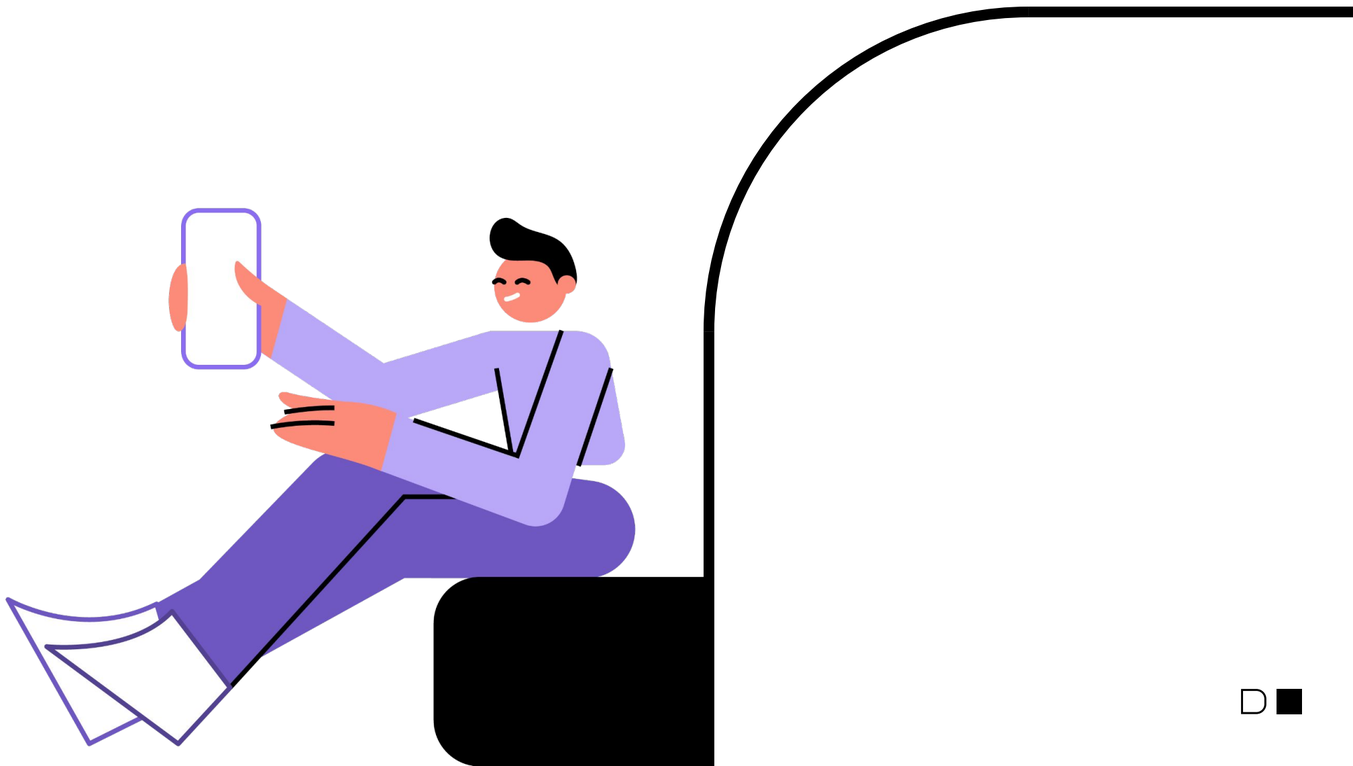
DIGITALHOUSE

Implantação de microsserviços com Docker

Certified Tech Developer
The Ultimate Degree

Índice



01 Dockerfile

Dockerfile

A primeira coisa que precisamos fazer para começar a executar nossos microsserviços de containers são as imagens do Docker. Como sabemos, podemos criá-las utilizando arquivos conhecidos como **"Dockerfile"**. Vamos dar uma olhada em um exemplo de configuração para projetos Java:

```
FROM adoptopenjdk/openjdk11:alpine-jre   01
ARG JAR_FILE=spring-boot-web.jar    02
COPY ${JAR_FILE} app.jar       03
ENTRYPOINT ["java","-jar","app.jar"]     04
EXPOSE 8080    05
```

01 Indicamos que vamos usar Java JDK versão 11 para executar nosso projeto no container. Aqui podemos mudá-la para a versão que usamos para a codificação.

02 Indicamos onde se encontra o arquivo JAR de nosso projeto (normalmente criado com Maven ou Gradle). Neste exemplo, o arquivo é chamado **spring-boot-web**.

03 Copiar o arquivo JAR para o container e nomeá-lo “**app.jar**”.

04 Executamos o arquivo JAR com o comando “**java -jar app.jar**”**.**

05 Expomos a porta 8080 do container. Como cada container funciona em um endereço IP diferente, podemos configurar todos os projetos com a mesma porta.

Construindo as imagens

Agora que criamos o Dockerfile, podemos construir a imagem usando o Docker. Localizado na mesma pasta do Dockerfile, a partir do console, ingressamos:

`docker build --tag java-docker`.

Executando as imagens em containers

Uma vez criada a imagem, podemos executá-la em um container com o comando:

```
docker run -p8080:8080 java-docker
```

É isso aí! Temos nosso container executando nosso microsserviço.

02

Implantação de microsserviços com Docker Compose

Implantando com Docker Compose

Antes de usar o Docker Compose, devemos criar o Dockerfile em cada projeto que queremos usar. Suponha que tenhamos dois microsserviços —**product-service** e **user-service**— localizados dentro de **/microservices**. Para usar o Docker Compose precisamos criar um arquivo de configuração no formato YML. Vejamos um exemplo disso e o que devemos indicar em cada microsserviço:

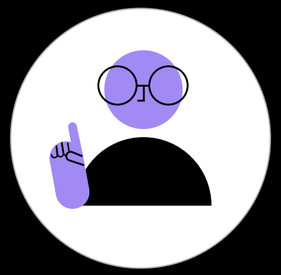
```yaml
version: '2.1'

services:
  product-service:
    build: microservices/product-service
    mem_limit: 512m
    ports:
      - "8080:8080"
  user-service:
    build: microservices/user-service
    mem_limit: 512m
    ports:
      - "8080:8080"
```

Nome do microsserviço que também será usado como hostname na rede interna do Docker.

Indicamos onde está localizado o Dockerfile que será usado para construir a imagem.

O limite de memória é indicado (512mb como exemplo).

Mapeamos a porta do container com o host.



Aqui, mostramos os passos e comandos que serão necessários para executar no arquivo **docker-compose.yml**
Para colocar os microsserviços incluídos no arquivo YML em funcionamento:

01

Criar o arquivo JAR de cada microsserviço:

a) Com Maven: **mvn package**
b) Com gradle: **gradle build**

02

Criar as imagens Docker dos microsserviços com os seguintes comandos: `docker-compose build`

03

Executar os containers com o seguinte comando: `docker-compose up -d`

04

Escalando product-service: `docker-compose up -d --scale product-service=2`
No momento em que precisamos desligar os containers, podemos usar o seguinte comando:
`docker-compose down`

03

Implantando banco de dados com Docker Compose

Implantando bancos de dados com Docker Compose

Vamos supor que o microsserviço **product-service** utiliza o MongoDB como banco de dados. Acrescentamos no arquivo **docker-compose.yml**:

```
mongodb:
    image: mongo:4.4.2
    mem_limit: 512m
    ports:
      - "27017:27017"
    command: mongod
     healthcheck:
      test: "mongo --eval 'db.stats().ok'"
      interval: 5s
      timeout: 2s
      retries: 60
```

Para o exemplo, usamos o MongoDB versão 4.4.2, expomos a porta 27017 e configuramos um *healtcheck* para saber o status do banco de dados em tempo de execução.

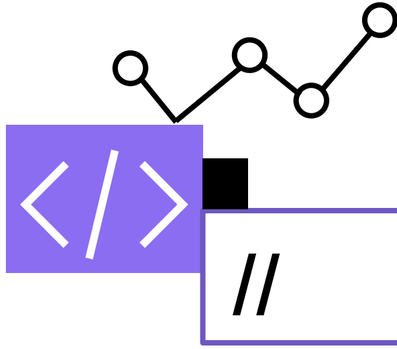
Para evitar problemas de conexão com o microsserviço **product**, acrescentamos:

```
depends_on:
  mongodb:
    condition: service_healthy
```

Desta forma, o container que opera o microsserviço do produto não se executará até que o container esteja funcionando corretamente.

O arquivo **docker-compose.yml** deveria ter este aspecto:

```yaml
version: '2.1'

services:
  product-service:
    build: microservices/product-service
    mem_limit: 512m
    ports:
      - "8080:8080"
    depends_on:
      mongodb:
        condition: service_healthy
  user-service:
    build: microservices/user-service
    mem_limit: 512m
   ports:
      - "8080:8080"
 mongodb:
    image: mongo:4.4.2
    mem_limit: 512m
    ports:
      - "27017:27017"
    command: mongod
     healthcheck:
      test: "mongo --eval 'db.stats().ok'"
      interval: 5s
      timeout: 2s
      retries: 60
```



Finalmente, temos a configuração em **application.properties** de **product service**. Quando desenvolvemos e testamos, geralmente o banco de dados está rodando em localhost, e usando Docker temos que substituir localhost pelo nome que usamos no **docker-compose.yml.** No nosso caso, a conexão com MongoDB ficaria assim:

```
spring.data.mongodb.host: mongodb
spring.data.mongodb.port: 27017
spring.data.mongodb.database: product-db
```

DIGITALHOUSE

Muito obrigado!