaml":

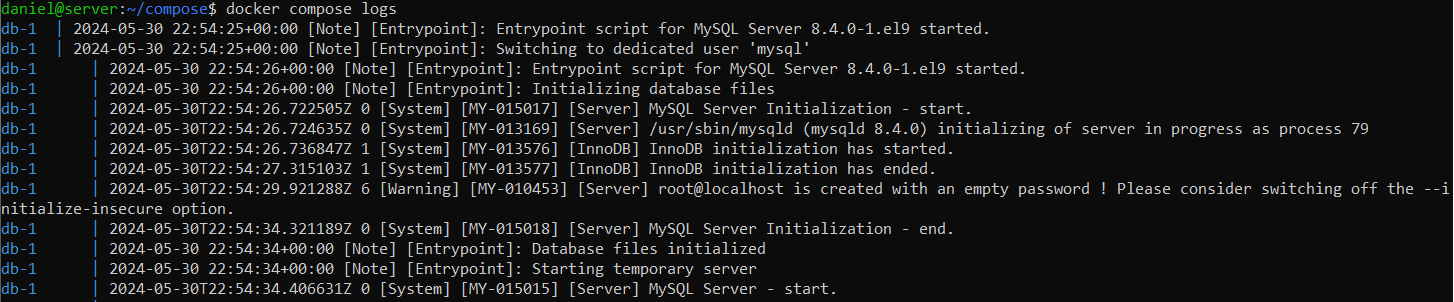


29. Запустим приложение и проверим его работоспособность при помощи браузера:

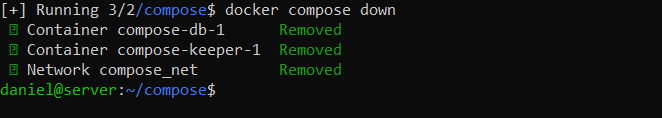


 Ошибка

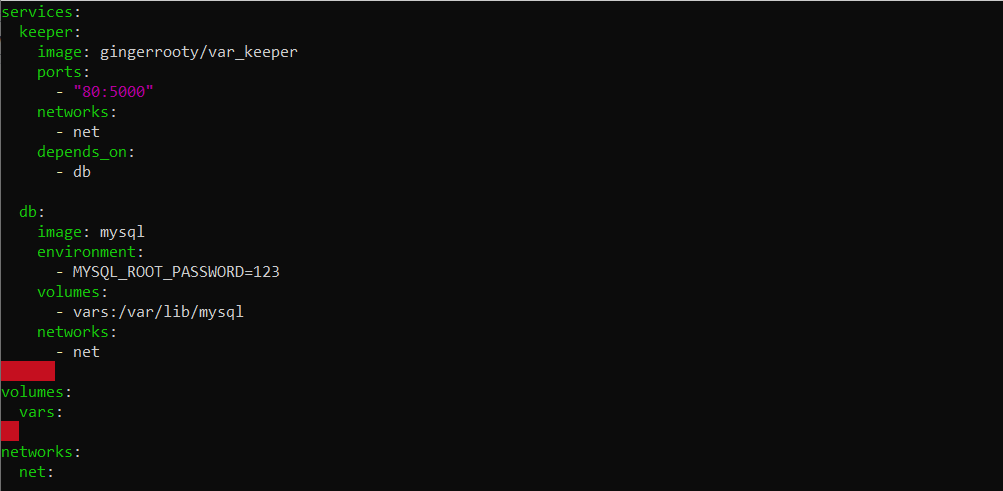
30. Посмотрим compose logs:



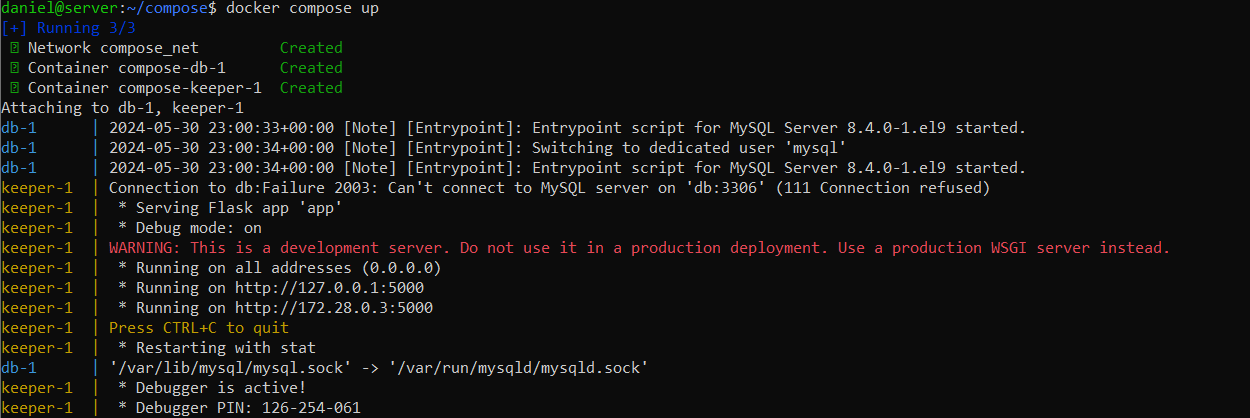
31. Остановим приложение:

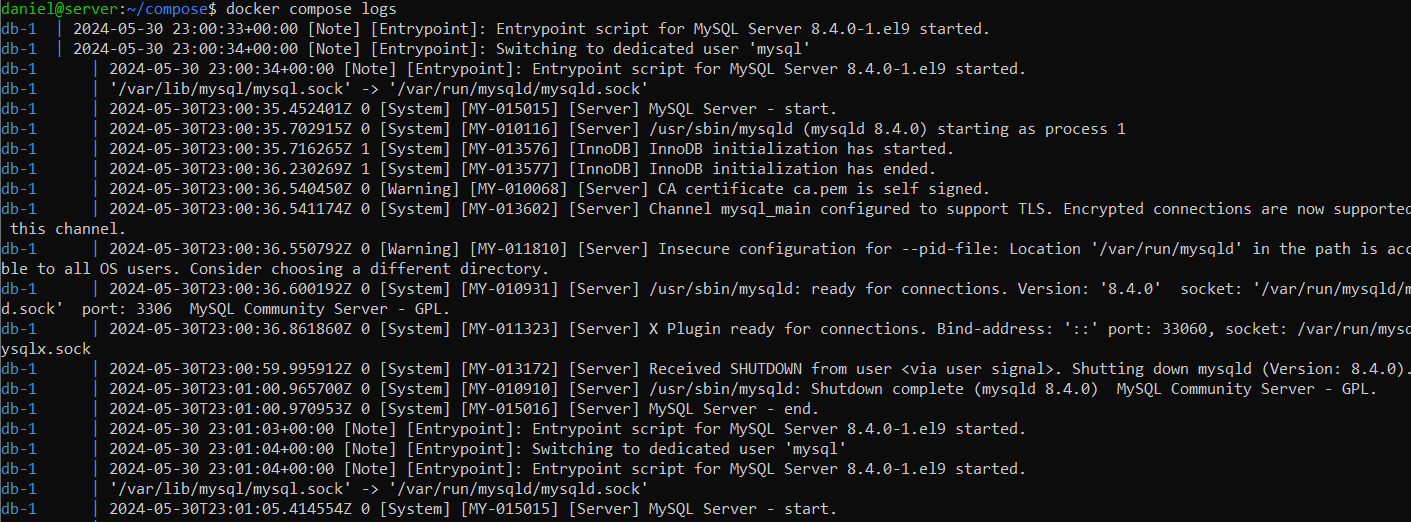


32. Модифицируем "compose.yaml":



33. Запустим приложение и сразу проверим логи:





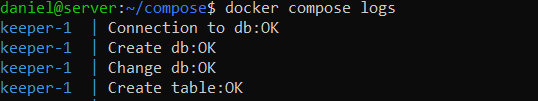
34. Приложение по-прежнему не работает - остановим его:

35. Чтобы решить нашу задачу compose предоставляет возможность проверять "здоровье" (healthy) контейнера при помощи ключа healthcheck.

Модифицируем "compose.yaml":



36. Запустим приложение и затем проверим логи:



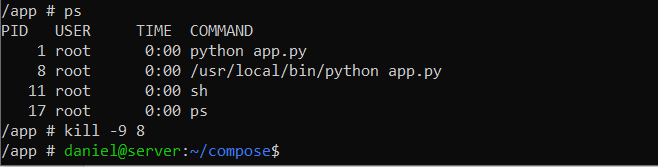
37. Если в процессе работы приложения один или несколько контейнеров остановятся, то приложение перестанет работать нормально. Чтобы поддерживать его работоспособность можно назначить политику перезапуска при помощи ключа restart.

38. Модифицируем файл compose.yaml:

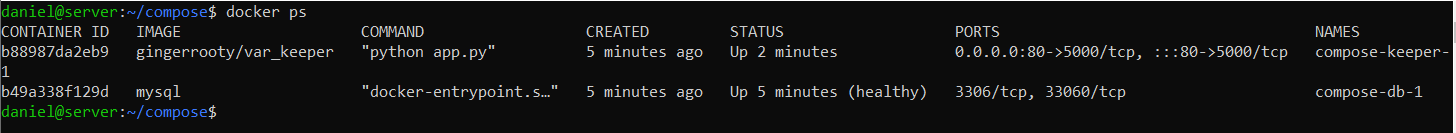


39. Запустим приложение и зайдём в контейнер сервиса "keeper":





40. Проверим список запущенных контейнеров, и мы должны обнаружить, что "keeper" по-прежнему на месте, но в столбце STATUS (я 2 минуты ничего не делал):

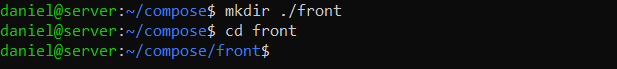


41. Остановим приложение:

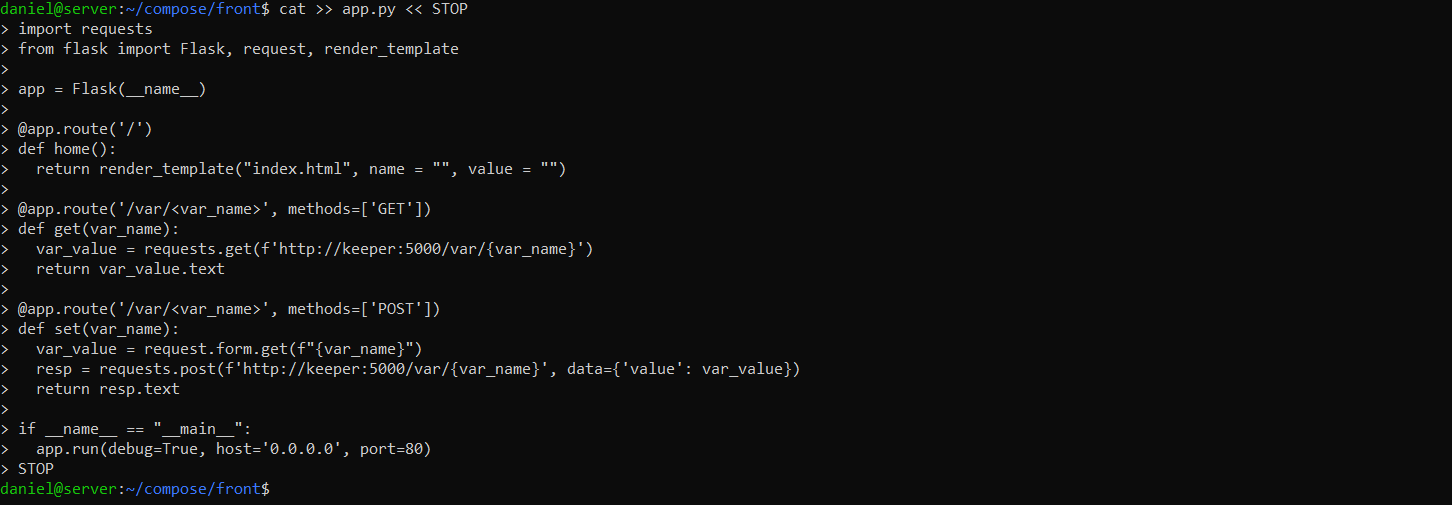


42. Сейчас для сохранения и получения значений переменных нужно напрямую посылать приложению запросы, что не очень удобно. Поэтому добавим ему простенький фронтенд. Сборку контейнера с фронтендом поручим compose, но для этого ему нужен Docerfile и код фронтенда.

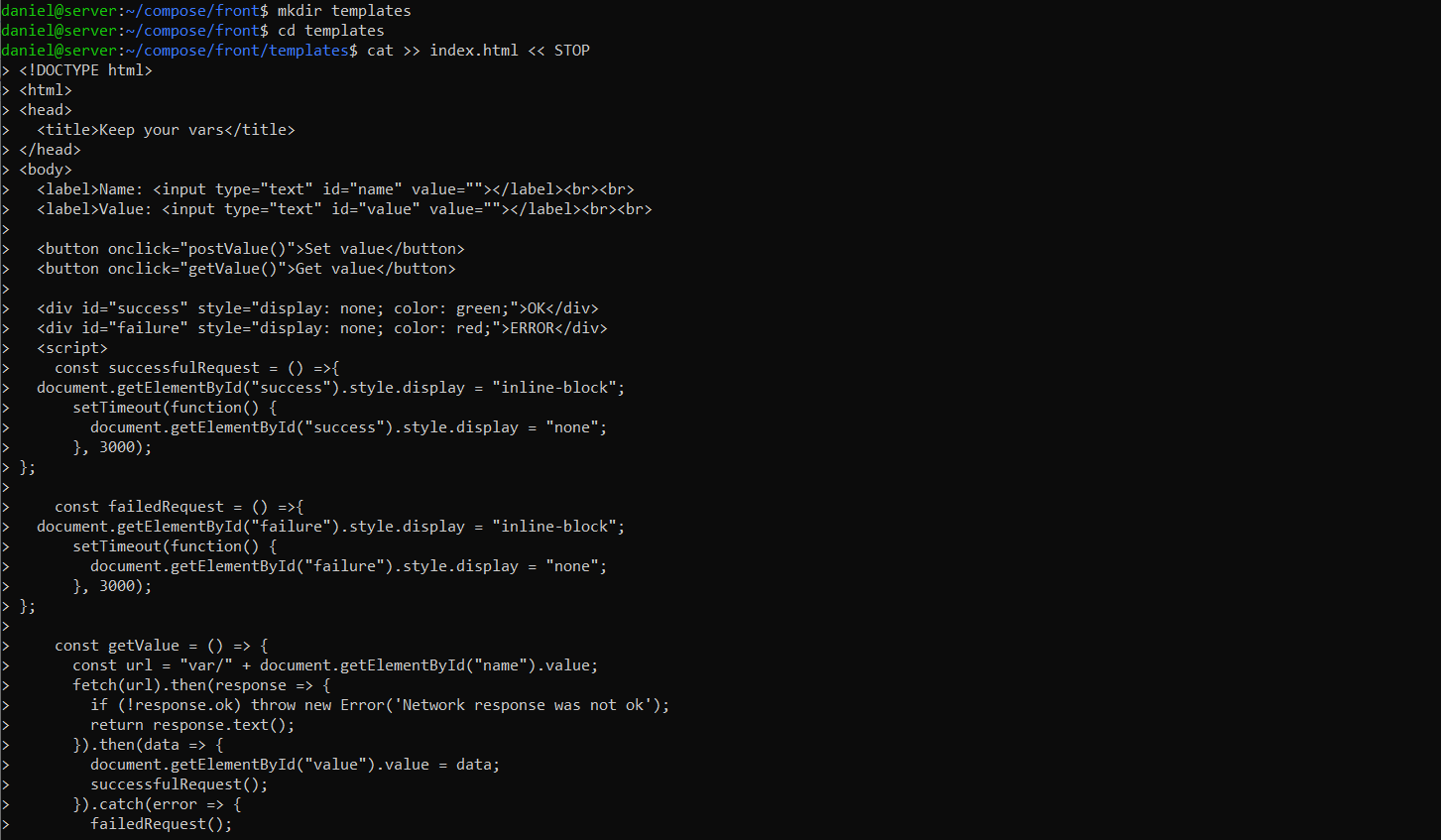
Создадим в директории "compose" ещё одну с именем "front" и перейдём туда:

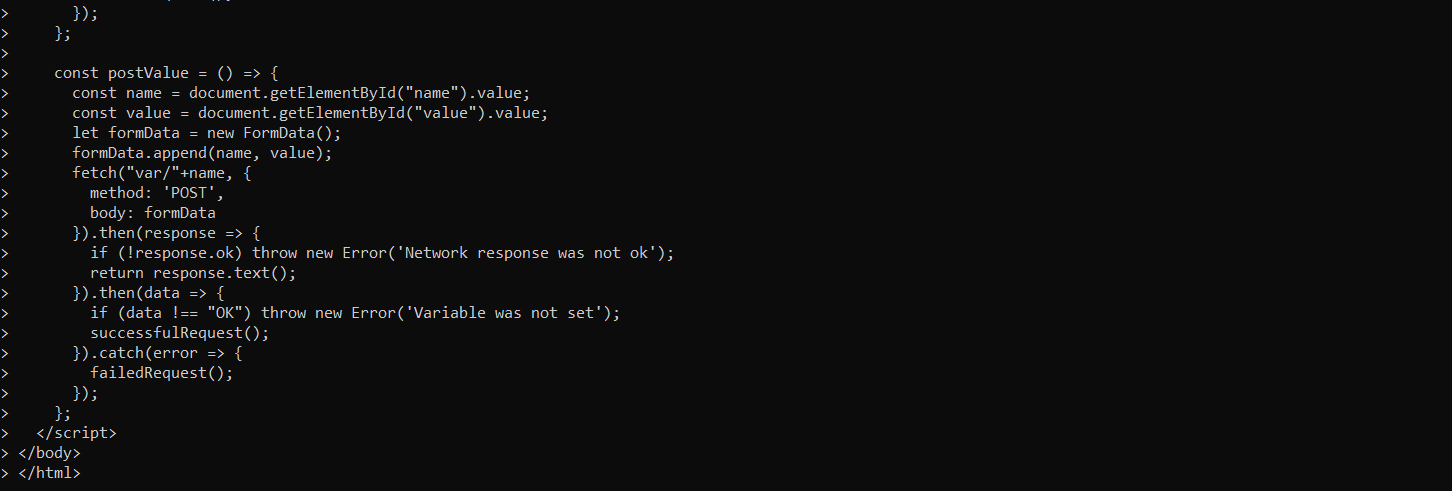


Создадим файл "app.py" содержащий:

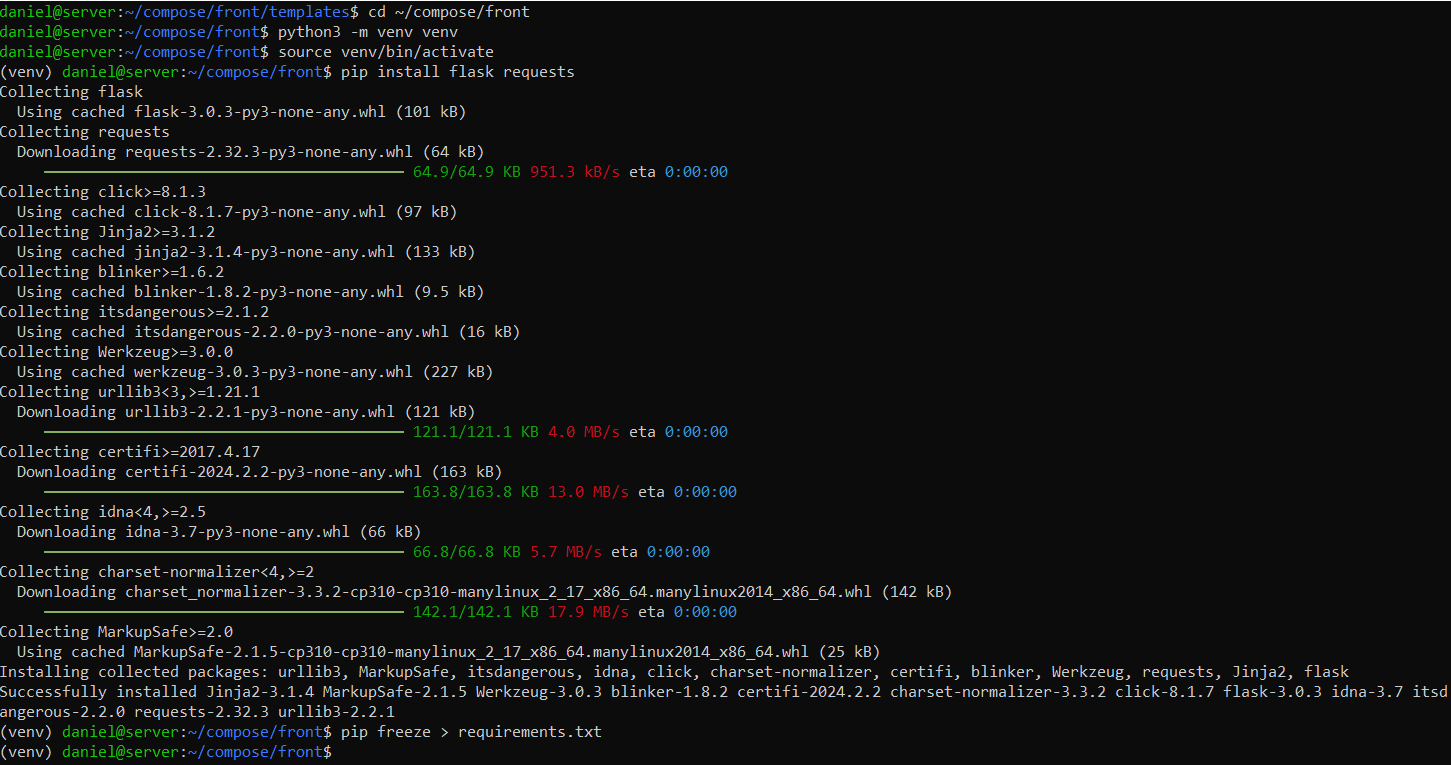


Создадим каталог "templates" и в нём файл "index.html" содержащий:

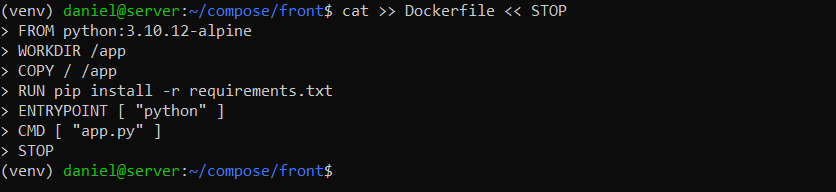




Вернёмся в каталог "front" и чтобы получить список зависимостей выполним команды:



Создадим Dockerfile:



43. Теперь у нас есть готовый фронтенд и Dockerfile, чтобы упаковать его в контейнер.

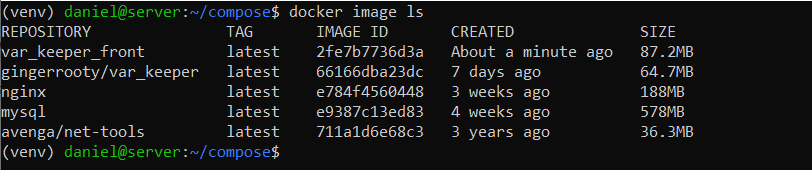
Выйдем из каталога "front" и модифицируем "compose.yaml" следующим образом:



44. Выполним команду:

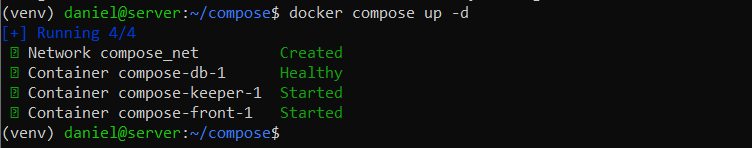


45. Проверим список образов:

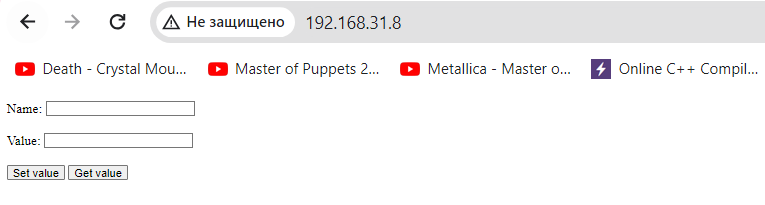


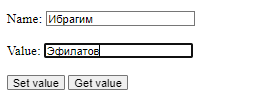
Теперь здесь есть образ var\_keeper\_front с тегом latest.

46. Выполним команду:



47. Перейдём в браузер и введём в строку адреса ip виртуальной машины:





**Ответы на вопросы.**

1. Как запустить контейнер в интерактивном режиме

- Docker: `docker run -it <имя\_образа> <команда>`

- Podman: `podman run -it <имя\_образа> <команда>`

2. Что нужно сделать, чтобы при вводе команд для работы с контейнерами не нужно было вводить sudo?

- Docker: Добавить пользователя в группу `docker`: `sudo usermod -aG docker <имя\_пользователя>`

- Podman: Не требуется использование sudo для выполнения команд, так как Podman не требует привилегированных прав.

3. Каким образом можно соединить 2 контейнера в отдельную для них сеть?

- Docker: `docker network create <имя\_сети>` для создания сети, затем `docker run --network=<имя\_сети> <имя\_образа>`

- Podman: `podman network create <имя\_сети>` для создания сети, затем `podman run --network=<имя\_сети> <имя\_образа>`

4. Два контейнера хотят обмениваться файлами. Каким образом можно организовать такое взаимодействие?

- Docker и Podman: Можно использовать `docker cp` или `podman cp` для копирования файлов между контейнерами и хостом.

5. Мне нужен образ с wordpress на основе дистрибутива alpine. Приведите команду которой я могу скачать такой образ с dockerhub (версию wordpress и php выберите любую).

- DСоздайте файл с именем Dockerfile (nano Dockerfile) и скопируйте в него приведенный ниже код:ocker: `docker pull wordpress:php7.4-fpm-alpine`

- Podman: `podman pull wordpress:php7.4-fpm-alpine`

6. Как указать имя контейнера (не образа) в docker-compose.yml файле?

- Docker: В файле `docker-compose.yml` для каждого сервиса используйте параметр `container\_name`.

- Podman: В файле `docker-compose.yml` для каждого сервиса используйте параметр `name`.

7. Как директива healthcheck влияет на работу контейнера?

- Docker и Podman: Директива `healthcheck` позволяет определить команду для проверки здоровья контейнера. Она используется для автоматического определения работоспособности контейнера и его автоматического перезапуска при необходимости.

8. Как указать статический IP адрес для контейнера в docker-compose.yml?

- Docker и Podman: В файле `docker-compose.yml` используйте параметр `ipv4\_address` для назначения статического IP-адреса контейнеру.

9. Как depends\_on влияет на порядок запуска контейнеров?

- Docker и Podman: `depends\_on` определяет порядок запуска контейнеров. Контейнеры будут запущены в порядке, указанном в `depends\_on`.

10. В чем разница между up, run и start?

- `docker run` используется для запуска нового контейнера из образа.

- `docker start` используется для запуска остановленного контейнера.

- `docker-compose up` используется для создания и запуска всех контейнеров из файла `docker-compose.yml`.