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
```