GO-11 02: Продвинутый образ Docker

Описание:

В предыдущем задании вы познакомились с Docker и контейнерами. Мы собрали образ нашего приложения, и теперь требуется понять, как его запустить. В данном задании мы познакомим вас со способами запуска контейнеров, инструментами docker-compose и с multi-stage сборкой образов. Кроме того, вы увидите несколько best practice по запуску контейнеров.
Способы запуска контейнеров:
Существует несколько способов запуска, давайте разберем самый легкий из них.
У нас уже был образ с пердыдущего задания, давайте соберем его еще раз.

```
docker build .
```

В ответ мы получим следущее сообщение:

```
Sending build context to Docker daemon  3.723MB
Step 1/7 : FROM golang:latest
 ---> 2b88645b406c
Step 2/7 : RUN mkdir /app
 ---> Using cache
 ---> 52231b19269f
Step 3/7 : ADD . /app/
 ---> 40318ed8d603
Step 4/7 : WORKDIR /app
 ---> Running in 1924c0a366a9
Removing intermediate container 1924c0a366a9
 ---> 045b7d9e8c17
Step 5/7 : EXPOSE 7777
 ---> Running in b7ec14b50c2e
Removing intermediate container b7ec14b50c2e
 ---> 2289fdaf05f7
Step 6/7 : RUN go build -o main .
 ---> Running in 09a892079c71
Removing intermediate container 09a892079c71
 ---> dd7fa21ac16d
Step 7/7 : CMD ["/app/main"]
 ---> Running in a61523b3db81
Removing intermediate container a61523b3db81
 ---> 62a187920e4f
```

```
Successfully built 62a187920e4f
```

Таким образом, id нашего образа - это : 62a187920e4f.
Давайте запустим контейнер с помощью командной строки. Чтобы каждый раз не обращаться к образу по id, мы можем использовать tag. Далее следует опциональная команда, все инструкции также применимы и с id, но тег позволит вам не копировать id, а запоминать, какой образ вы уже собрали.

```
docker tag 62a187920e4f rebrainapp
```

```
docker images
```

```
REPOSITORY                                      TAG
IMAGE ID            CREATED             SIZE
rebrainapp                                      latest
62a187920e4f        1 minutes ago       815MB
```

Теперь мы сможем использовать образ по имени.
Запускаем контейнер:

```
docker run -p 7777:7777 --name rebraincontainer rebrainapp
```

После этого контейнер будет запущен. Если вы не напутали ничего с портами, вы можете открыть в браузере адрес вашего приложения и увидеть, что handler отработал верно.
Теперь представим, что наше приложение завершило свою работу критическим образом, то есть просто упало. Контейнер сам по себе не поднимется, нам придется перезапустить его руками. Но можно указать специальный флаг для работы данного контейнера - always.

```
docker run -p 7777:7777 --restart=always --name rebraincontainer
rebrainapp
```

Теперь при возникновении паники контейнер docker перезапустит наш контейнер. Также у флага restart есть и другие значения:

- no - не перезапускать контейнер (устанавливается по умолчанию);
- unless-stopped - перезапускает контейнер всегда, за исключением явной остановки самим пользователем;
- on-failure[:2] - перезапускать в случае сбоя, в скобках указывается значение максимального количества перезапусков.

Посмотреть на запущенные контейнеры можно с помощью команды docker ps.

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS                          PORTS               NAMES
78f8af817429        rebrainapp          "/app/main"         About a
minute ago   Restarting (2) 28 seconds ago
rebraincontainer
```

В коде мы видим пример того, что будет, если запустить контейнер с ошибкой runtime и restart=always. Контейнер будет бесконечно перезапускаться, в скобочках указано количество перезапусков.
Теперь давайте посмотрим на собранный нами образ, он занимает 824MB. Это почти гиг!!! Для сборки и dev запусков это не критично, но это критично для продакшена. Почему образ весит так много? Дело в том, что мы используем образ golang:latest, он тащит за собой все зависимости компилятора + все инструменты Go для разработки. В продакшн версии нам не требуются все эти инструменты. Плюс в наш образ попали исходники нашего приложения, из которого происходила сборка. Как же облегчить наш образ? Убрать все эти зависимости!
Вместо golang:latest теперь мы будем использовать scratch, а вместо исходников - передавать в образ уже собранный нами бинарик.
В этом случаи нам требуется собрать у себя на компьютере бинарный файл под сборку linux amd64 и переписать наш Dockerfile следующим образом:

```
FROM scratch

ADD ./bin/rebrainapp /usr/bin/rebrainapp

ENTRYPOINT [ "/usr/bin/rebrainapp" ]
```

Посмотрим, сколько же теперь весит наш образ. 7.42MB! Это почти в 10 раз меньше!
Docker-compose:
Docker Compose используется для одновременного управления несколькими контейнерами, входящими в состав приложения. Этот инструмент предлагает те же возможности, что и Docker, но позволяет работать с более сложными приложениями. Docker-compose позволяет поднять ваш сервис и рядом все сервисы, от которых он зависит, например, базу данных или брокер сообщений. Если бы нам требовалось поднять несколько сервисов с помощью обычных команд докер, мы бы повторяли несколько действия подряд: создать образ, поднять контейнер. Docker-compose позволят оптимизировать эти шаги. Давайте посмотрим на структуру Docker-compose:

```
version: '3.3'

services:
    db:
       image: mongo:3.4
       volumes:
           - /user/lib/mongo/data:/data/db
       logging:
          driver: "none"

    rebrain-app:
```

```
        build: .
        environment:
          DB_ADDR: db
        entrypoint: /usr/bin/rebrainapp
        ports:
            - 7777:7777
```

Выкладка выше показывает пример, что было бы, если бы наше приложение использовало базу данных. Мы могли бы из одного файла поднять сразу два контейнера, например, один с mongodb, а второй - с нашим приложением.
Команда для запуска:

```
docker-compose up
```

Чтобы команда заработала, требуется, чтобы файл docker-compose.yml лежал в той же папке, что и наш проект, и содержал код, который приведен выше.
Также docker-compose может содержать всего один сервис - наше приложение. Запуск с помощью docker-compose упрощает сборку и поднятие контейнера. Это делается всего в одну команду. Тег build: . указывает, что сборка образа должна быть из Dockerfile, лежащего рядом, то есть докер-файл все равно требуется. ******

```
version: '3.3'

services:
    portal-admin:
        build: .
        entrypoint: /usr/bin/rebrainapp
        ports:
            - 7777:7777
```

Вот так бы выглядел docker-compose.yml только для нашего сервиса.
Multi-Stage Docker Builds:
С Docker 17.05 появилась возможность использовать один единственный Dockerfile для определения последовательности шагов сборки Docker образов! Это значит, что мы можем в одном Dockerfile описать процесс сборки и запуска нашего контейнера. При этом собирать из образа golang, а запускать, например, из scratch.
При многоэтапной сборке вы используете несколько операторов FROM в вашем Dockerfile. Каждая инструкция FROM использует произвольный базовый образ и начинает новый этап сборки. Вы можете выборочно копировать артефакты с одного этапа на другой, оставляя только то, что вам необходимо в конечном образе. Таким образом, мы получим средства сборки из одного образа и легкий образ для запуска. В образ пойдет только последний этап сборки.
Таким образом, наш Dockerfile перепишется следующим образом:

```
FROM golang:latest as builder
```

```
RUN mkdir /app
ADD . /app/
WORKDIR /app
EXPOSE 7777
RUN GOOS=linux GOARCH=amd64 go build -o rebrainapp

#поднимаем бинарник
FROM scratch

COPY --from=builder /app/rebrainapp /usr/bin/rebrainapp

ENTRYPOINT [ "/usr/bin/rebrainapp" ]
```

Полезные ссылки:

- https://docs.docker.com/develop/develop-images/multistage-build/
- https://docs.docker.com/compose/compose-file/

Задание:

1. В вашем проекте module11 сделайте новую ветку module11_02.
2. Измените Dockerfile для запуска скомпилированного приложения через scratch.
3. Напишите docker-compose, который поднимает наш сервис вместе с базой postgresql(или любой другой).
4. В качестве ответа пришлите:

- скриншот выполненной команды docker ps при запущенном через docker-compose проекте;
- ссылку на merge request в ветку master вашего проекта ветки module11_02.