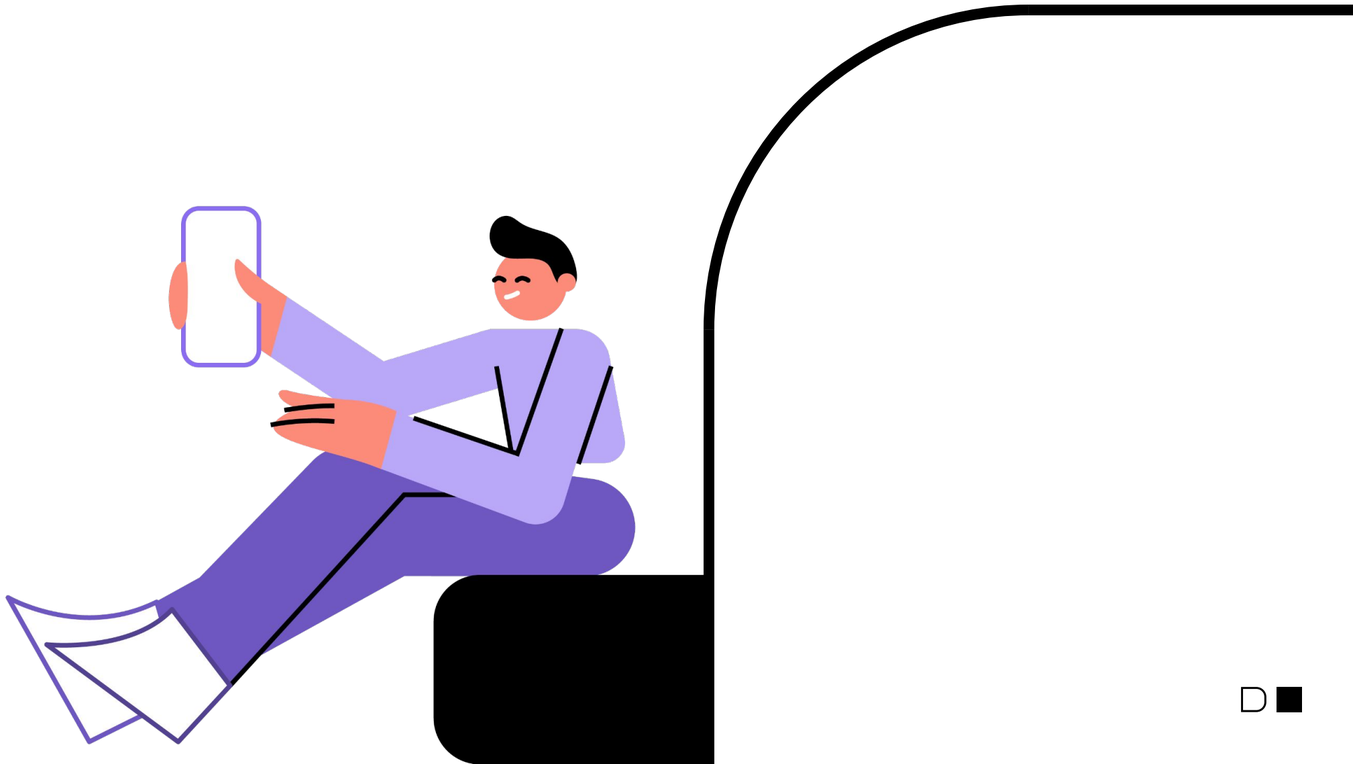
DIGITALHOUSE

Deployando microservicios con Docker

Certified Tech Developer
The Ultimate Degree

Índice



01 Dockerfile

Dockerfile

Lo primero que tenemos que hacer para empezar a ejecutar nuestros microservicios en contenedores son las imágenes Docker. Como sabemos, podemos crearlas mediante los archivos conocidos como “**Dockerfile**”. Veamos un ejemplo de configuración para proyectos en 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 a utilizar la versión 11 del JDK de Java para ejecutar nuestro proyecto en el contenedor. Aquí podemos cambiarla por la versión que usamos para codificar.

02 Indicamos en dónde se encuentra el archivo JAR de nuestro proyecto (típicamente creado con Maven o Gradle). En este ejemplo, el archivo se llama **spring-boot-web**.

03 Copiamos el archivo JAR dentro del contenedor y lo nombramos “**app.jar**”.

04 Ejecutamos el archivo JAR con el comando “**java -jar app.jar”.**

05 Exponemos el puerto 8080 del contenedor. Como cada contenedor se ejecuta en una dirección IP diferente, podemos configurar a todos los proyectos con el mismo puerto.

Construyendo las imágenes

Ahora que ya tenemos creado el archivo Dockerfile, podemos construir la imagen utilizando Docker. Ubicados en la misma carpeta que el Dockerfile, desde la consola, ingresamos:

```
docker build --tag java-docker.
```

Ejecutando las imágenes en contenedores

Una vez creada la imagen podemos ejecutarla en un contenedor con el comando:

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

¡Listo! Tenemos nuestro contenedor ejecutando nuestro microservicio.

02

Deployando microservicios con Docker Compose

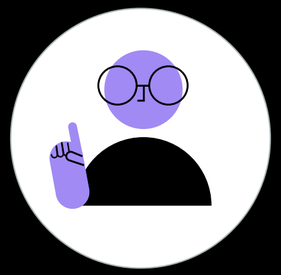
Deployando con Docker Compose

Antes de poder utilizar Docker Compose debemos crear el archivo Dockerfile en cada proyecto que queremos utilizar. Supongamos que tenemos dos microservicios —**product-service** y **user-service**— ubicados dentro de **/microservices**, para utilizar Docker Compose necesitamos crear un archivo de configuración en formato YML. Veamos un ejemplo de esto y qué debemos indicar en cada microservicio:

```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"
```

- Nombre del microservicio que además será utilizado como hostname en la red interna de Docker.
- Indicamos dónde se encuentra ubicado el archivo Dockerfile que será usado para construir la imagen.
- Se indica el límite de memoria (512mb a modo de ejemplo).
- Mapeamos el puerto del contenedor con el host.



A continuación te mostramos los pasos y comandos que serán necesarios ejecutar sobre el archivo **docker-compose.yml** para dejar funcionando los microservicios incluidos en el archivo YML:

01

Crear el archivo JAR de cada microservicio:

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

02

Crear las imágenes Docker de los microservicios con los siguientes comandos: `docker-compose build`

03

Ejecutar los contenedores con el siguiente comando: `docker-compose up -d`

04

Escalando product-service: `docker-compose up -d --scale product-service=2`
En el momento que necesitemos apagar los contenedores, podemos usar el siguiente comando: `docker-compose down`

03

Deployando bases de datos con Docker Compose

Deployando bases de datos con Docker Compose

Supongamos que el microservicio **product-service** utiliza MongoDB como base de datos.
Agregamos en el archivo **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 el ejemplo, usamos la versión 4.4.2 de MongoDB, exponemos el puerto 27017 y configuramos un *healtcheck* para conocer el status de la base de datos en tiempo de ejecución.

Para evitar problemas de conexión con el microservicio **product**, agregamos:

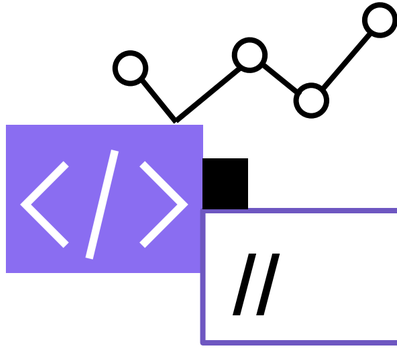
```
depends_on:
  mongodb:
    condition: service_healthy
```

De esta manera, el contenedor que ejecuta el microservicio de productos no se ejecutará hasta que el contenedor esté funcionando correctamente.

El archivo **docker-compose.yml** nos debería quedar de la siguiente manera:

```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
```



Por último, tenemos la configuración en **application.properties** de **product-service**. Cuando desarrollamos y hacemos pruebas, generalmente la base de datos se está ejecutando en localhost, utilizando Docker tenemos que reemplazar localhost por el nombre que utilizamos en el **docker-compose.yml**. En nuestro caso, la conexión con MongoDB quedaría de la siguiente manera:

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

DIGITALHOUSE

¡Muchas gracias!