Тема: **Установка Docker, сборка образа для развертывания веб-сервера на Flask в ОС «Astra Linux Smolensk 1.7»**

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

Санкт-Петербург

2024

**Цель работы:** установить Docker в ОС СН Astra Linux Special Edition 1.7. Собрать образ для развертывания веб-сервера с использованием фреймворка Flask

**Материально-техническое обеспечение:** ОС Astra Linux Special Edition 1.7 (Версия Смоленск), версия ядра 6.1.50-1-generic

**ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Контейнеризация — это форма виртуализации, при которой приложения запускаются на хостовой ОС в изолированных пользовательских пространствах, называемых контейнерами. Контейнер представляет собой среду выполнения для приложения, которая содержит все необходимое для его работы.

Рассмотрим программные средства изоляции контейнеров в Linux-системах на материале инструмента Docker. Его главным функциональным компонентом является движок (docker engine), который включает в себя:

• докер-демон – сервис, запущенный с правами суперпользователя в хостовой ОС, который управляет docker объектами (образами, контейнерами, сетью);

• докер-клиент – консольная утилита, через которую пользователь производит управление Docker-демоном;

• программный интерфейс (Docker Engine API) – интерфейс, который докер-клиент и докер-демоном используют для взаимодействия.

Рассмотрим программные механизмы Linux, используемые docker для поддержания изоляции контейнеров:

1. Контрольные группы (cgroups):

Контрольные группы – функции ядра Linux, позволяющие контролировать потребление системных ресурсов процессами (процессорное время, память, …). В приложении к Docker контрольные группы применяются для поддержания желаемого распределения системных ресурсов хостовой ОС между запущенными на ней контейнерами.

1. Пространства имен (namespaces):

Пространства имён – механизм ядра Linux, позволяющий изолировать ресурсы ОС для экземпляров процессов. Помещая процесс в пространство имен, можно ограничить ресурсы, видимые данному процессу.

При старте ОС создается пространство имен каждого типа, они называются инициализирующими или корневыми. Далее в них можно создавать дочерние пространства имен и помещать в них процессы. Процесс всегда находится ровно в одном пространстве имен каждого типа.

В версии ядра 5.11 присутствуют 8 пространств имен:

• Cgroup (контрольные группы):

Пространства имен cgroup изолируют информацию, связанную с контрольными группами процессов. Этот тип пространств имен позволяет скрыть структуру виртуальной файловой системы хостовой ОС для контейнеров.

• IPC (ресурсы межпроцессного взаимодействия):

Пространства имен IPC управляют ресурсами межпроцессного взаимодействия, а именно, объектами IPC System V и очередями сообщений POSIX. Любой объект, созданный в пространстве имен IPC, виден только процессам, находящимся в том же пространстве имен.

• Network (сетевые ресурсы):

Каждое отдельное пространство имен Network имеет свой набор сетевых ресурсов, к ним относятся список IP-адресов, список сокетов, таблица IP-маршрутизации, брандмауэр. Каждый физический или виртуальный сетевой интерфейс может находиться только в одном пространстве имен Network.

• Mount (точки монтирования файловой системы);

• PID (идентификаторы процессов):

Пространства имен PID предоставляют процессам независимый от других пространств набор идентификаторов, что позволяет нескольким процессам иметь один и тот же идентификатор в разных пространствах имен PID. Исходное пространство имен PID может видеть все процессы, хотя их идентификаторы будут отличаться от тех, что эти процессы имеют во вложенных пространствах имен. В каждом пространстве имен PID есть инициализирующий процесс, чей идентификатор всегда равен единице.

• Time (системное время):

Пространства имен Time позволяют процессам видеть разное системное время.

• User (идентификаторы пользователей и групп):

Пространства имен User изолируют идентификаторы пользователей и групп, корневой каталог, ключи и привилегии. Процесс может иметь разные идентификаторы пользователя и группы в разных пространствах имен User. Например, непривилегированный в корневом пространстве имен процесс может иметь права суперпользователя в одном из вложенных пространств.

• UTS (хостовые и доменные имена):

Пространства имен UTS обеспечивают изоляцию хостовых и доменных имен. Изменения этих идентификаторов видны всем процессам, находящимся в одном и том же пространстве имен UTS, но невидимы для процессов, принадлежащих разным пространствам.

С точки зрения хостовой ОС контейнер представляет собой некий процесс. Благодаря пространствам имен он имеет ограниченное представление о ресурсах хостовой ОС. Docker-демон при запуске контейнера создает для него индивидуальный набор пространств имен 6 типов: IPC, Network, Mount, PID, UTS, cgroup. В область видимости контейнера попадают только те объекты ядра хостовой ОС, которые были ему выделены или им же порождены. Так, например, контейнер видит только те процессы, которые были запущены им самим. При этом хостовая ОС видит их тоже, потому что, пространства имен PID являются вложенными.

Экземпляр контейнера строится на основе образа — шаблона, который содержит как само приложение, так и все необходимые для его работы компоненты: среду исполнения, библиотеки, переменные среды, файлы конфигурации. Сборка образа производится согласно инструкциям конфигурационного файла Dockerfile.

В Docker для хранения данных образа используется технология каскадно-объединённого монтирования файловых систем, благодаря чему данные образа представлены в виде слоев данных. Каждый слой данных является набором отличий от слоя, находящегося перед ним. Каждая последующая команда Dockerfile, вносящая изменения в содержимое уже имеющихся слоев, при сборке образа добавляет новый слой.

Слои данных образа находятся в режиме «только для чтения». При инициализации контейнера поверх слоев данных образа накладывается слой, для которого разрешена запись. Все изменения, произведенные в процессе работы контейнера, такие как запись новых файлов, изменение и удаление существующих файлов, записываются на этот слой данных. Так как слои данных образа не модифицируются, один образ может использоваться сразу несколькими контейнерами, что позволяет избежать дублирования данных и экономит время при запуске контейнеров.

**ВЫПОЛНЕНИЕ РАБОТЫ**

1. **Установка docker:**

Войдем в систему под административной учетной записью и перейдем к файлу /etc/apt/sources.list для настройки репозиториев, откуда пакетный менеджер apt сможет устанавливать пакеты.

Добавим в содержимое файла ссылки на следующие репозитории:

deb https://dl.astralinux.ru/astra/stable/1.7\_x86-64/repository-main/

deb https://dl.astralinux.ru/astra/stable/1.7\_x86-64/repository-update/

deb https://dl.astralinux.ru/astra/stable/1.7\_x86-64/repository-base/

deb https://dl.astralinux.ru/astra/stable/1.7\_x86-64/repository-extended/

deb https://dl.astralinux.ru/astra/stable/1.7\_x86-64/repository-extended/

deb https://dl.astralinux.ru/astra/stable/1.7\_x86-64/uu/last/repository-update/

Таким образом файл примет вид, представленный на рисунке 1:

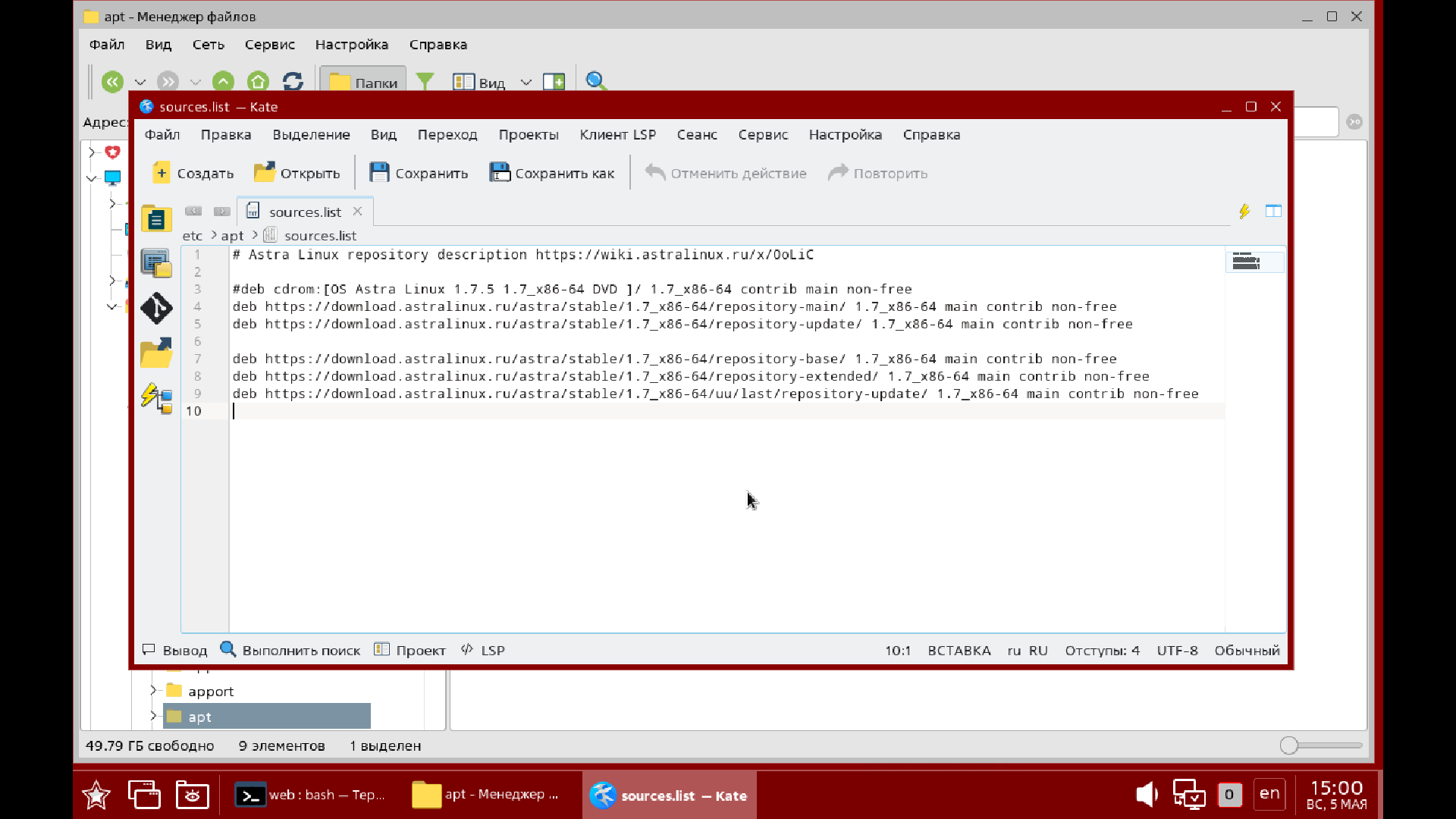


Рисунок 1 – Настройка репозиториев для пакетного менеджера apt

Перейдем к терминалу и выполним от имени суперпользователя сначала команду apt update для обновления списка репозиториев, а затем команду apt install docker.io для установки docker (рисунок 2):

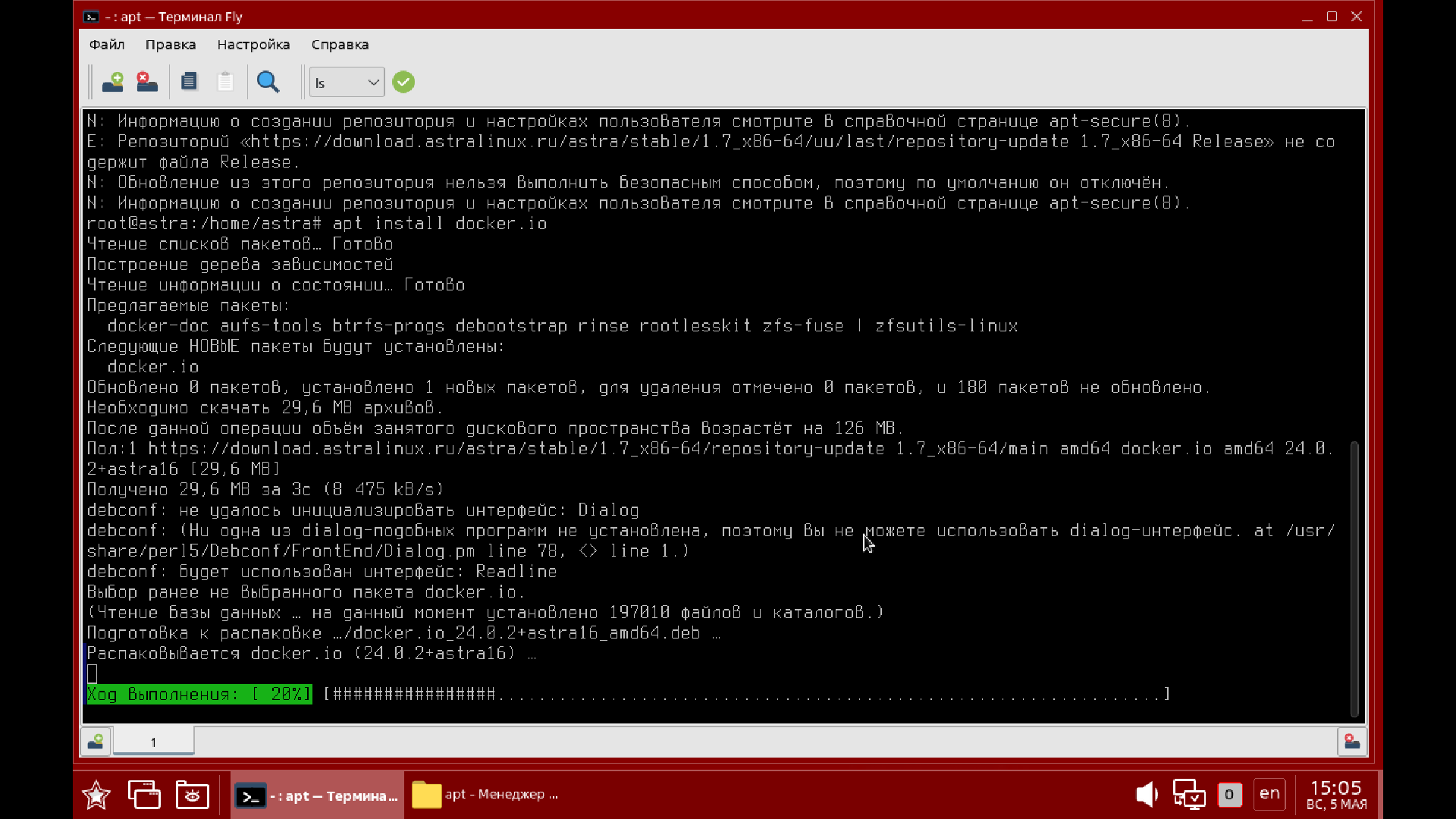


Рисунок 2 – Установка пакета docker

При корректной установке после ее завершения вывод команды docker --help будет выглядеть следующим образом (рисунок 3):

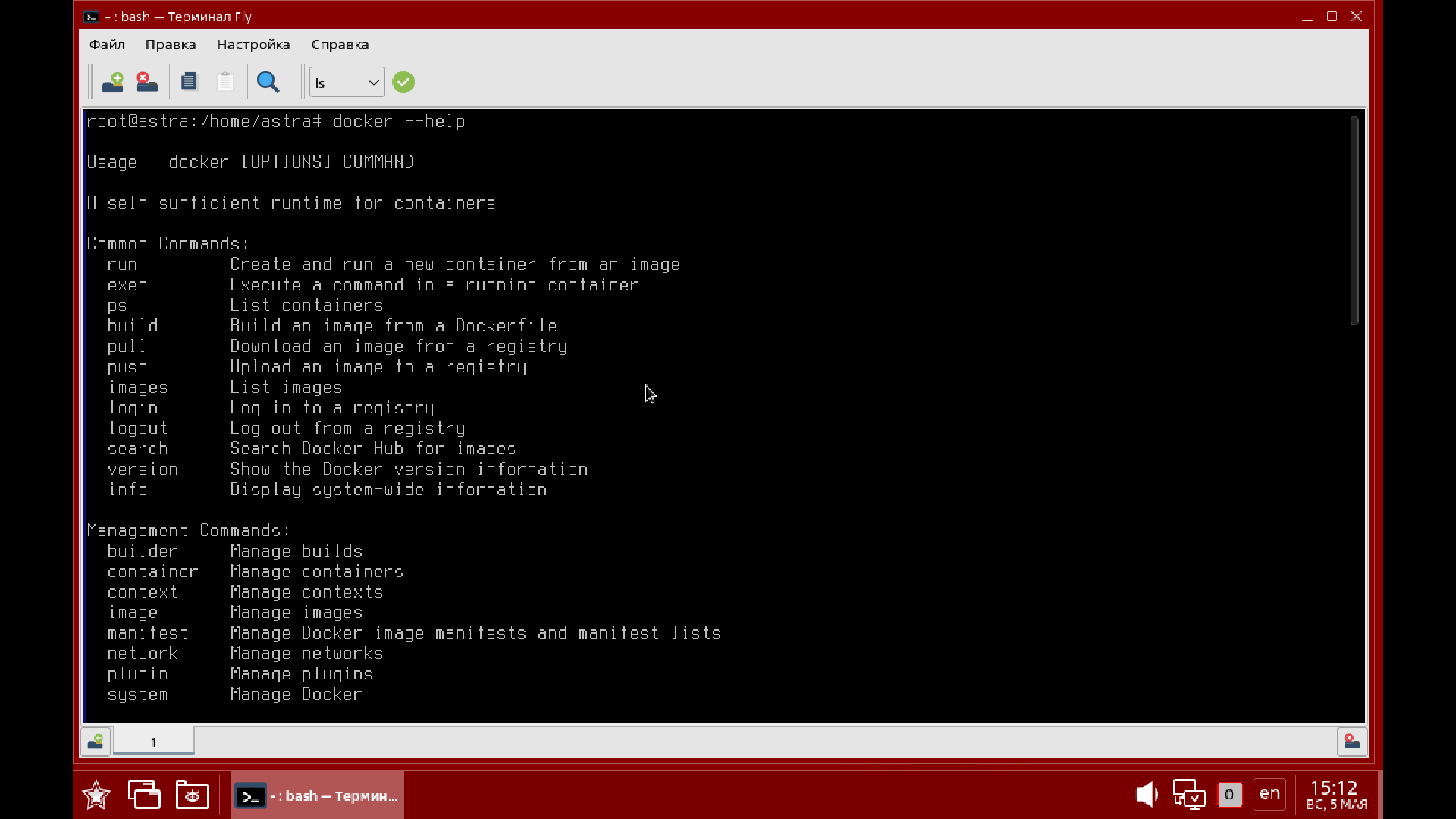


Рисунок 3 – Корректная установка docker

Для работы с функциональностью docker без использования команды sudo добавим рабочего пользователя astra в группу docker командой addgroup astra docker (рисунок 4):



Рисунок 4 – Добавление пользователя astra в группу docker

1. **Сборка образа:**

Подготовим файлы для запуска веб-сервера. Создадим в домашней директории папку web, а в ней – файл server.py, в котором будет содержаться код для запуска веб-сервера (рисунок 5):

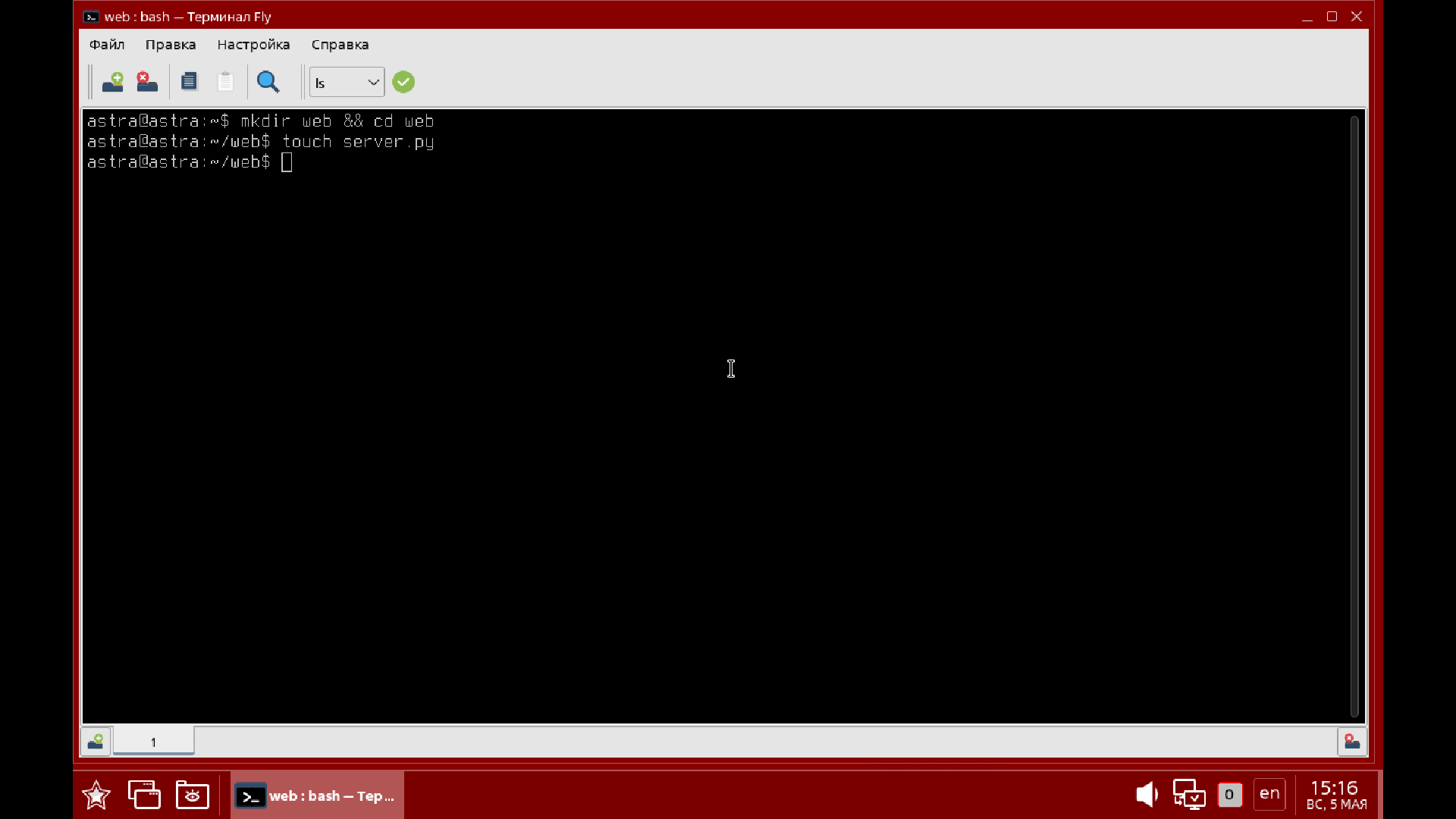


Рисунок 5 – Подготовка файла сервера

Запишем в созданный файл server.py код для запуска веб-сервера с использованием фреймворка Flask (рисунок 6):

#импортируем модуль с фреймворком Flask

from flask import Flask

#создаем объект веб-приложения

app = Flask(\_\_name\_\_)

#определяем поведение для корневого маршрута

@app.route("/")

def start():

return "Hello from container!"

#поднимаем приложение на порту 3333 при запуске файла

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host=’0.0.0.0’, port=3333)

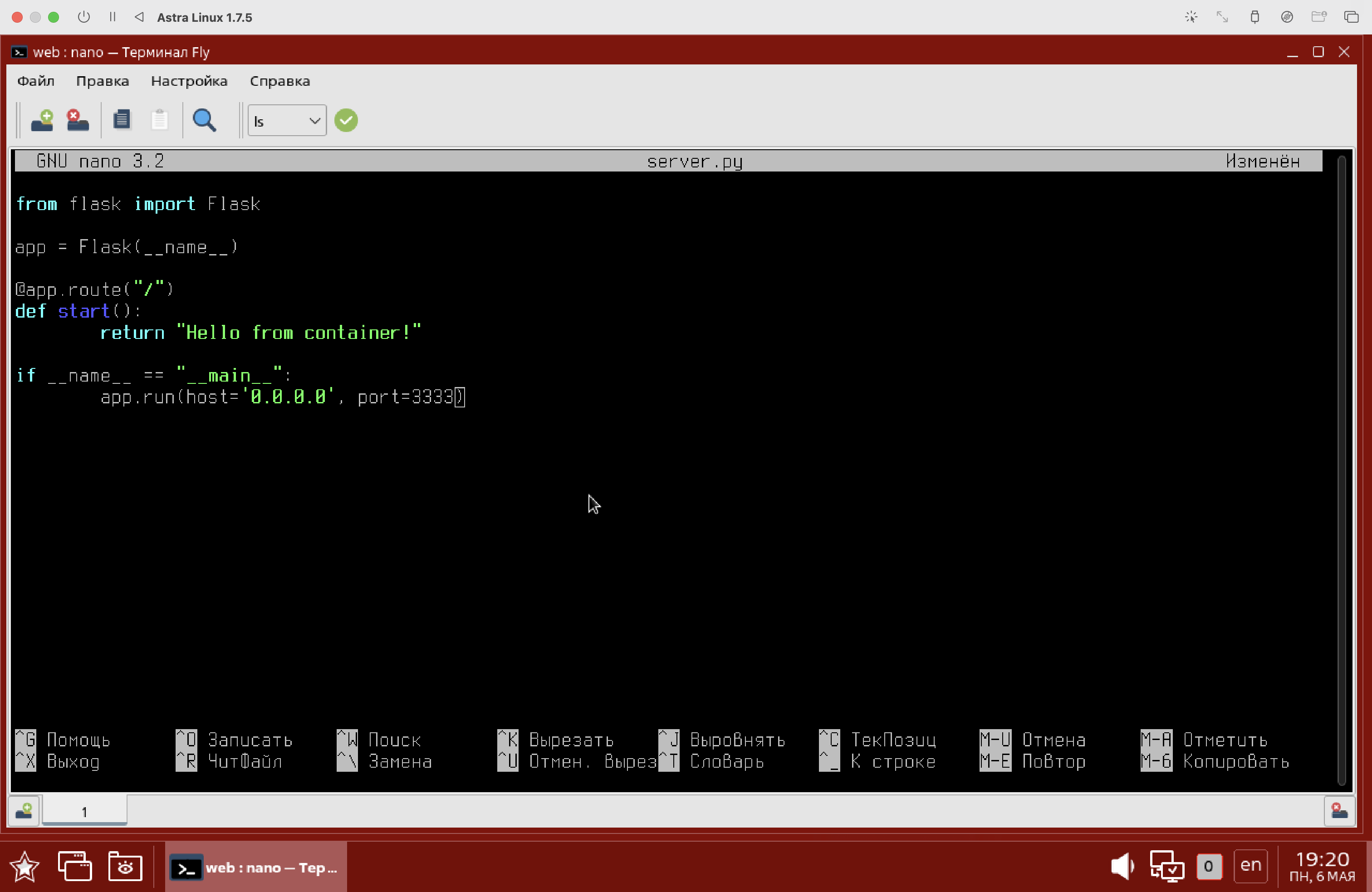


Рисунок 6 – Содержимое файла server.py

Теперь создадим файл requirements.txt и запишем в него требуемый для работы приложения пакет flask версии 3.0.3 (рисунок 7):

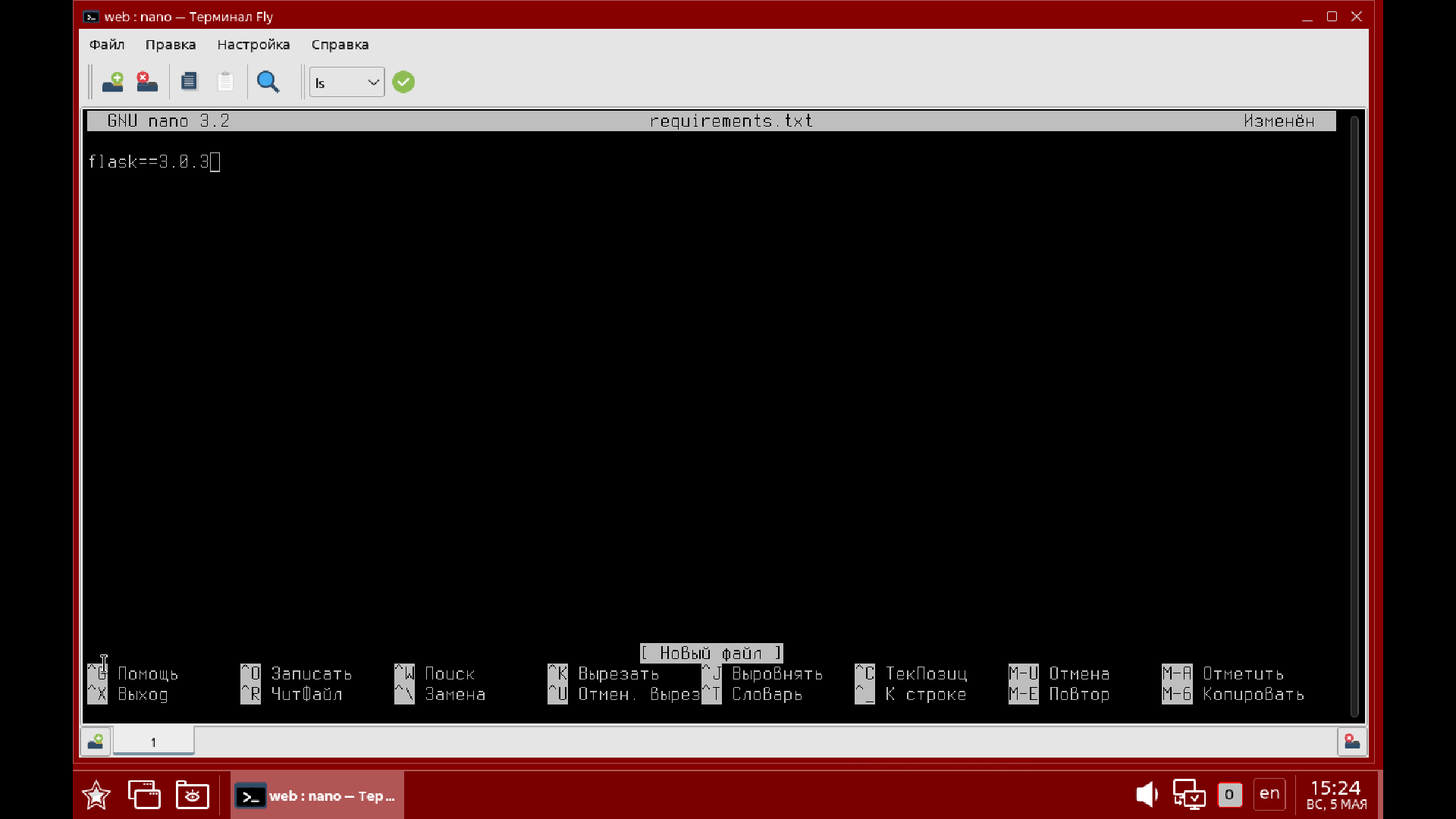


Рисунок 7 – Содержимое файла requirements.txt

Начнем сборку образа с создания файла с инструкциями Dockerfile.flask. В качестве первой инструкции в файле укажем базовый образ с python версии 3.9, на котором будет производиться сборка образа. Для этого используем инструкцию FROM, в качестве аргумента которой задается образ в формате <имя образа>:<тег образа>. Таким образом, первая инструкция в докерфайле будет иметь вид:

FROM python:3.9

#FROM – инструкция для указания базового образа

#python:3.9 – имя и тег базового образа

Далее инструкцией WORKDIR укажем на директорию внутри файловой системы образа, в которой будут выполняться дальнейшие действия. Данная инструкция имеет следующий синтаксис WORKDIR <абсолютный или относительный путь к директории внутри файловой системы образа>. Запишем в докерфайл инструкцию следующего вида:

WORKDIR /home/app

#WORKDIR – инструкция для указания рабочей директории

#/home/app – директория, внутри которой будут выполняться дальнейшие команды

Теперь с использованием инструкции COPY перенесем файлы сервера и его зависимостей в собираемый образ. Инструкция имеет следующий синтаксис COPY <файл или директория на хосте> <файл или директория внутри образа>. Путь к файлу или директории может задаваться как в абсолютном виде, так и в относительном:

COPY ./server.py ./server.py

#COPY – инструкция для переноса данных из файловой системы хоста в файловую систему образа

#./server.py ./server.py – перенос файла server.py из текущей директории файловой системы хоста (в виде абсолютного пути - /home/demo/web/server.py) в текущую директорию файловой системы образа (в виде абсолютного пути - /home/app/server.py)

COPY ./requirements.txt ./requirements.txt

#./requirements.txt ./requirements.txt – перенос файла requirements.txt из текущей директории файловой системы хоста (в виде абсолютного пути - /home/demo/web/requirements.txt) в текущую директорию файловой системы образа (в виде абсолютного пути /home/app/requirements.txt)

Для работы сервера требуется установить зависимости из файла requirements.txt, что возможно с использованием инструкции RUN и пакетного менеджера pip. Инструкция RUN имеет следующий синтаксис RUN <команда для выполнения в командной оболочке образа>. Запишем в докерфайл выполнение команды pip install, которая используется для установки python-пакетов, с флагом -r requirements.txt для передачи зависимостей из данного файла. Команда будет иметь следующий вид:

RUN pip install -r requirements.txt

#RUN – инструкция для выполнения команд в командной оболочке образа

#pip install -r requirements.txt – команда для установки зависимостей из файла requirements.txt

Далее укажем пользователя, от имени которого будет запускаться контейнер. Инструкция для определения пользователя имеет синтаксис USER <идентификатор или имя пользователя>. Запишем ее в файл:

USER 1000

#USER – инструкция для определения пользователя, от имени которого будут выполняться следующие в файле команды

#1000 – идентификатор пользователя

Последними командами укажем процесс, которому будет передаваться управление при запуске контейнера и файл, который будет выполняться:

ENTRYPOINT [“python”]

#ENTRYPOINT – определение процесса, который будет запущен при запуске контейнера

#[“python”] – при запуске контейнера запустится интерпретатор python

CMD [“server.py”]

#CDM – определение команды, которая будет выполнена при запуске контейнера

#[“server.py”] – при запуске контейнера будет запущен файл server.py с использованием определенного ранее интерпретатора python

Таким образом, итоговый файл будет иметь вид (рисунок 8):

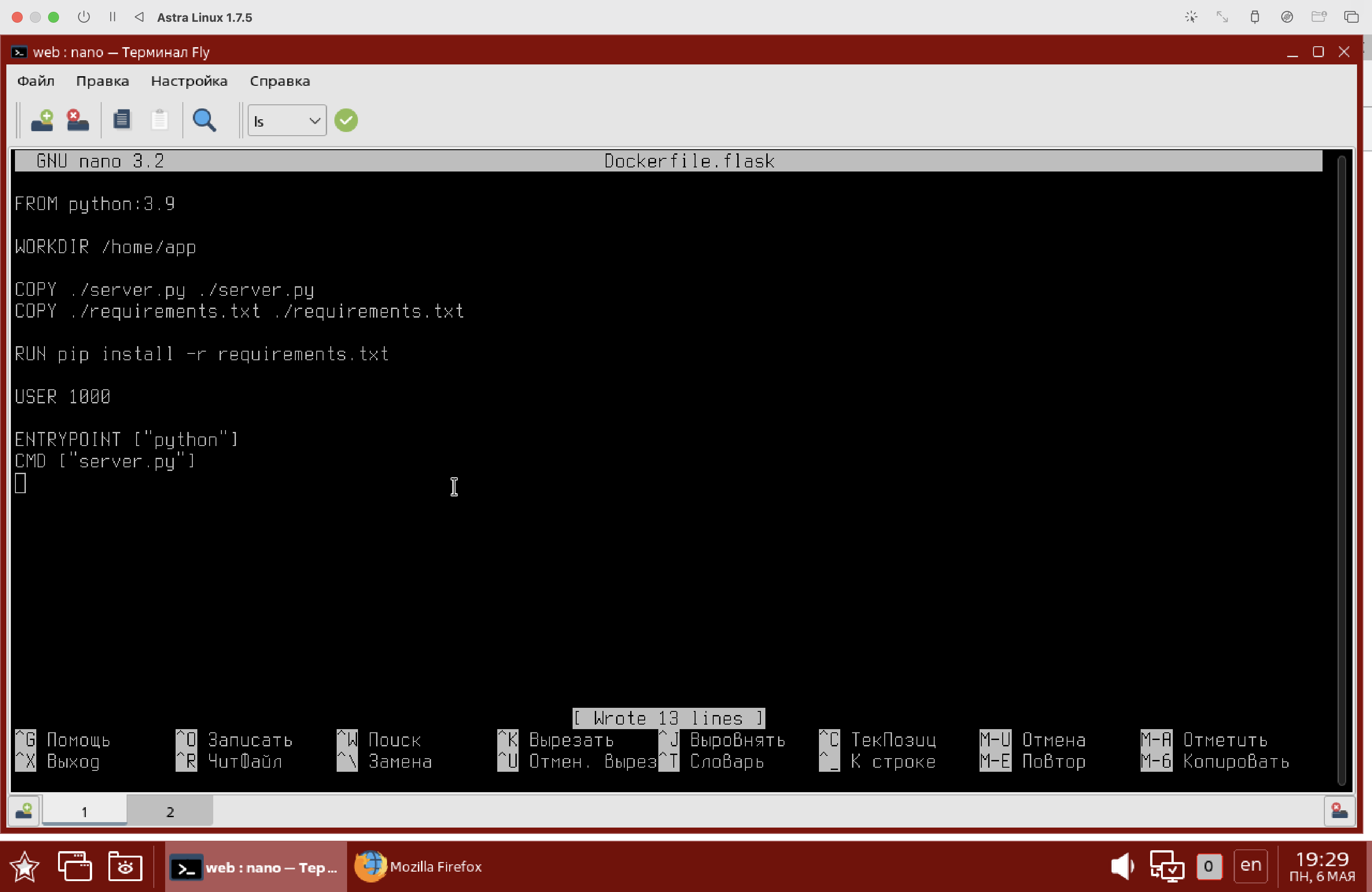


Рисунок 8 – Докерфайл для сборки образа с веб-сервером

Для сборки образа из написанного докерфайла выполним команду docker build со следующими флагами:

- флаг -t задает имя образа в формате <имя образа>:<тег образа>;

- флаг -f задает название файла, из которого будет собираться образ;

- единственный аргумент задает директорию, где лежит докерфайл.

Таким образом, команда для сборки образа будет иметь вид docker build -t flask:latest -f Dockerfile.flask .

Из рисунка 9 видно, что первым этапом производится скачивание базового образа python:3.9.

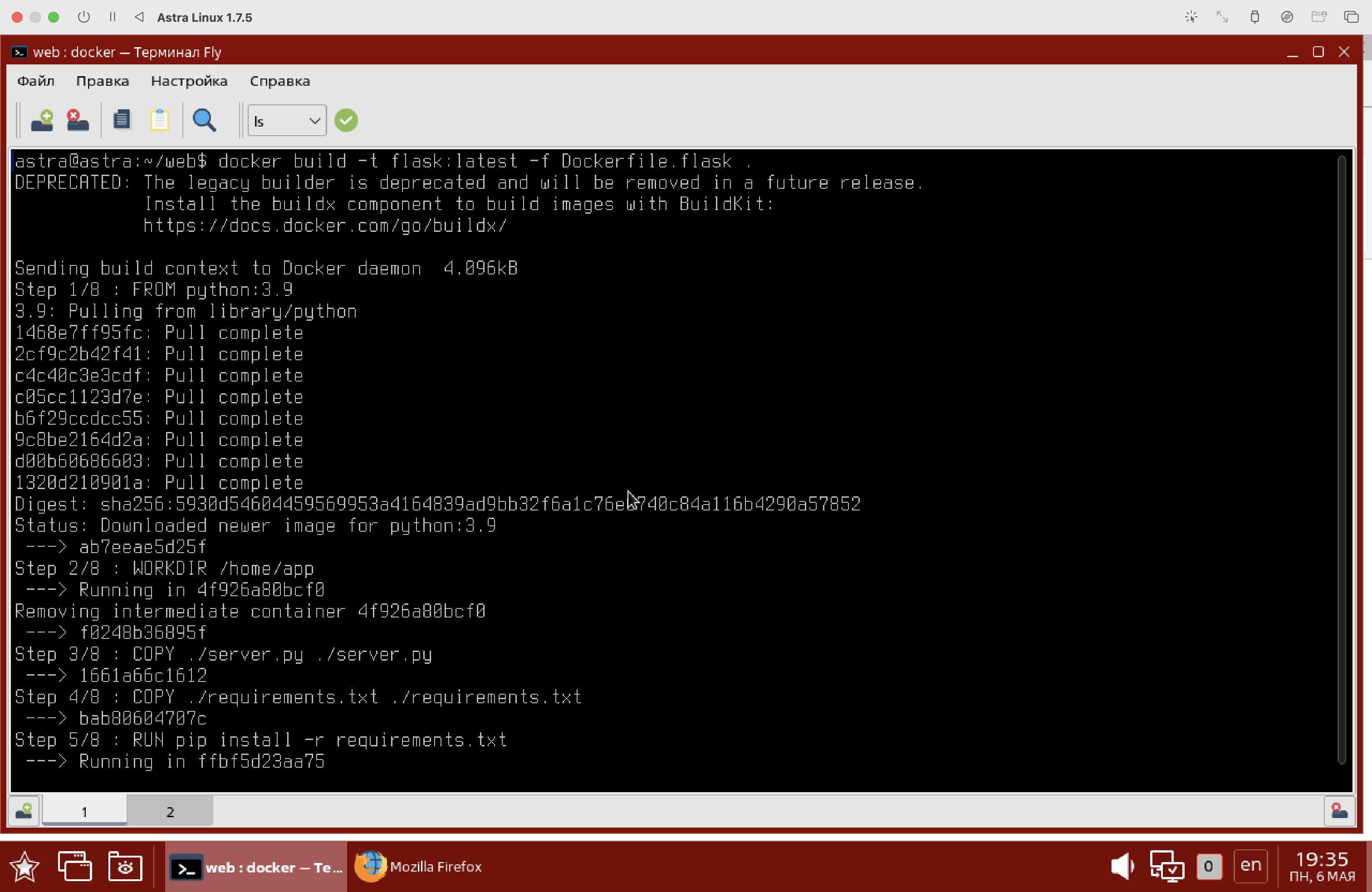


Рисунок 9 – Выполнение первого шага для сборки образа

Затем выполняется команда перехода к директории /home/app, после выполнения команды указывается хэш слоя (рисунок 10).

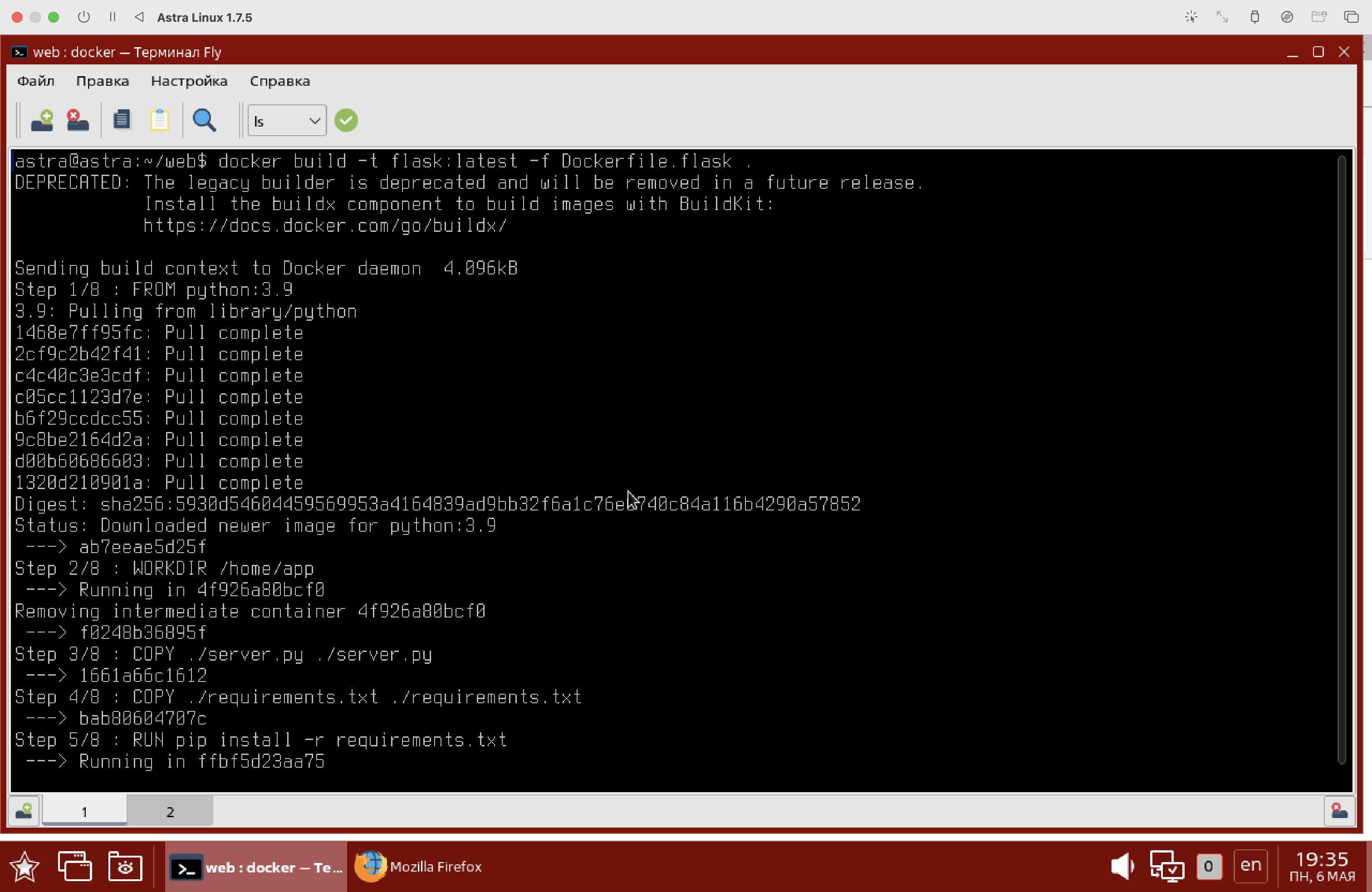


Рисунок 10 – Выполнение второго шага сборки образа

Аналогично производится копирование в образ файлов server.py и requirements.txt на шагах 3 и 4 (рисунок 11):

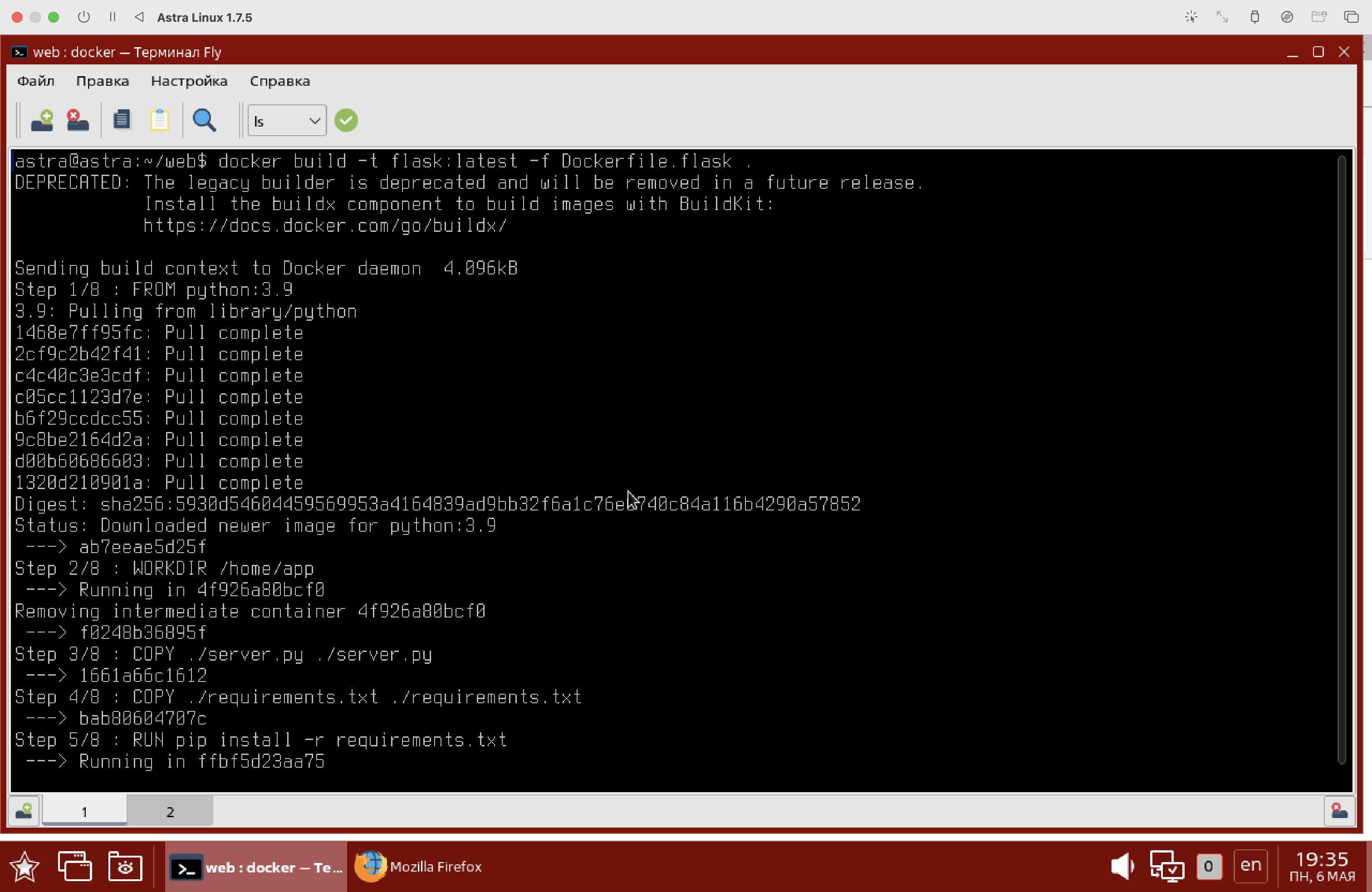
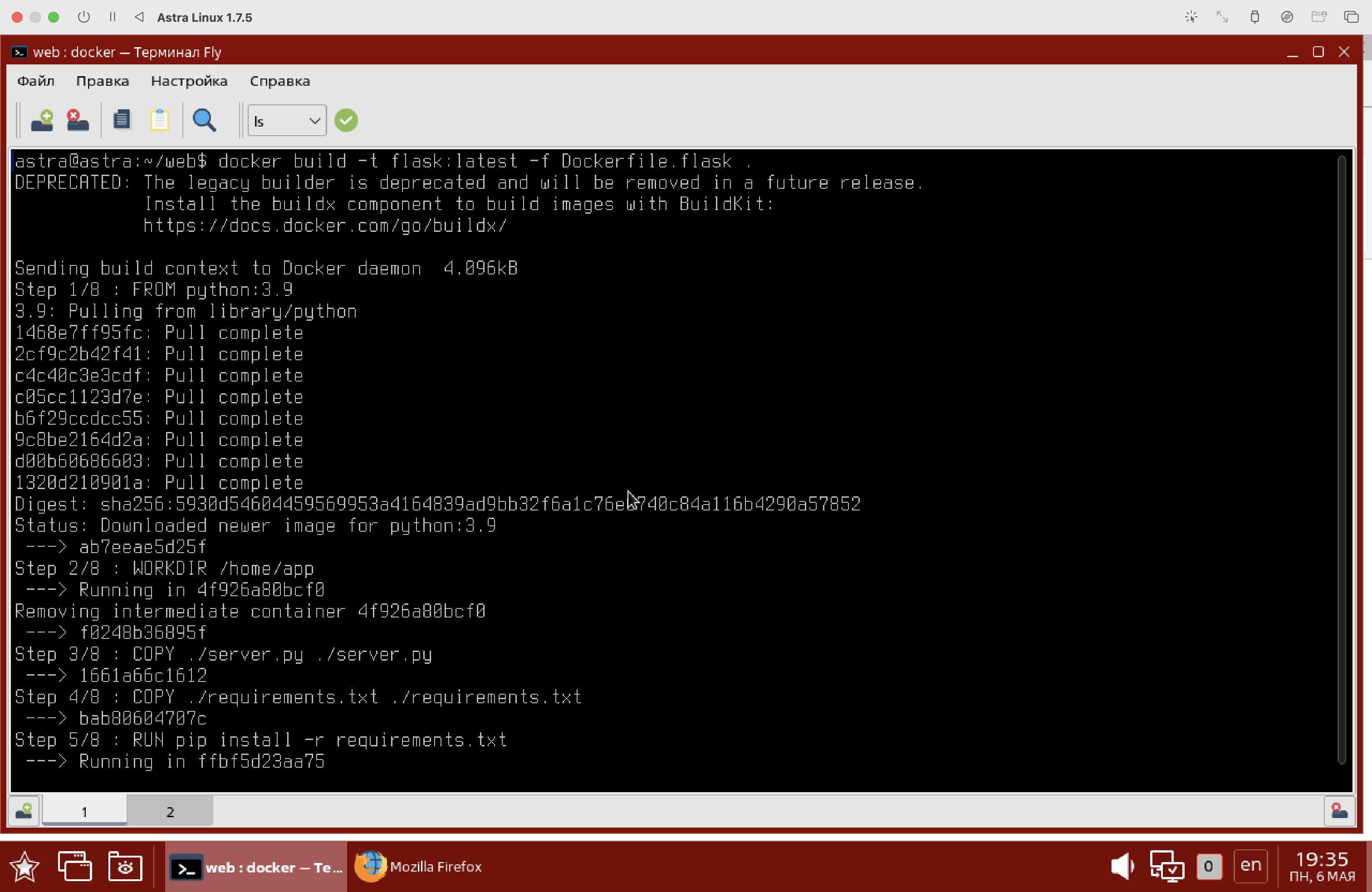


Рисунок 11 – Выполнение третьего и четвертого шагов сборки образа

На пятом шаге происходит выполнение команды pip install для установки python-зависимостей (рисунок 12):



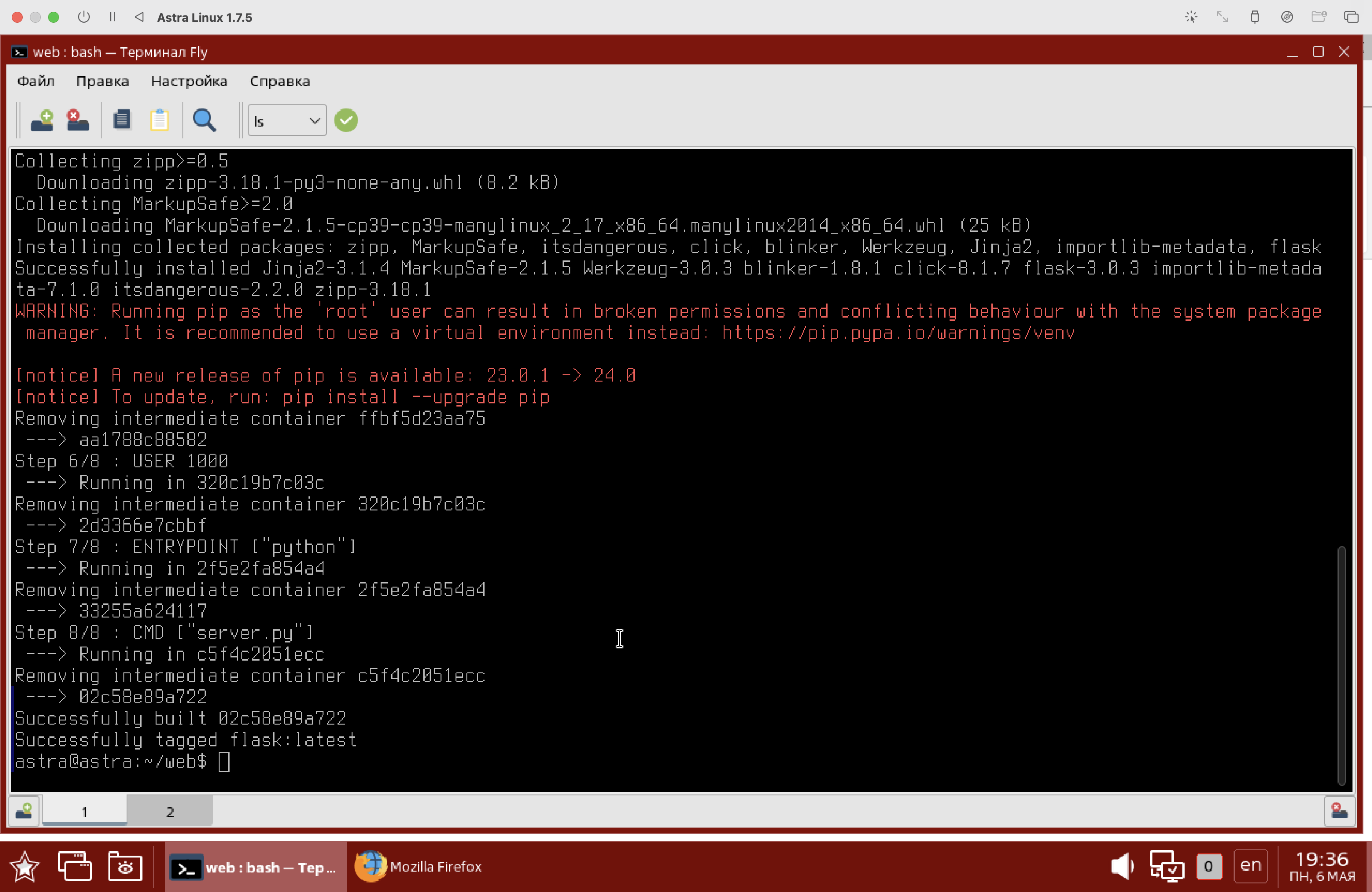


Рисунок 12 – Выполнение пятого шага сборки образа

На шестом шаге происходит установление идентификатора пользователя, от которого будет запускаться контейнер, а на седьмом и восьмом шагах – определение процесса, запускаемого при создании контейнера, и выполняемого файла (рисунок 13):

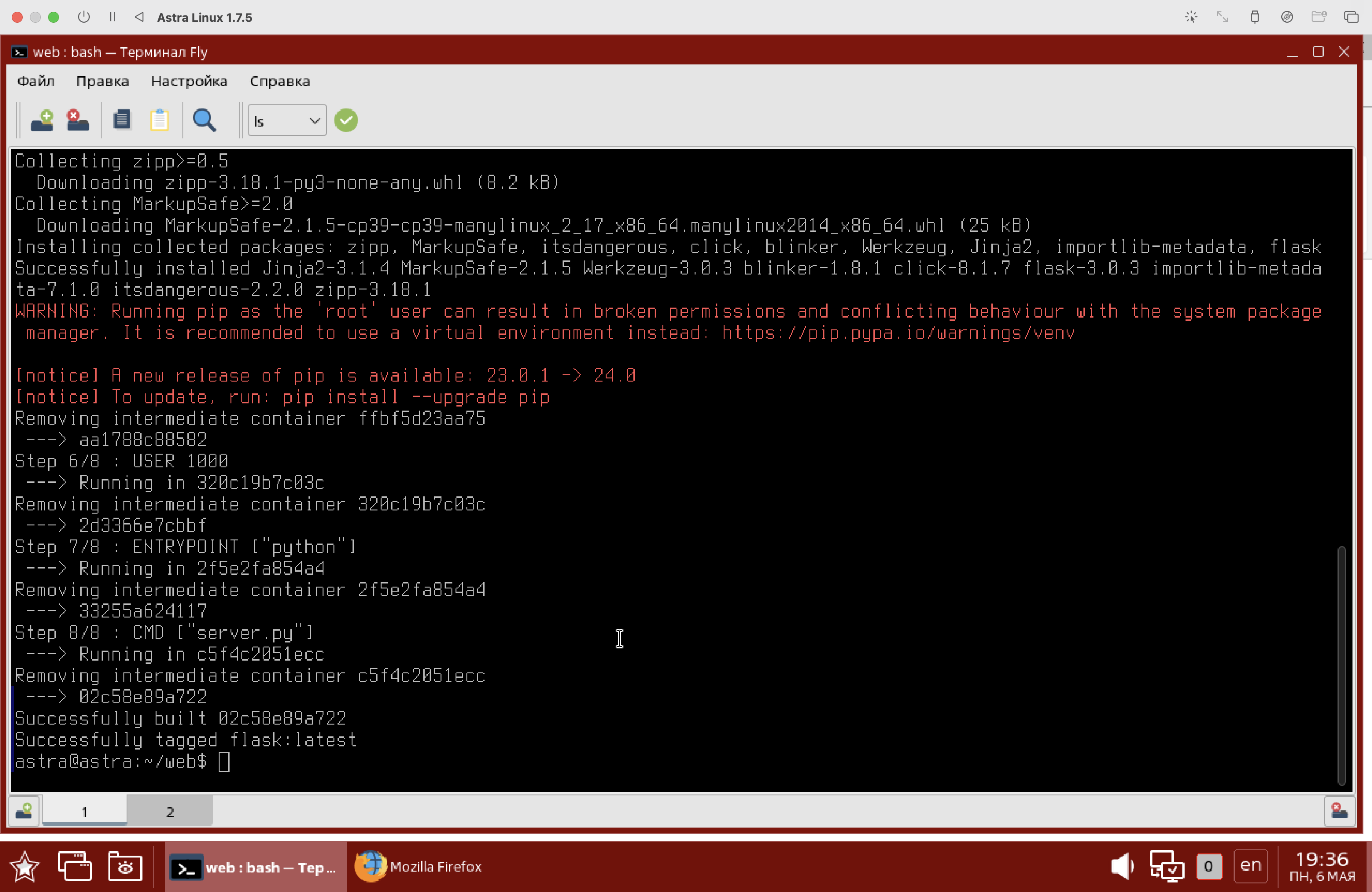


Рисунок 13 – Выполнение шестого, седьмого и восьмого шагов сборки

В последних строках на рисунке 13 видно, что образ был успешно собран и назван flask:latest. Собранный образ можно увидеть с использованием команды docker images (рисунок 14).

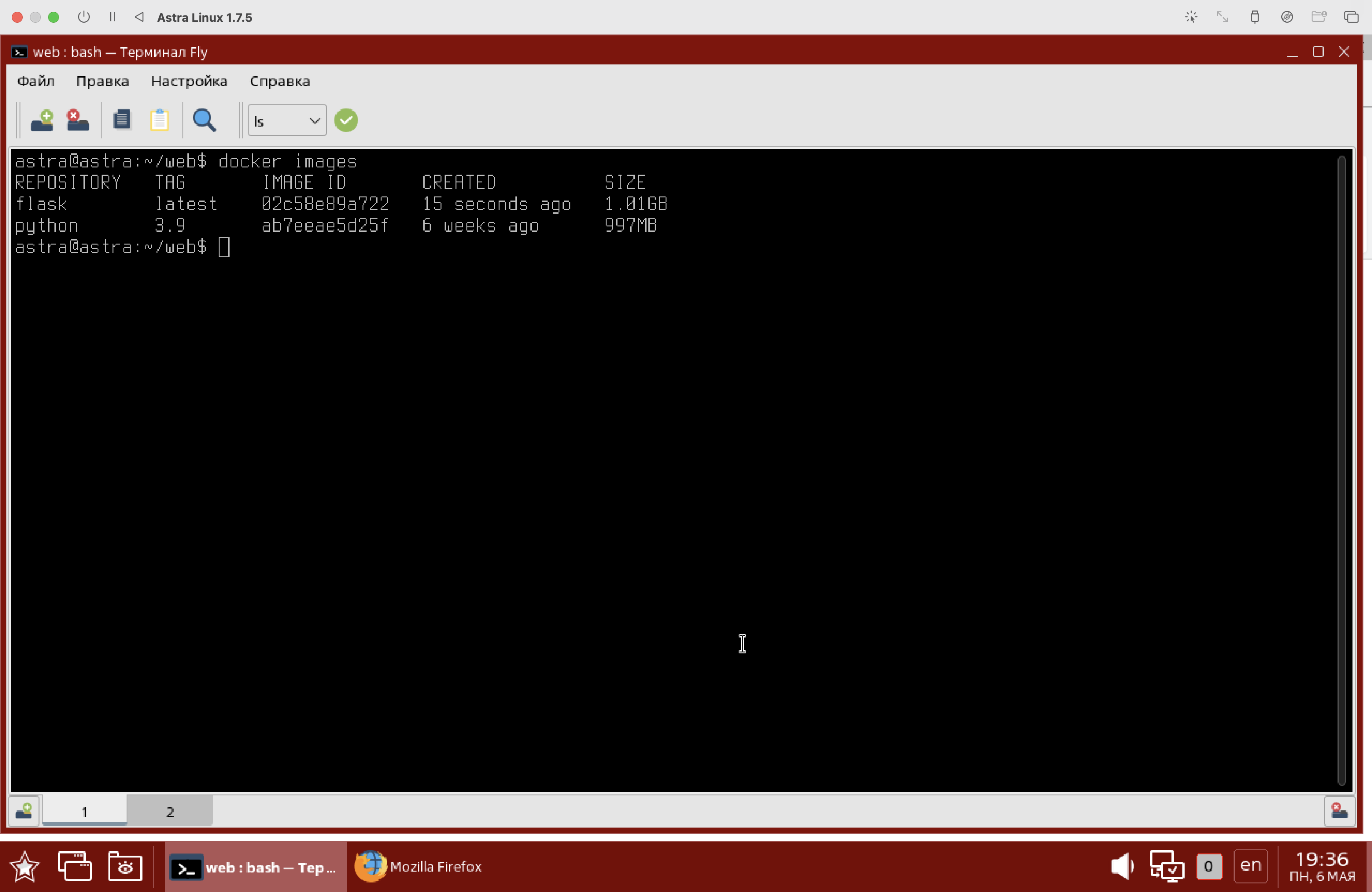


Рисунок 14 – Вывод команды docker images

Как видно из рисунка 14, локально находится не только собранный ранее образ flask:latest, но и образ python:3.9, который является для него базовым.

При выполнении команды docker history можно увидеть все слои образа, причем как основного, так и базового (рисунок 15):

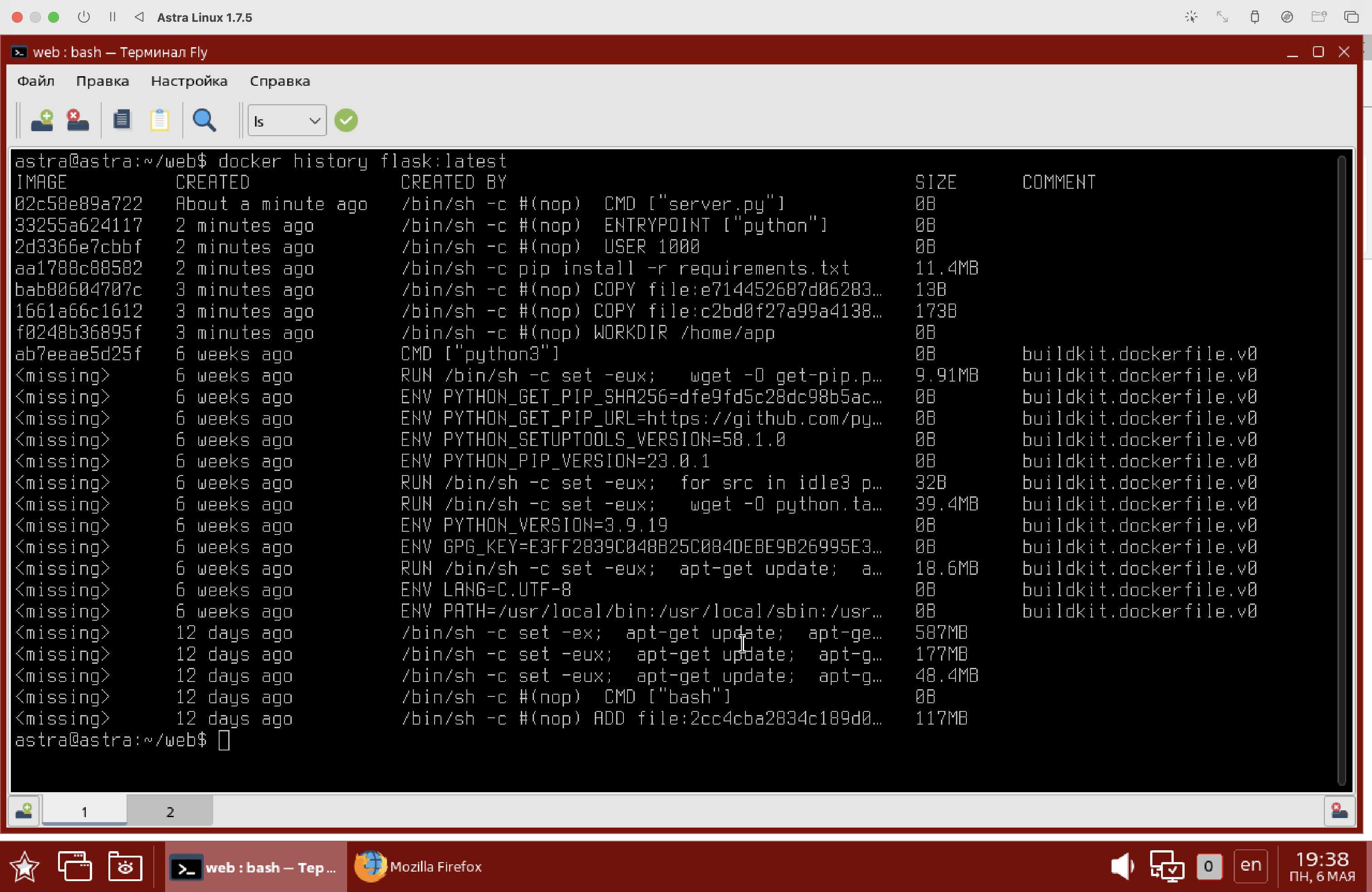


Рисунок 15 – Слои собранного образа

1. **Запуск контейнера:**

Запустим контейнер из собранного образа с использованием команды docker run --rm flask:latest (рисунок 16). Флаг --rm отвечает за удаление контейнера после его завершения.

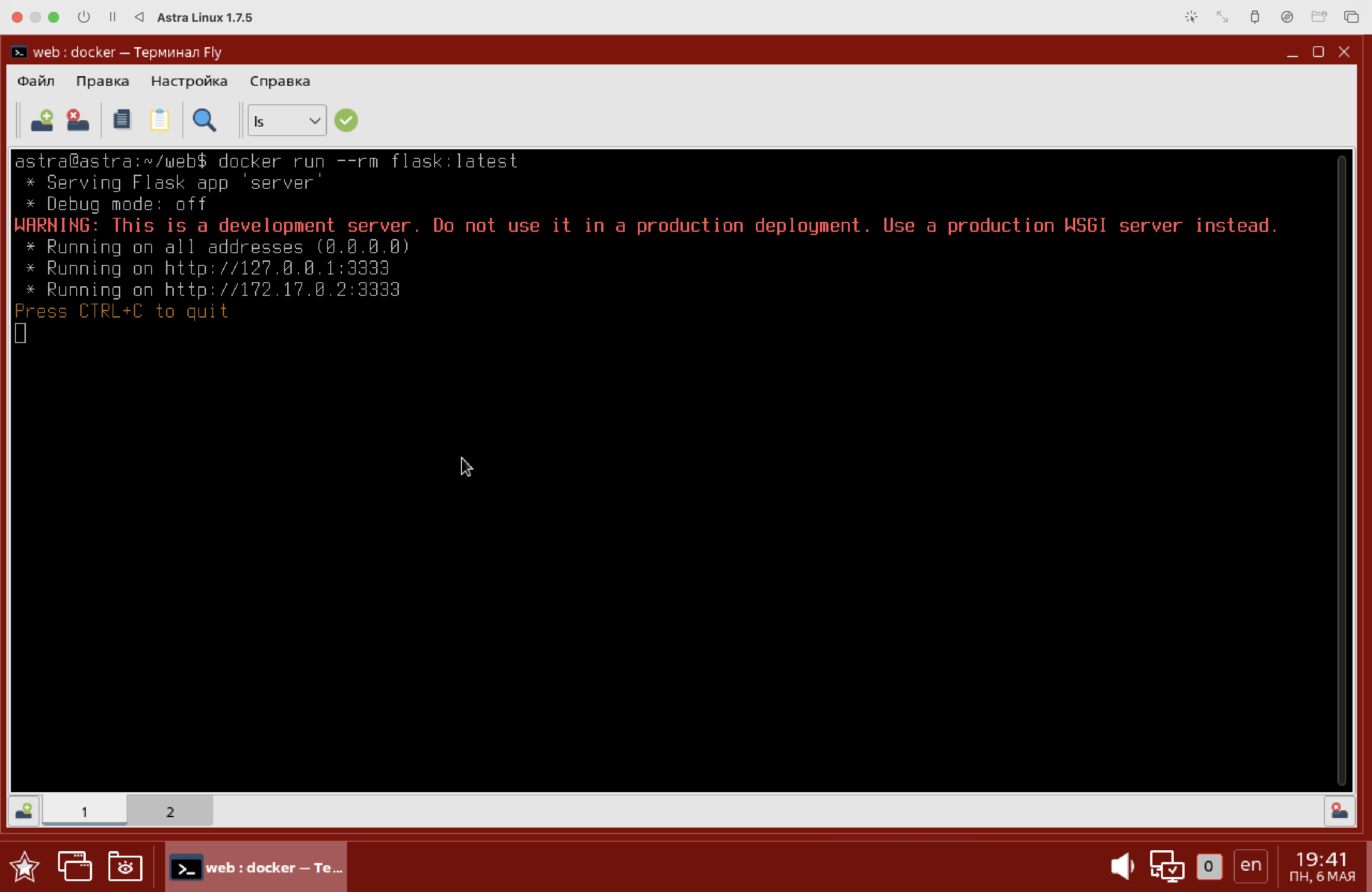


Рисунок 16 – Запуск собранного контейнера с веб-приложением на flask

Видим, что сервер был успешно запущен на порту 3333. Попытаемся получить доступ к содержимому корня сервера с использованием утилиты curl (рисунок 17):

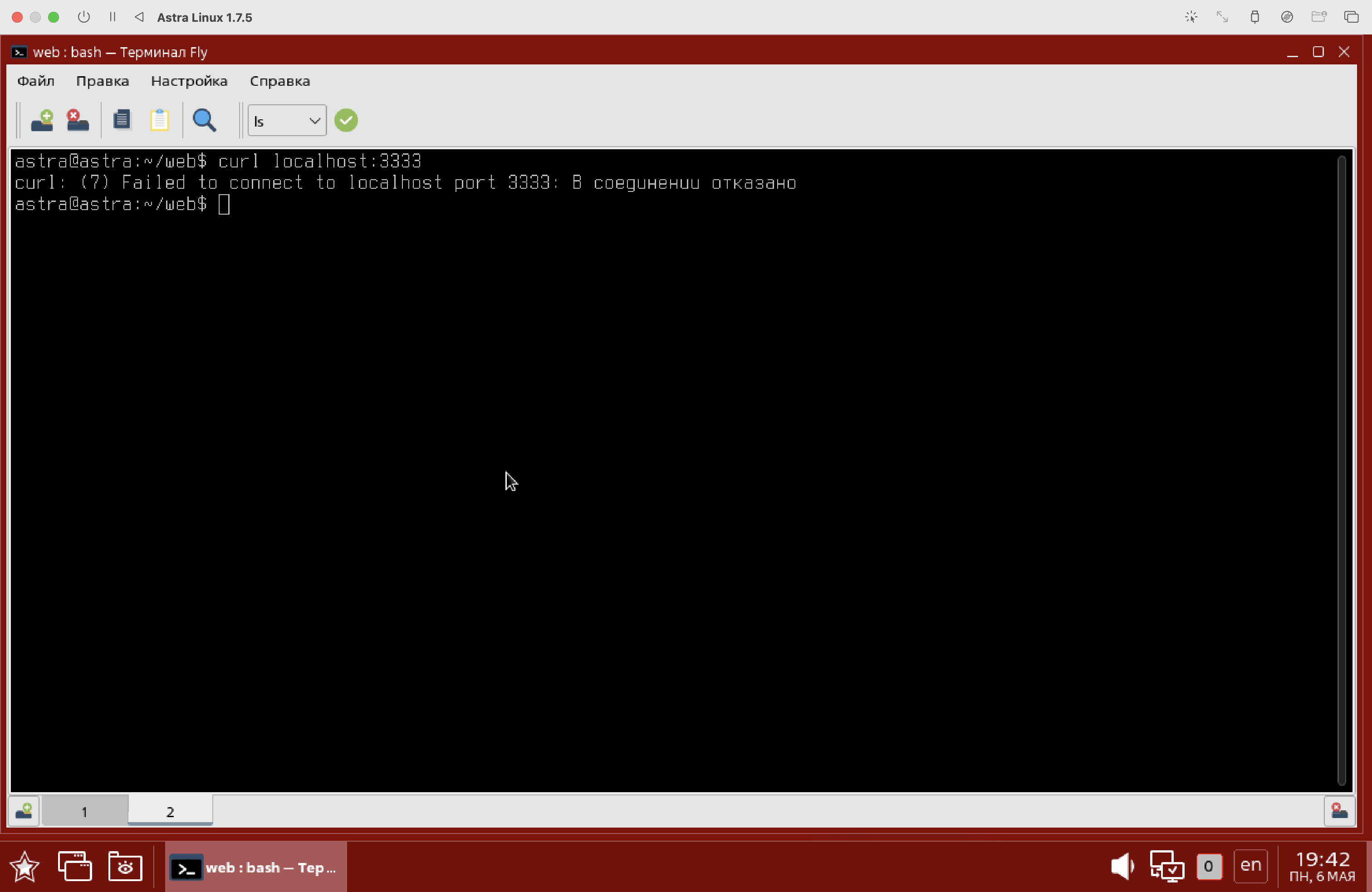


Рисунок 17 – Попытка получения содержимого корня сервера

Как видно из рисунка 18, нам не удалось подключиться к порту 3333 на локальной машине. Если просканировать localhost с использованием nmap, то также можно убедиться, что порт 3333 закрыт:

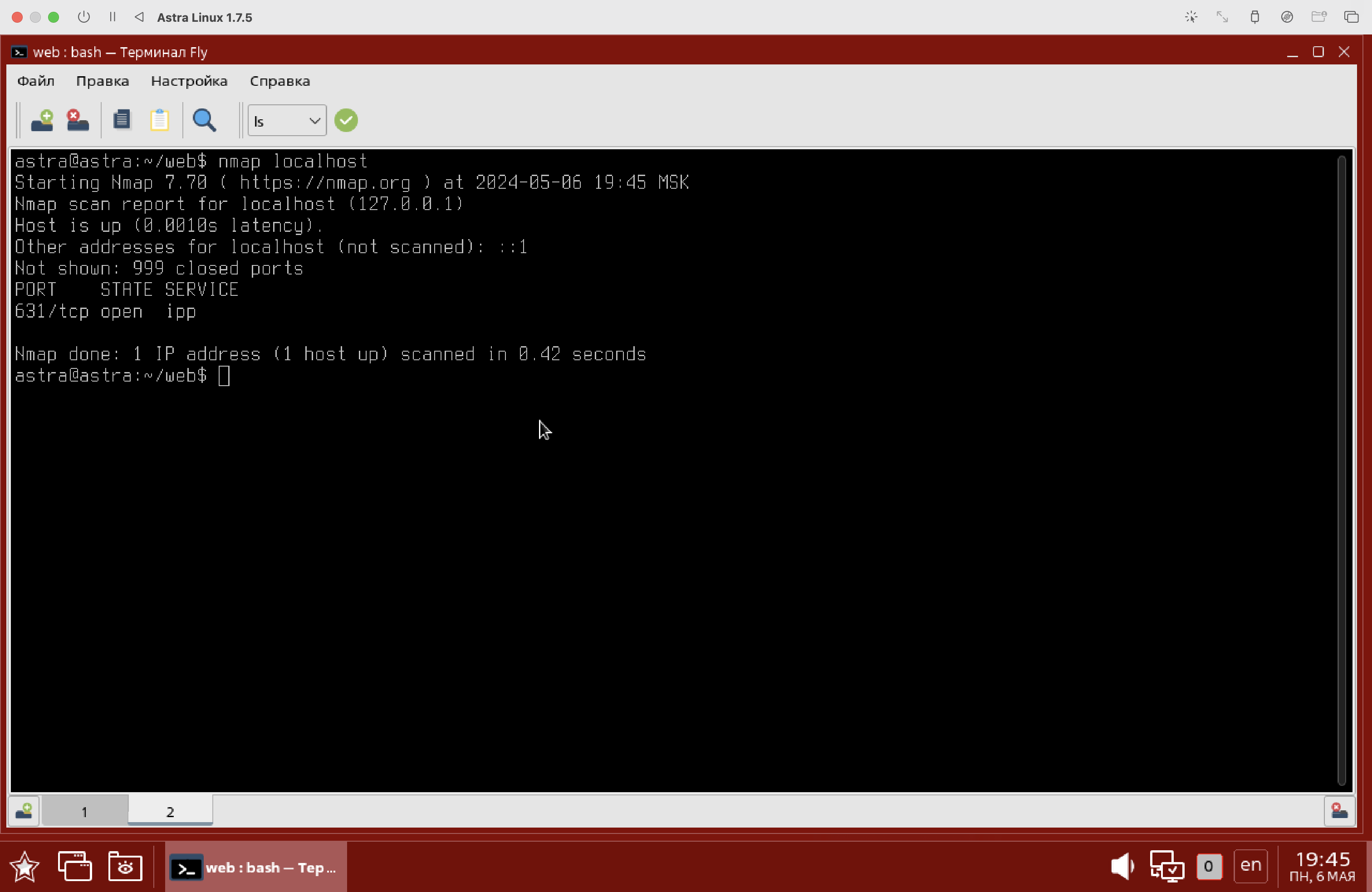


Рисунок 18 – Отсутствие порта 3333 среди открытых

Так происходит, потому что по умолчанию сеть контейнера изолирована от сети хоста. Посмотрим идентификатор процесса, который соотносится с данным контейнером с использованием команды docker ps (рисунок 19):

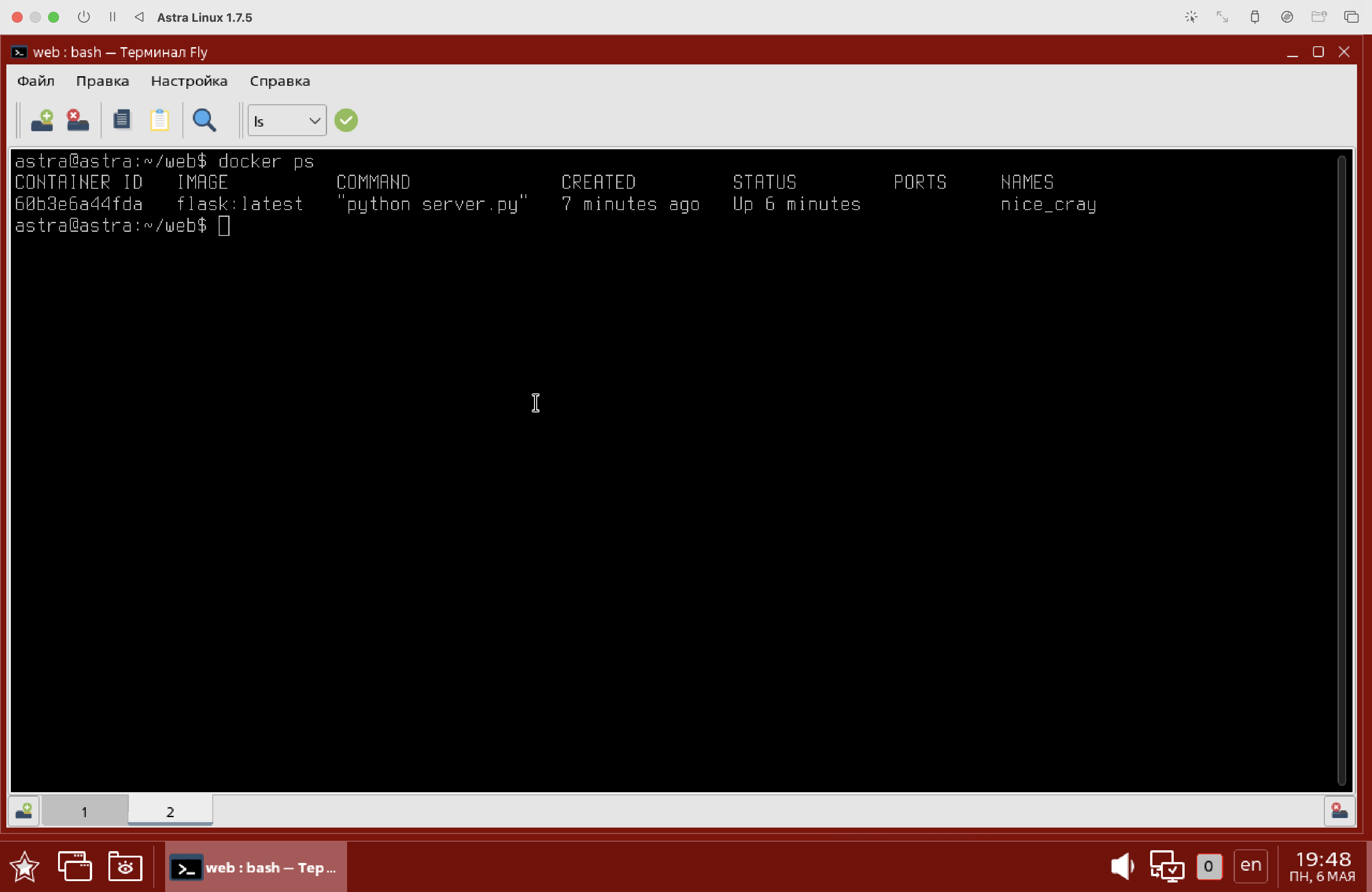


Рисунок 19 – Список запущенных контейнеров

Для того, чтобы войти внутрь контейнера используем команду docker exec -it <идентификатор контенера> /bin/sh. После выполнения данной команды откроется возможность взаимодействия с командной оболочкой sh внутри выбранного контейнера в интерактивном режиме (рисунок 20):

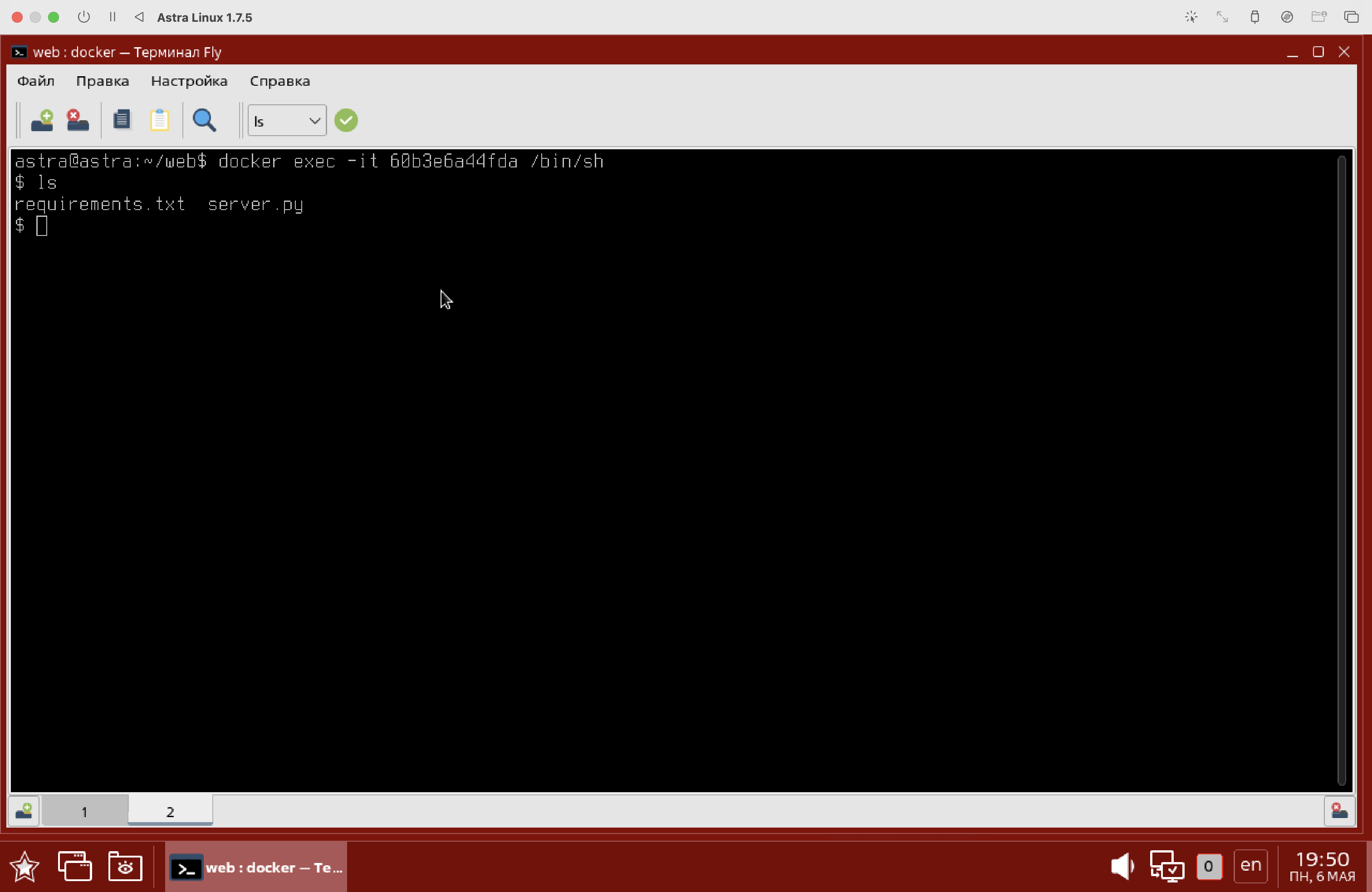


Рисунок 20 – Попадание внутрь контейнера

Теперь попробуем инициировать взаимодействие с сервером изнури контейнера (рисунок 21):

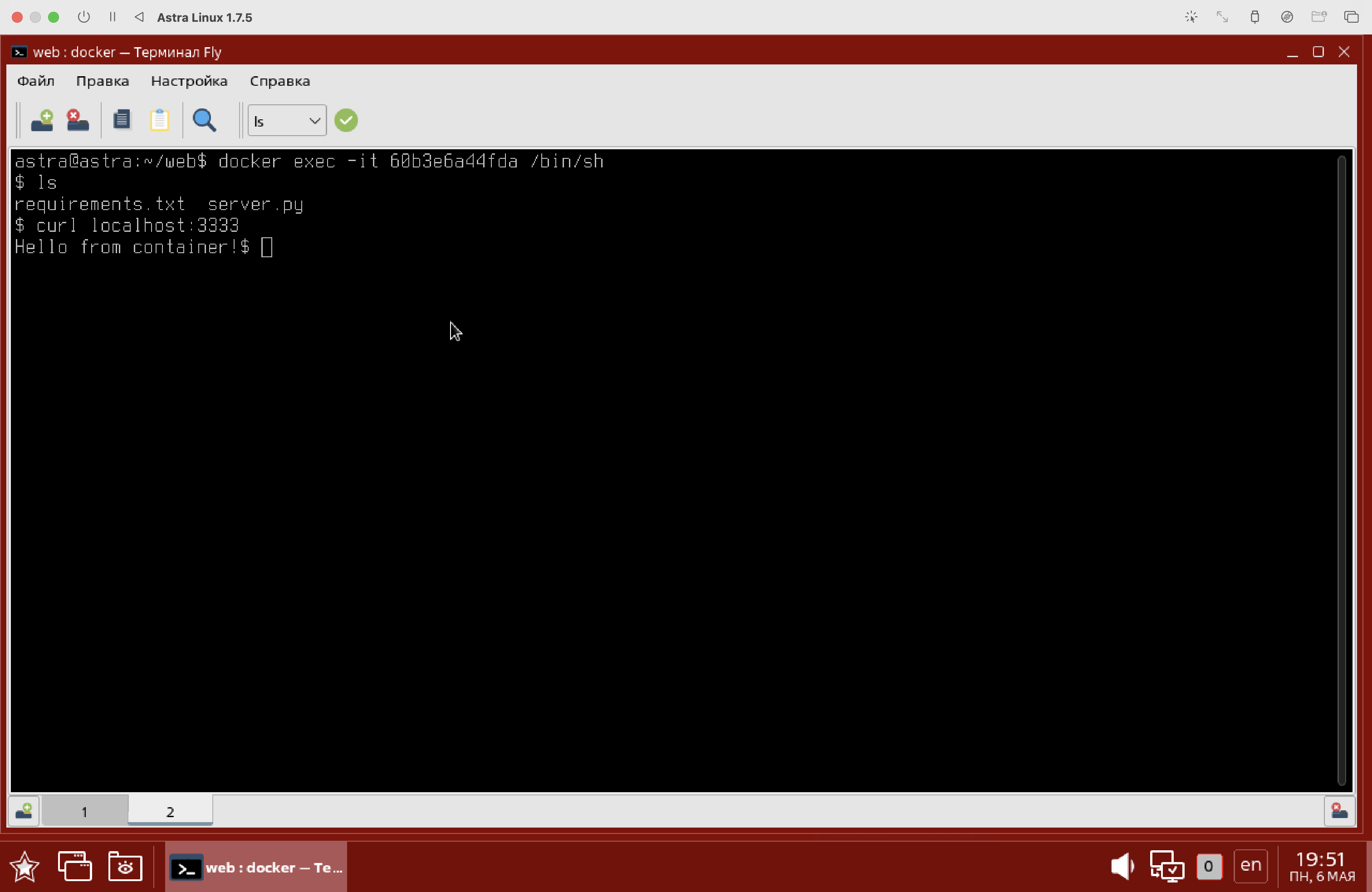


Рисунок 21 – Взаимодействие с сервером изнутри контейнера

Как видно из рисунка 21, взаимодействие прошло успешно и был получен ответ от сервера, однако его сеть является полностью изолированной от сети хоста и взаимодействие с ним возможно только изнутри того же контейнера.

Для того, чтобы порт контейнера, на котором работает веб-сервер, стал доступен с хоста требуется его пробросить во время запуска контейнера. Сначала завершим текущий контейнер либо с использованием сочетания клавиш CTRL+C из сессии терминала, из которой он был запущен, либо с использованием команды docker stop <идентификатор контейнера>.

Заново запустим контейнер из того же образа, но определим дополнительно значение для флага -p со значением вида <порт хоста>:<порт контейнера> для создания сетевого моста между портом контейнера и портом хоста (рисунок 22):

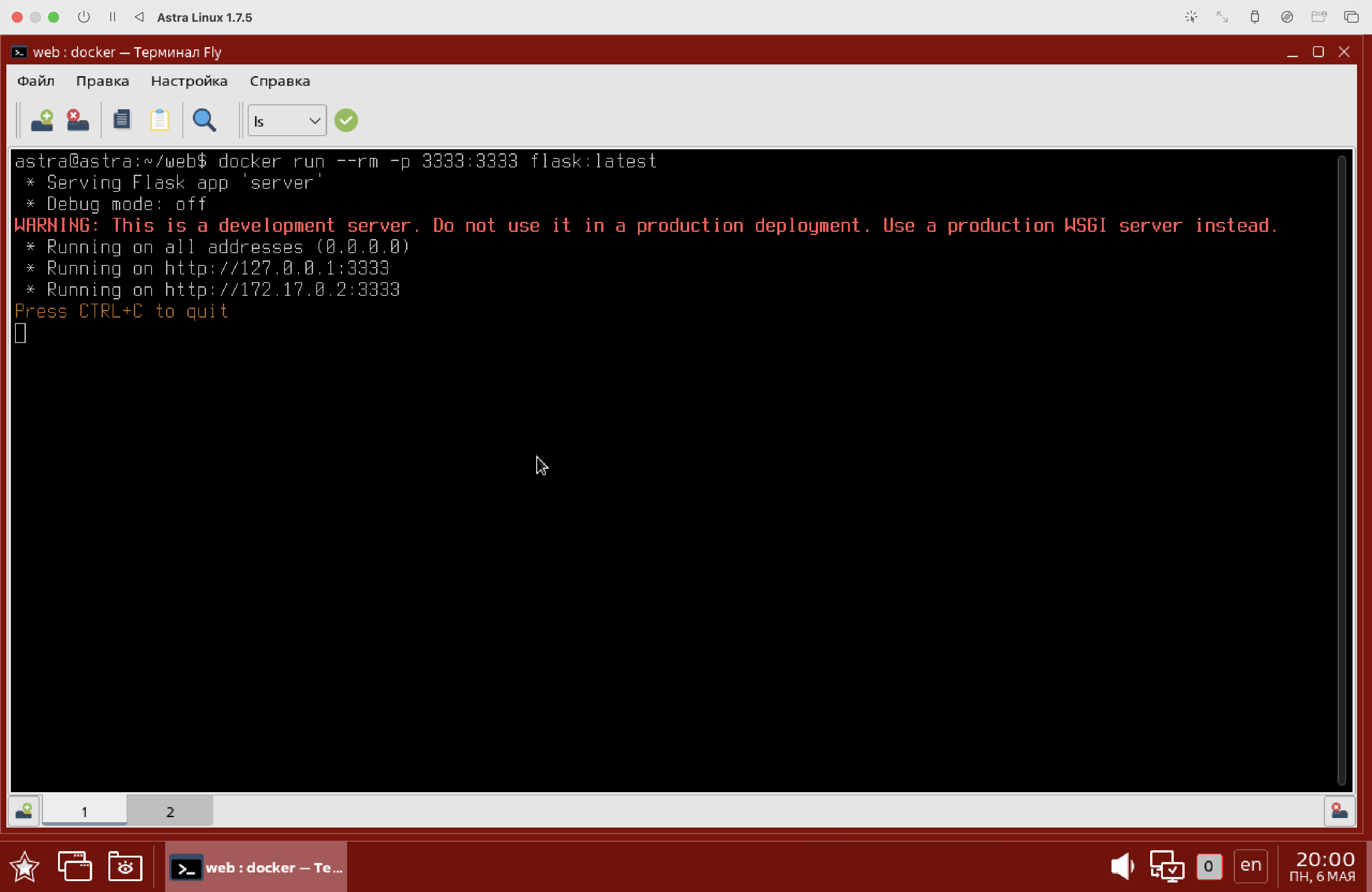


Рисунок 22 – Запуск контейнера с пробросом порта 3333 на хост

Теперь при попытке получить доступ к веб-серверу с хоста с использованием утилиты curl будет получен положительный ответ (рисунок 23), а при помощи nmap можно убедиться, что порт 3333 является открытым (рисунок 24):

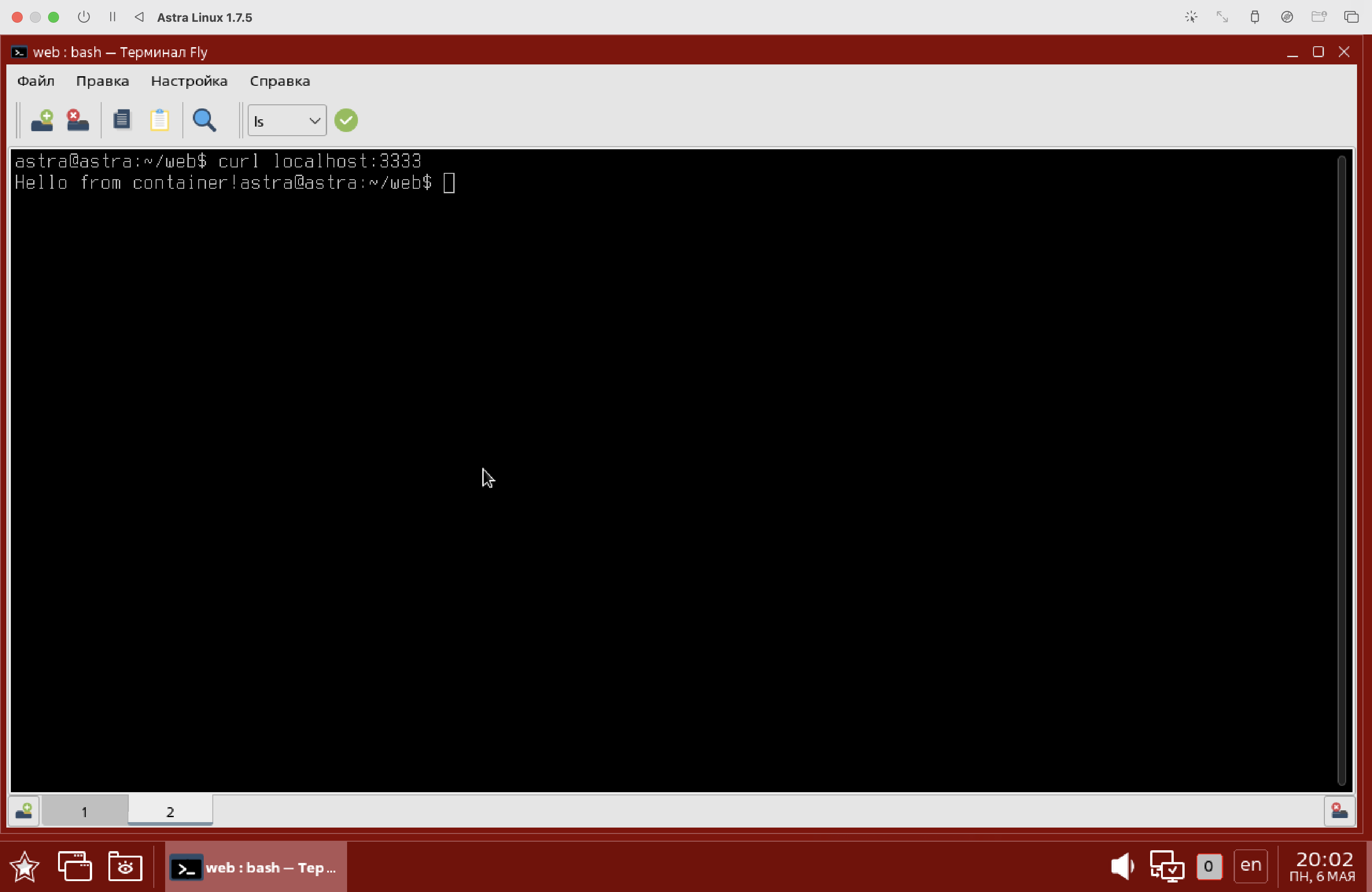


Рисунок 23 – Взаимодействие с контейнером с хоста

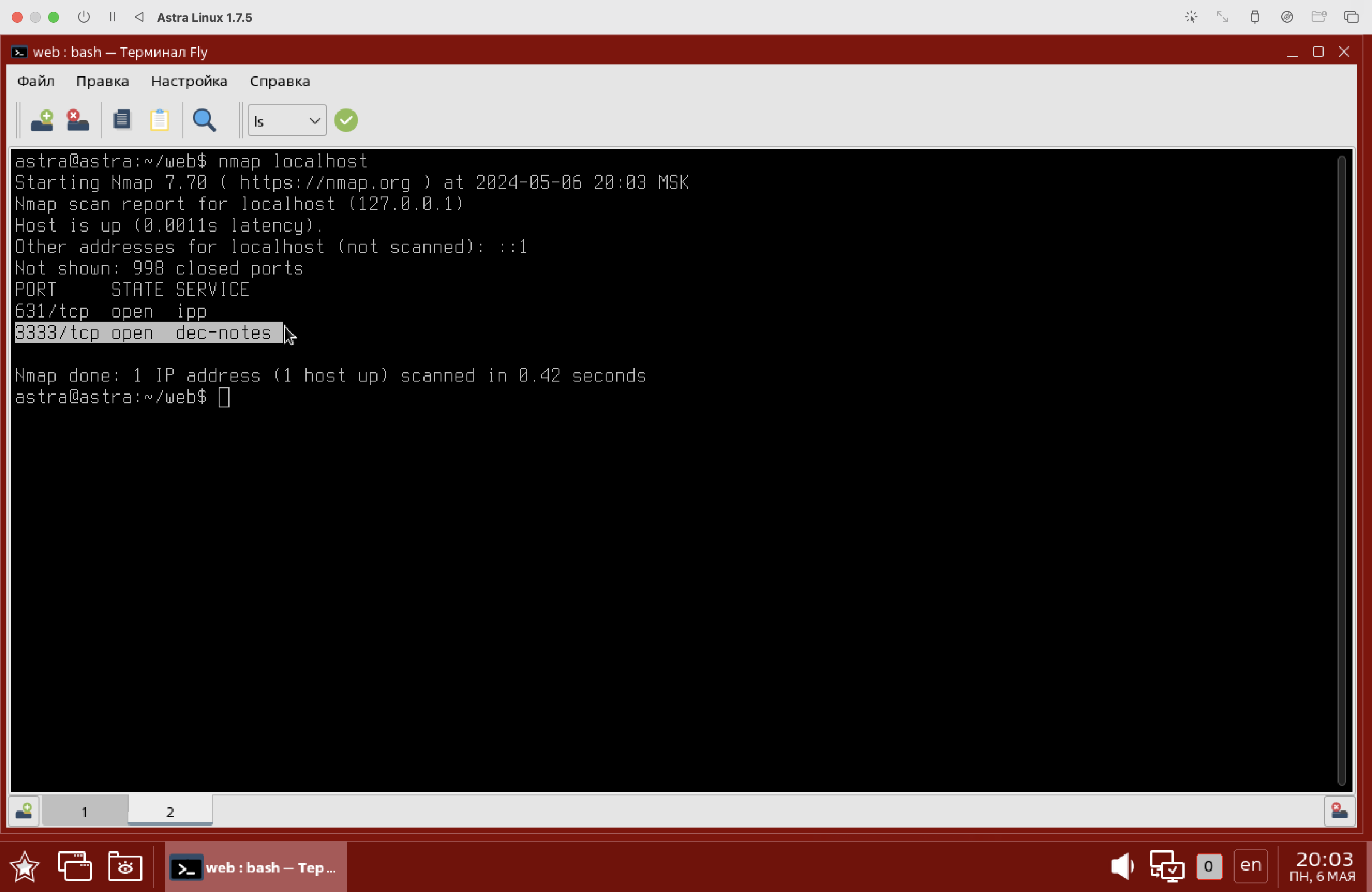


Рисунок 24 – Сканирование портов хоста

Доступ к серверу можно также получить с использованием веб-браузера на хосте (рисунок 25):

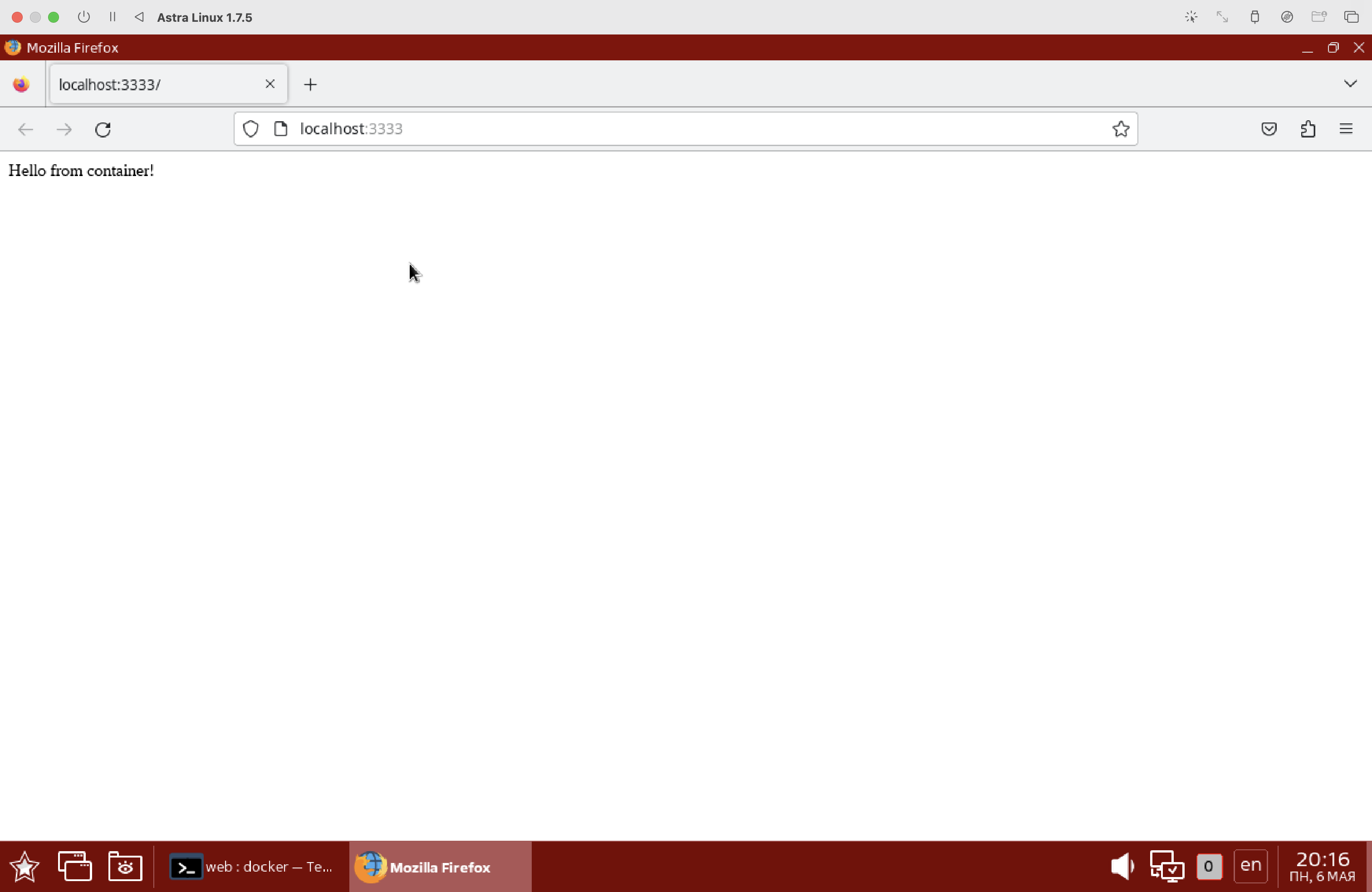


Рисунок 25 – Получение доступа к веб-серверу через браузер

При этом если мы посмотрим изнутри контейнера, какие процессы в нем запущены, то увидим следующий перечень (рисунок 26):

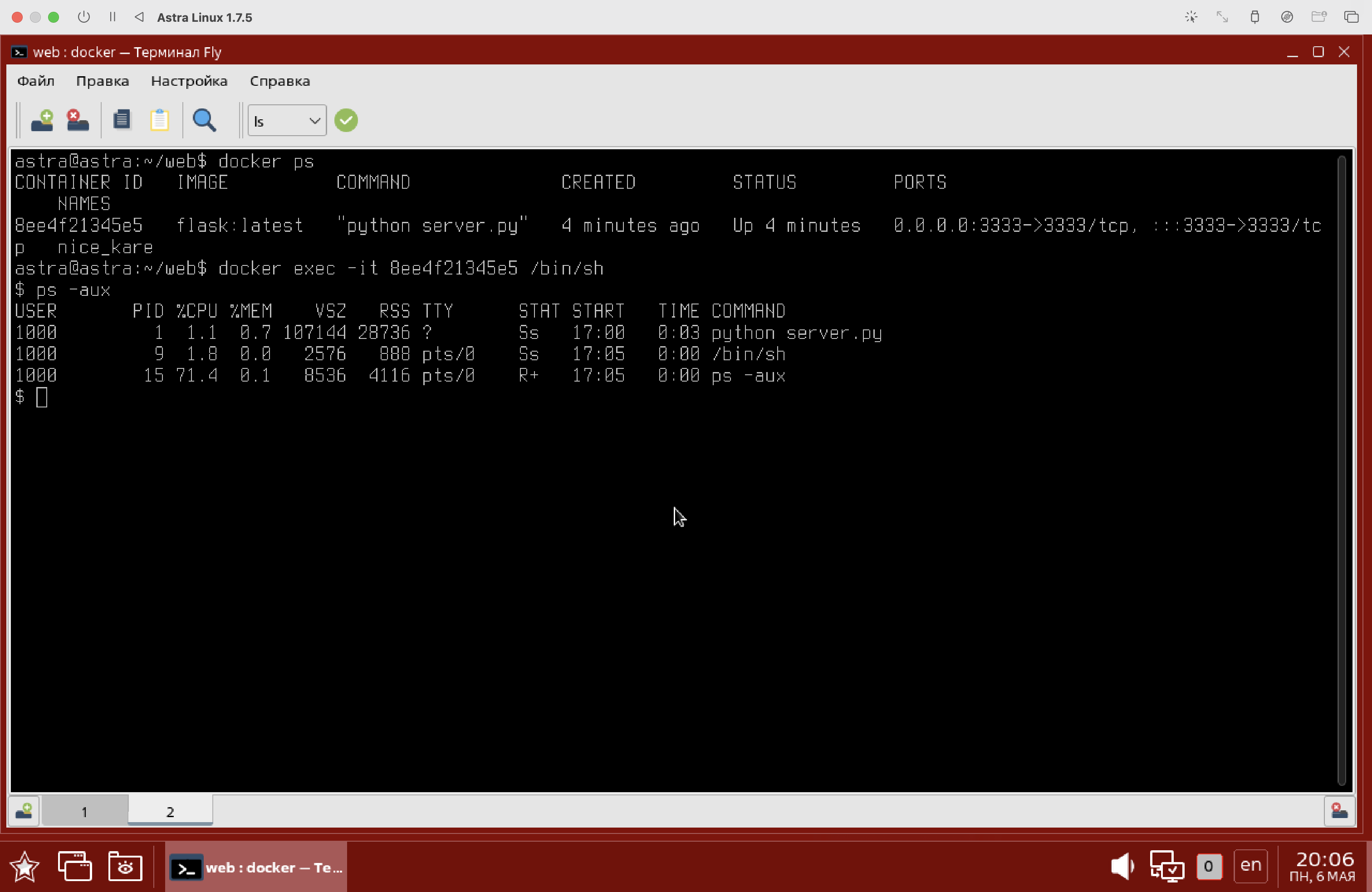


Рисунок 26 – Процессы изнутри контейнера

Текущие процессы хоста имеют вид (рисунок 27):

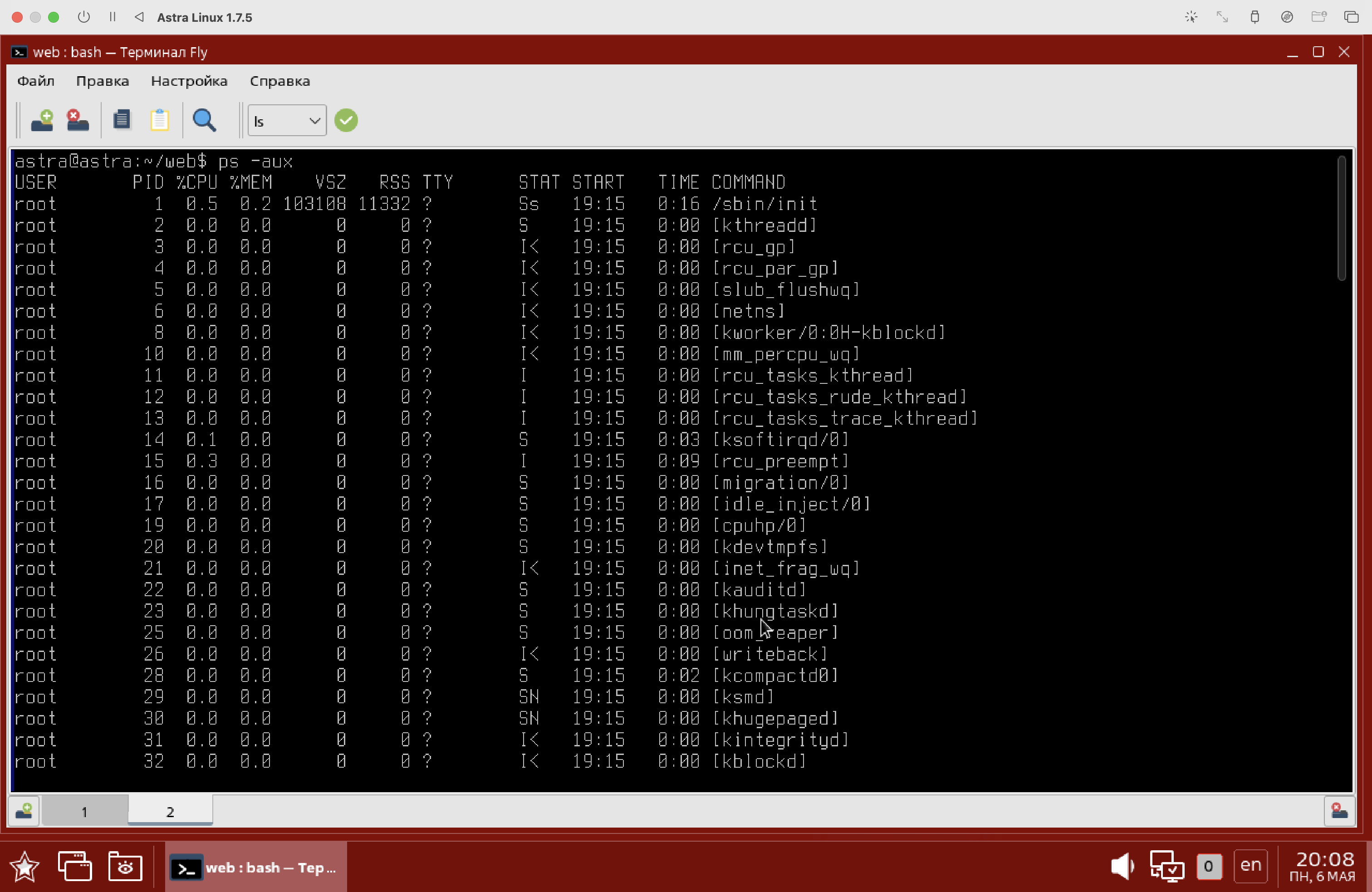


Рисунок 27 – Текущие процессы хоста

Также среди процессов хоста можно найти и процесс docker, и процесс самого сервера python server.py (рисунок 28):

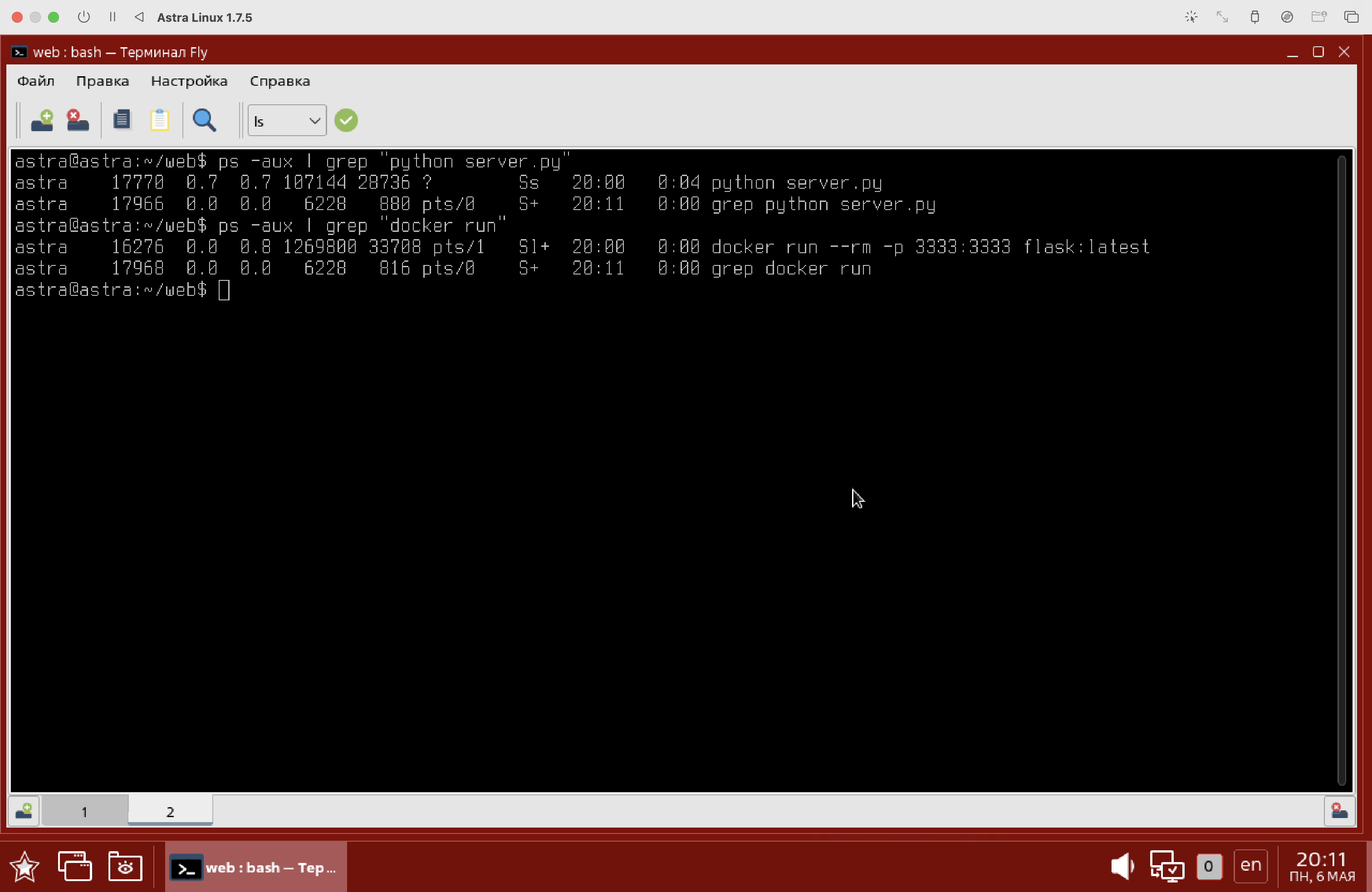


Рисунок 28 – Видимость процессов контейнера на хосте

**Выводы.**

В ходе выполнение лабораторной работы была выполнена установка docker на ОССН Astra Linux, после чего был собран образ с веб-приложением, написанном на фреймворке flask. Из собранного образа был запущен контейнер, на примере которого были показаны механизмы изоляции сети и процессов контейнера.