Деплой приложения на сервер

1. 1 шаг. Собираем образ

   ```
   docker build -t aboba-app:0.1 .
   ```

2. 2 шаг. Сохраняем образ в файл

   ```
   docker save aboba-app:0.1 > aboba-app.tar
   ```

3. 3 шаг. Загружаем файл образа на сервер

   ```
   scp ./aboba-app.tar user@server:/path/to/destination
   ```

4. 4 шаг. Подключаемся к серверу

   ```
   ssh user@server
   ```

5. 5 шаг. Загружаем образ из файла

   ```
   docker load < aboba-app.tar
   ```

6. 6 шаг. Копируем docker-compose (прописав в него название образа, загруженного из файла). И запускаемся:

   ```
   nvim docker-compose.yaml
   docker compose up -d
   ```

7. 7 шаг. Поздравляем! Мы запустились! Теперь можем проверить, что всё работает, с компьютера клиента

   ```
   curl http://server:port/products
   ```

Эксперименты над Dockerfile'ом сервиса на Go

1. Для начала втупую скопируем все файлы проекта, после чего запустим процесс компиляции.

   ```
   FROM golang:1.23-alpine as builder

   WORKDIR /app
   COPY . .
   RUN go build -o ./main main.go

   FROM alpine:3.20
   COPY --from=builder /app/main /app/main

   ENTRYPOINT ["/app/main"]
   ```

2. Попробуем собрать наш образ

   ```
   docker compose build app
   ```

3. Всё хорошо, но давайте теперь модифицируем наш код

   ```
   nvim main.go
   ```

4. Попробуем собрать наш образ ещё раз

   ```
   docker compose build app
   ```

   Как мы можем заметить, наши зависимости начали скачиваться заново.

5. Теперь напишем наш Dockerfile по-другому

```
FROM golang:1.23-alpine as builder

WORKDIR /app
COPY go.mod go.sum .
RUN go mod download

COPY . .
RUN go build -o ./main main.go

FROM alpine:3.20
COPY --from=builder /app/main /app/main

ENTRYPOINT ["/app/main"]
```

6. Соберём

   ```
   docker compose build app
   ```

7. Попробуем снова модифицировать наш код

   ```
   nvim main.go
   ```

8. И опять соберём образ

   ```
   docker compose build app
   ```

   Теперь мы не скачиваем зависимости заново

Разница между shell- и exec-режимами

Если мы взглянем на Dockerfile питоновского проекта из первой части, то мы увидим очень интересную конструкцию `CMD`, где каждое слово в команде пишется в кавычках, а между ними ставится запятая.

```
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Но разве нельзя просто записать команду строкой? На самом деле можно. Давайте так и сделаем.

```
CMD uvicorn main:app --host 0.0.0.0 --port 8000
```

Выглядит лаконично, но есть нюанс.

Давайте для простоты сделаем специальный Dockerfile, на котором мы посмотрим разницу между shell- и exec-режимами.

```
FROM alpine:3.20
CMD ["ping", "ya.ru"]
```

Запустим контейнер и выполним команду `ps` внутри него:

```
docker build -t aboba:1.0 .
docker run aboba:1.0
docker ps  # Смотрим ID контейнера
docker exec <ID-контейнера> ps
```

```
PID   USER     TIME  COMMAND
    1 root      0:00 ping ya.ru
    6 root      0:00 ps
```

Мы наблюдаем 2 процесса. Один процесс – это команда `ps`. Он тут есть в целом по понятным причинам. А вот другой процесс – это команда `ping`, которую мы прописали в Dockerfile.

Поскольку `ps` обычно отрабатывает и завершает свою работу, фактически в нашем контейнере работает только один процесс – `ping`. Более того, он имеет PID = 1. Этот факт нам понадобится дальше, когда мы перепишем Dockerfile в shell-режиме:

```
FROM alpine:3.20
CMD ping ya.ru
```

Давайте теперь соберём и запустим наш контейнер:

```
docker build -t aboba:2.0 .
docker run aboba:2.0
docker ps  # Смотрим ID контейнера
docker exec <ID-контейнера> ps
```

И получим... Тоже самое?

```
PID   USER     TIME  COMMAND
    1 root      0:00 ping ya.ru
    7 root      0:00 ps
```

Окей. А тогда в чём же разница? Давайте попробуем заменить Alpine на Debian:

```
FROM debian:12.9
RUN apt-get update -y
RUN apt-get install -y iputils-ping
RUN apt-get install -y procps
CMD ping ya.ru
```

Собираем и запускаем:

```
docker build -t aboba:2.0 .
docker run aboba:2.0
docker ps  # Смотрим ID контейнера

# Введём флаг -ef, чтобы видеть ID родительнского процесса (PPID)
docker exec <ID-контейнера> ps -ef
```

А вот тут уже есть какие-то различия в списке процессов:

```
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 05:28 ?        00:00:00 /bin/sh -c ping ya.ru
root           7       1  0 05:28 ?        00:00:00 ping ya.ru
root          20       0 75 05:29 ?        00:00:00 ps -ef
```

Что мы видим?

1. Процессом с PID = 1 является `/bin/sh`, а не `ping`.
2. `ping` имеет PID равный 7.
3. Кроме того, его PPID равен 1, а это значит, что `/bin/sh` является родительским процессом для `ping`.

Что же будет, если мы попробуем остановить контейнер, послав сигнал `SIGINT` при помощи Ctrl+C?

1. Контейнер, созданный из образа `aboba` будет завершён.
2. Контейнер, созданный из образа `aboba2` аналогично.
3. А вот `aboba3` будет игнорировать наши попытки его завершить (именно так и начинается Skynet).

Чтобы понять, в чём разница, мы взглянем на вывод команды `docker inspect aboba` и `docker inspect aboba3`. Эти команда нам распечатают JSON, в котором содержится

метаинформация про наши образы. Там много любопытной информации, проливающей свет на то, как Docker устроен, но нас интересуют конкретные несколько строк:

1. `docker inspect aboba`

```
[
  {
    ...
    "Config": {
      ...
      "Cmd": ["ping", "ya.ru"],
      ...
    }
    ...
  }
]
```

2. `docker inspect aboba3`

```
[
  {
    ...
    "Config": {
      ...
      "Cmd": ["/bin/sh", "-c", "ping ya.ru"],
      ...
    }
    ...
  }
]
```

Как мы можем наблюдать, у нас по-разному запускается наш `ping`. В первом случае он запускается напрямую. Во втором же случае он запускается через `/bin/sh`. Собственно поэтому он и является родительским процессом для `ping`. И именно поэтому сигналы до процесса `ping` не доходят, ведь в Docker'е сигналы, посланные контейнеру, всегда идут до процесса с PID = 1, которым в `aboba3` является `/bin/sh`.

Но что же с `aboba2`? Давайте тоже для него запустим `docker inspect aboba2`:

```
[
  {
    ...
    "Config": {
      ...
      "Cmd": ["/bin/sh", "-c", "ping ya.ru"],
      ...
    }
    ...
  }
]
```

И мы получаем то же самое... Но почему же мы получаем то же поведение, что и у `aboba`? Я задался таким же вопросом, когда готовился к этой лекции. Для изучения этой темы я решил воспользоваться статьёй на Хабер за 2017 год: https://habr.com/ru/companies/slurm/articles/329138/

Сама по себе статья хорошая, однако, она оказалось немного неактуальной для новых версий Alpine. Дело в том, что Alpine вместо стандартного пакета GNU Coreutils использует BusyBox. При чём, видимо модифицированный, поскольку в других дистрибутивах, где используется

BusyBox, поведение `sh` было больше похоже на образ `aboba3`. Скорее всего, разработчики Alpine, нацеленные на пользователей Docker, решили модифицировать оболочку командной строки, чтобы она не имела тех багов, которые возникают с `aboba3`.

Тем не менее, несмотря на то, что в Alpine shell-форма не имеет тех багов, которые есть в Debian, всё же разработчики Docker рекомендуют использовать exec-форму.

Разница между CMD и ENTRYPOINT

1. `CMD` определяет команду, которая будет выполнена при запуске, контейнера.

   ```
   FROM alpine:3.20
   CMD ["echo", "Hello, World!"]
   ```

   ```
   $ docker build -t hello-world-image:1.0 .
   $ docker run hello-world-image:1.0
   Hello, world!
   ```

   При этом мы можем спокойно переопределить команду, которая будет выполнена, при запуске контейнера:

   ```
   $ docker run hello-world-image:1.0 echo "Aboba"
   Aboba
   ```

2. `ENTRYPOINT` определяет команду, которая будет выполнена при запуске контейнера.

   ```
   FROM alpine:3.20
   ENTRYPOINT ["echo", "Hello, World!"]
   ```

   ```
   $ docker build -t hello-world-image:2.0 .
   $ docker run hello-world-image:2.0
   Hello, world!
   ```

   Казалось бы, то же самое. Однако различия появляются, когда мы добавим аргументы:

   ```
   $ docker run hello-world-image:2.0 echo "Aboba"
   Hello, World! echo Aboba
   ```

   Как мы видим, при использовании `ENTRYPOINT` переопределяется не вся команда, а только её аргументы. Это может быть удобно, если ваша программа принимает какие-либо аргументы.

   При этом мы всё ещё можем переопределить команду, которая будет выполнена при запуске контейнера, используя флаг `--entrypoint`:

   ```
   docker run --entrypoint ps hello-world-image:2.0
   ```

   ```
   PID   USER     TIME  COMMAND
       1 root      0:00 ps
   ```

3. `ENTRYPOINT` + `CMD`

   Мы можем использовать `CMD` для указания аргументов по умолчанию, которые будут переданы в `ENTRYPOINT`:

   ```
   FROM alpine:3.20
   ENTRYPOINT ["echo"]
   CMD ["Hello, world!"]
   ```

   В таком случае по-умолчанию команде `echo` в качестве аргумента будет передаваться строка `Hello, world!`. Однако, если мы укажем другой аргумент, он заменить аргумент, прописанный `CMD`.

```
$ docker build -t hello-world-image:3.0 .
$ docker run hello-world-image:3.0
Hello, world!
$ docker run hello-world-image:3.0 Aboba
Aboba
```

Конфигурирование приложений в Docker при помощи переменных окружения

Одним из самых частых способов для конфигурации приложения является использование переменных окружения. Давайте напишем небольшой проект на Python, который будет считывать переменные окружения `DB_URL` и `API_KEY` выводить их на экран. Для управления зависимостями мы будем использовать `uv`.

```
uv init example1
cd example1
uv add pydantic pydantic-settings python-dotenv
```

```
from pydantic_settings import BaseSettings, SettingsConfigDict
from time import sleep


class Settings(BaseSettings):
    db_url: str
    api_key: str

    model_config = SettingsConfigDict(env_file=".env", env_file_encoding="utf-8")


def main():
    settings = Settings()
    print(settings, flush=True)

    while True:
        sleep(1)


if __name__ == "__main__":
    main()
```

Здесь мы использовали библиотеки `pydantic` и `python-dotenv`. Первая позволяет производить валидацию нашей конфигурации, а вторая позволяет записывать значения переменных окружения из файла `.env`. Как правило, это делается для удобства разработчика, чтобы ему не пришлось постоянно вводить эти переменные в терминале. Впрочем, если `.env` файла нет, `python-dotenv` будет считывать значения, что называется, “по-старинке” из непосредственно переменных окружения. Этот факт нам пригодится позже.

Напишем Dockerfile для нашего приложения (данный файл можно улучшить, но это мы рассмотрим позже):

```
FROM python:3.13-slim AS builder
COPY --from=ghcr.io/astral-sh/uv:latest /uv /uvx /bin/
ADD . /app
WORKDIR /app
RUN ["uv", "sync", "--frozen"]
CMD ["uv", "run", "main.py"]
```

Теперь напишем .env файл:

```
DB_URL=aboba
API_KEY=aboba
```

Соберём и запустим наш образ:

```
docker build -t config-example:1.0 .
docker run config-example:1.0
```

Что ж, наше приложение работает!

```
db_url='aboba' api_key='aboba'
```

Однако есть несколько проблем:

1. Мы в образ копируем директорию .venv, которая создана на хостовой машине. Это проблема по двум причинам:
    1. Операционная система на хостовой машине и в контейнере могут отличаться. А поскольку помимо зависимостей на Python у нас могут быть и бинарные зависимости, это может привести к проблемам. (Впрочем, если мы посмотрим внимательно на логи, `uv` автоматически удаляет случайно скопированное нами виртуальное окружение и создаёт своё. Но так происходит не всегда, особенно если не использовать `uv`).
    2. `.venv` может весить очень много (например, в больших проектах или проектах, использующих ML-библиотеки) => копирование будет происходить долго. Мы просто тратим время на операции, которые нам не нужны. В любом случае, нам нужно создавать своё виртуальное окружение для образа.
2. Также мы копируем `.env` файл в наш образ. Это уже проблема в безопасности. В образе не должны храниться секреты, по типу паролей, API-ключей и прочего, так как в случае утечки образа (или если ваш образ общедоступный) ваши секреты будут скомпрометированы, не говоря уже о том, что мы не сможем по-разному конфигурировать контейнеры, запускаемые из этого образа.
3. Мы копируем кэш-файлы и папки, по типу `__pycache__`. Как и `.venv`, кэши являются платформозависимыми и не должны быть включены в образ по тем же причинам.
4. Если вы используете Git и директория `.git` находится в корне проекта, то она тоже будет копироваться, увеличивая размер образа, хотя для сборки образа она как правило не нужна.

Мы бы могли модифицировать команду `ADD` в Dockerfile, например, используя регулярные выражения или вручную прописав все необходимые для копирования файлы. Но есть более удобный способ исключить из копирования ненужные файлы – `.dockerignore` файл. Он работает аналогично тому, как работает `.gitignore`. В `.dockerignore` файле можно указать паттерны файлов и директорий, которые не должны быть скопированы в образ.
Создадим `.dockerignore` файл в корне проекта:

```
.gitignore
.git
.venv
__pycache__
.env
```

Давайте снова соберём и запустим наш образ:

```
docker build -t config-example:2.0 .
docker run config-example:2.0
```

И теперь наше приложение не работает:

```
...
pydantic_core._pydantic_core.ValidationError: 2 validation errors for Settings
db_url
  Field required [type=missing, input_value={}, input_type=dict]
    For further information visit https://errors.pydantic.dev/2.10/v/missing
api_key
  Field required [type=missing, input_value={}, input_type=dict]
    For further information visit https://errors.pydantic.dev/2.10/v/missing
```

Очевидно, это происходит, потому что `.env` файл не был скопирован в образ. Как вы помните, если `python-dotenv` не находит `.env` файл, то он просто считывает параметры из переменных окружения. Но как установить переменные окружения при запуске образа? Мы бы могли использовать команды `Dockerfile`:

```
ENV DB_URL=aboba
ENV API_KEY=aboba
```

Но тогда мы не решаем проблему с тем, что в случае утечки образа наши секреты будут скомпрометированы. Вместо этого мы воспользуемся флагом `-e` при запуске образа:

```
docker run -e DB_URL=aboba -e API_KEY=aboba config-example:2.0
```

Теперь наше приложение работает!

```
db_url='aboba' api_key='aboba'
```

Docker compose

Давайте теперь посмотрим, как передавать конфигурацию в контейнер при использовании docker compose:

```
services:
  app:
    image: my-app:latest
    environment:
      DB_URL: aboba
      API_KEY: aboba
```

Запустим наше приложение и посмотрим, как оно работает:

```
docker compose up
```

И... Всё работает!

```
...
[+] Running 3/3
 ✔ app                       Built
0.0s
 ✔ Network example1_default  Created
0.2s
 ✔ Container example1-app-1  Created
0.1s
Attaching to app-1
app-1  | db_url='aboba' api_key='aboba'
```

Если ваш docker compose файл хранится в репозитории вместе с проектом, в нём тоже очень нежелательно хранить секреты. Благо, docker compose также позволяет прописывать переменные окружения при запуске команды:

```
services:
  app:
```

```
    build: .
    environment:
      DB_URL: ${DB_URL}
      API_KEY: ${API_KEY}
```

```
DB_URL=aboba API_KEY=pupupu docker compose up
```

```
[+] Running 1/1
 ✔ Container example1-app-1  Created
0.0s
Attaching to app-1
app-1  | db_url='aboba' api_key='pupupu'
```

Если же вы попробуете запустить приложение с указанием переменных окружения, но используя первый docker compose файл, то ничего не поменяется.

Помимо того, что мы можем напрямую прописать переменные окружения, docker compose может считывать их из файла `.env`.

Модифицируем `.env` файл:

```
DB_URL=csit
API_KEY=sgu
```

Потом напишем новый docker compose файл:

```
services:
  app:
    build: .
    env_file: .env
```

Запустим приложение:

```
docker compose up
```

```
[+] Running 1/1
 ✔ Container example1-app-1  Recreated
0.1s
Attaching to app-1
app-1  | db_url='csit' api_key='sgu'
```

Важно! Из .env файла считывает переменные окружения не наш проект, а docker compose. После считывания он их записывает в переменные окружения контейнера, из которых параметры уже берёт наше приложение. Никакого .env файла (если вы прописали .dockerignore файл по инструкции ниже, в образе или контейнере нет)

```
$ docker ps  # Смотрим ID контейнера
$ docker exec <ID контейнера> ls -a
..
.dockerignore
.python-version
.venv
Dockerfile
README.md
docker-compose.yaml
main.py
pyproject.toml
uv.lock
```

Кстати, в наш .gitignore можно добавить README.md, Dockerfile и docker-compose.yaml. Для функционирования приложения внутри контейнера они не нужны. При этом директория .venv

была создана uv при сборке образа, поэтому она тут есть, несмотря на то, что она прописана в .dockerignore.

Volumes

Давайте вспомним наш “слоённый пирог”. Наш контейнер формируется из образа, который в свою очередь формируется из нескольких слоёв. По итогу, самым верхним слоём является сам контейнер. Это единственный слой, который мы можем изменять в процессе выполнения. Соответственно, когда приложение работает, все изменения в файловой структуре, которое оно делает, сохраняются в этом слое.

Однако, у такое подхода есть несколько проблем:

1. **Производительность**. Запись в файловую систему контейнера происходит медленнее, чем в файловую систему хостовой машины.
2. **Сохранение данных**. Все изменения, которые делает приложение, сохраняются только в контейнере. Когда контейнер удаляется, все данные теряются.
3. **Совместное использование**. Вы не сможете запустить несколько контейнеров из одного образа, чтобы они могли использовать общие данные.

Очевидно, что как минимум мы не хотим терять данные в случае удаления контейнера.

Для решения этих проблем мы можем использовать **volumes**.

Пример использования

Рассмотрим использование **volumes** на примере приложения на Python + FastAPI, который сохраняет данные в JSON-файл (мы воспользуемся именно JSON-файлом вместо БД для простоты и наглядности).

```
from fastapi import FastAPI
from pydantic import BaseModel
import json
import os

app = FastAPI()

DATA_FILE = "/data/people.json"


def read_people():
    if os.path.exists(DATA_FILE):
        with open(DATA_FILE, "r") as f:
            return json.load(f)
    return []


def write_people(people):
    with open(DATA_FILE, "w") as f:
        json.dump(people, f)


class Person(BaseModel):
    name: str
    age: int


@app.get("/people")
```

```
def get_people():
    people = read_people()
    return people


@app.post("/people")
def add_person(person: Person):
    people = read_people()
    people.append(person.dict())
    write_people(people)
    return {"message": "Person added successfully"}
```

Теперь напишем Dockerfile для этого приложения:

```
FROM python:3.13-slim AS builder
COPY --from=ghcr.io/astral-sh/uv:latest /uv /uvx /bin/

ADD . /app
WORKDIR /app

RUN ["uv", "sync", "--frozen"]
RUN ["mkdir", "/data"]
RUN ["echo", "[]", ">>", "/data/people.json"]

ENTRYPOINT ["uv", "run", "uvicorn", "main:app"]
CMD ["--host", "0.0.0.0", "--port", "8000"]
```

Соберём и запустим:

```
docker build -t volumes-example:1.0 .
docker run -p 8000:8000 volumes-example:1.0
```

Вы можете протестировать приложение, используя Swagger UI, доступный по адресу `localhost:8000/docs`.

Теперь попробуем завершить наше приложение и запустить его заново. В таком случае мы обнаружим, что наши данные не сохранились.

Давайте теперь воспользуемся volume для сохранения данных. Но для начала его необходимо создать:

```
docker volume create some-volume
```

Теперь давайте взглянем, где Docker расположил наш volume. Для этого воспользуемся командой `docker volume inspect`:

```
docker volume inspect some-volume
```

```
[
    {
        "CreatedAt": "2025-03-01T08:24:53+04:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/some-volume/_data",
        "Name": "some-volume",
        "Options": null,
        "Scope": "local"
    }
]
```

Как мы можем наблюдать, Docker создал volume в директории `/var/lib/docker/volumes/some-volume/_data`.

Если мы взглянем на содержимое этого каталога, то на данный момент его содержимое пусто. Давайте это недоразумение исправим: запустим наш контейнер с этим volume. Но для начала немного поменяем Dockerfile, удалив команды для создания `/data/people.json` файла:

```
FROM python:3.13-slim AS builder
COPY --from=ghcr.io/astral-sh/uv:latest /uv /uvx /bin/

ADD . /app
WORKDIR /app

RUN ["uv", "sync", "--frozen"]

ENTRYPOINT ["uv", "run", "uvicorn", "main:app"]
CMD ["--host", "0.0.0.0", "--port", "8000"]
```

```
docker build -t volumes-example:2.0 .
docker run -p 8000:8000 -v some-volume:/data volumes-example:2.0
```

Теперь также протестируем наше приложение, воспользовавшись Swagger UI (ну или можете вручную запросы покидать, разницы никакой, на Сваггере не настаиваю). После тестирования перезапустим наше приложение и увидим, что данные сохранились.

Если же мы посмотрим на содержимое директории `/var/lib/docker/volumes/some-volume/_data`, то увидим, что там уже есть файл `people.json` с данными. Давайте глянем на его содержимое:

```
$ cat /var/lib/docker/volumes/some-volume/_data/people.json
[{"name": "string", "age": 0}, {"name": "string", "age": 0}, {"name": "string",
"age": 0}]
```