2. Создание собственных образов

Каждому образу Docker соответствует файл, который называется Dockerfile. Он представляет собой что-то вроде инструкции по сборке самого приложения в контейнер. Так как контейнер состоит из слоев, то именно в Dockerfile они создаются через инструкции FROM, RUN, COPY, и ADD. Остальные же больше относятся к настройке, описанию, действию.

Сам файл может быть простым (состоять из парочки строк), а может быть и сильно сложным.

FROM — представляет собой что-то вроде фундамента в доме (где каждый этаж является слоем). Его принято называть базовым образом. Все последующие команды как бы накладываются на него, тем самым изменяя в необходимый вид. Вообще, сам образ и его теги можно посмотреть на docker hub (но туда можно зайти сейчас только по vpn, так как docker официально заблокирован в нашей стране)

LABEL — описывает метаданные. Например — сведения о том, кто создал и поддерживает образ. По сути, вообще не обязательно использовать, только если хочется сохранить какие-то авторские права на образ.

ENV — устанавливает постоянные переменные среды. Другими словами, просто переменные окружения передаются. Например, вы храните там логин/пароль, константы и т.п. Например: `ENV ADMIN="user"`

RUN — Начинается создаваться новый слой. Обычно данная инструкция служит для скачивания/обновления пакетов

COPY — копирует в контейнер файлы и папки. Мы же в итоге должны будем запустить код в самом контейнере (изолированном). Поэтому необходимо его скопировать из нашего пути, где мы с ним работали, в контейнер. Обычно используется `COPY . ./app` (скопировать всё, находящееся в данной директории, в контейнер в папку `./app`)

ADD — тоже самое, что и copy, только позволяет распаковывать локальные .tar-файлы и добавлять файлы из удаленных источников (например с какого-то сайта)

CMD — Финальная часть Dockerfile, которая запускает весь процесс. Т.е если вы обычно запускаете свою программу в терминале (например, `python3 ./script.py`), то здесь тоже самое. Вы выполняете тоже самое, но уже в контейнере. В файле может присутствовать лишь одна инструкция CMD.

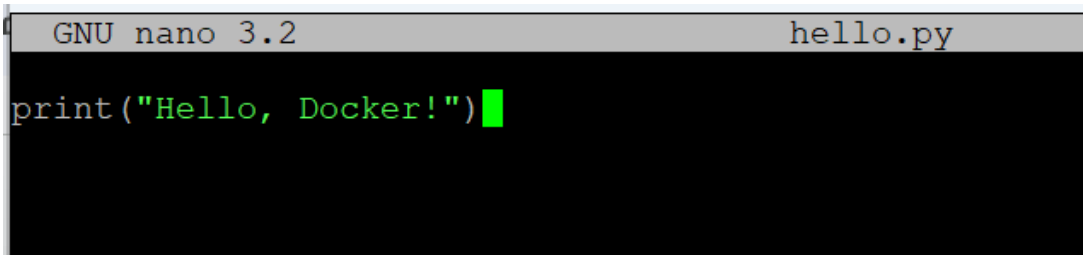
WORKDIR — задаёт рабочую директорию для следующей инструкции, то есть задает папку, в которой мы будем по умолчанию работать

ARG — задаёт переменные для передачи Docker во время сборки образа.

Пример 1.

Создадим простой код на python, выводящий `Hello world`. После чего упакуем его в контейнер.

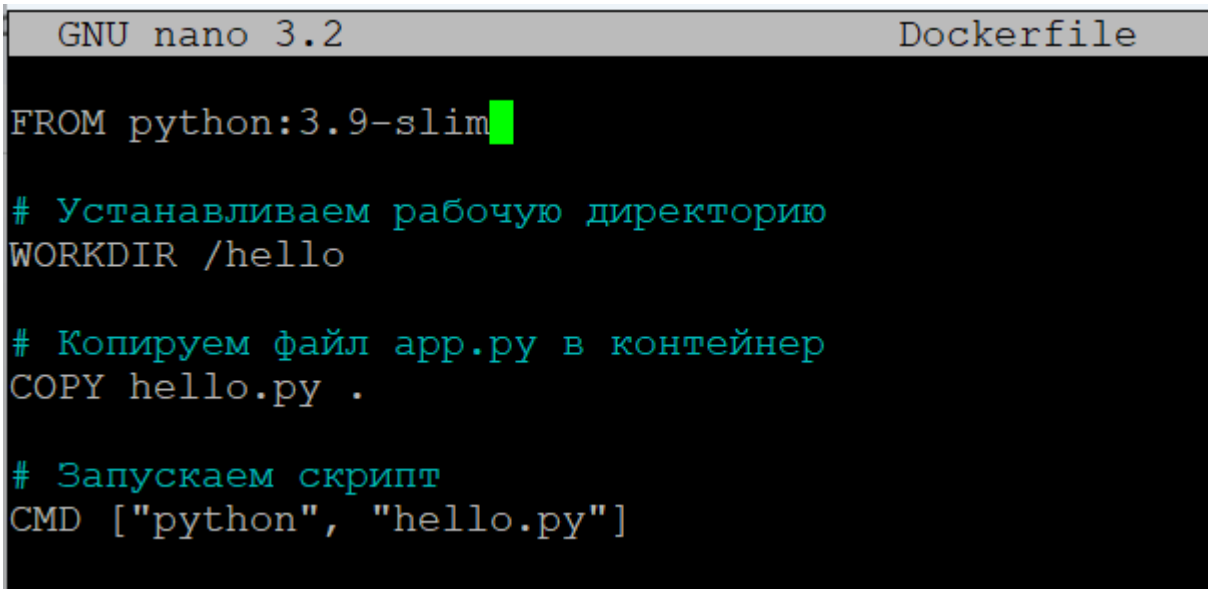
Создадим файл `hello.py` (командой `nano hello.py`) со след кодом:



Проверим вывод след командой

```
cm@template:~/python_test$ python hello.py
Hello, Docker!
```

Все работает, все отлично. Создадим `Dockerfile`, упаковывающий код из `hello.py`. Первая строчка - создает базовый образ `python` (вытекает хотя бы из того, что мы код создавали на нем). `Workdir` создал папку `hello`, в которую все будет впоследствии записываться. В данном случае копируется код hello.py в текущую директорию (об этом говорит точка в конце `copy`). После чего идет простой запуск скрипта, который запускает код (Аналогично проверке на прошлой картинке)



`Dockerfile` собран, осталось его только собрать/упаковать. След команда упаковывает образ под названием `my-hello` из `dockerfile` в данной директории (об этом говорит точка в конце). ВАЖНО: в одной папке может находиться только один `Dockerfile`

```
cm@template:~/python_test$ docker build -t my-hello .
[+] Building 10.3s (8/8) FINISHED                                docker:default
 => [internal] load build definition from Dockerfile                       0.0s
 => => transferring dockerfile: 277B                                       0.0s
 => [internal] load metadata for docker.io/library/python:3.9-slim         1.8s
 => [internal] load .dockerignore                                          0.0s
 => => transferring context: 2B                                            0.0s
 => [1/3] FROM docker.io/library/python:3.9-slim@sha256:2851c06da1fdc3c45  7.5s
 => => resolve docker.io/library/python:3.9-slim@sha256:2851c06da1fdc3c45  0.0s
```

Проверим наличие образа, после чего запустим его. В результате все работает.

```
cm@template:~/python_test$ docker images
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
my-hello            latest    f2db0d98a963   2 minutes ago    125MB
```

```
cm@template:~/python_test$ docker run my-hello
Hello, Docker!
```

Пример 2. Нужно поработать с веб-приложением. Например, создадим приложение во Flask и развернем его.

Создадим папку со след файлами

```
cm@template:~/python_test$ ls
app.py  Dockerfile  requirements.txt
```

1. Введем в терминале `nano app.py` и напишем след. Код

```
  GNU nano 3.2                                    app.py

from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Welcome to my simple Flask app!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

2. Так как контейнер по сути создается пустой, туда необходимо закачать все библиотеки, которые используются в проекте. В данном случае, только `flask`. Хорошей практикой является создание requirements.txt, в котором прописываются все названия библиотек. Почему так? Потому что библиотек может быть много, а каждую скачивать в контейнер по одной - очень долго и муторно. Поэтому легче сразу передать файл с названиями библиотек, а `Dockerfile` сам все закачает.

Вводим в терминале `nano Dockerfile` и пишем след код. Все аналогично, просто в контейнер копируется файл requirements.txt и далее запускается RUN, скачивающий библиотеки (можно ввести просто `run pip install -r requirements.txt`). `-r` просто влаг для указания текстового файла. `–no-cache-dir` - без использования кэша. `Expose` выделяет `5000 порт`

```
  GNU nano 3.2                                  Dockerfile

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .

EXPOSE 5000

CMD ["python", "app.py"]
```

Собираем образ по данному `Dockerfile`

```
cm@template:~/python_test$ docker build -t flask-app .
[+] Building 12.7s (10/10) FINISHED                                 docker:default
 => [internal] load build definition from Dockerfile                          0.0s
 => => transferring dockerfile: 206B                                          0.0s
 => [internal] load metadata for docker.io/library/python:3.9-slim            1.1s
 => [internal] load .dockerignore                                             0.0s
```

Запускаем контейнер в фоновом режиме (флаг `-d`) с перенаправлением портов на `8001` (к примеру) с названием `flask_container` по образу `fask-app.`

```
cm@template:~/python_test$ docker run -d -p 8001:5000 --name flask_container flask-app
44fa04e407ed28d3aaf9470e53249dd80eabcb1537ac023f4c562e98677188f3
```

Проверяем в браузере `localhost:8001` и получаем: